ANIME EXPLORER

Web Technologies (TWEB) 2025–2026

1 DE FEBRERO DE 2026

BEGOÑA ARANA MÉNDEZ DE VIGO
Università di Torino – Department of Computer Science

Contenido

Introduction

Anime Explorer is a small distributed web application that lets users search and explore a large anime dataset provided as CSV files for the TWEB assignment. The focus of the work is not "feature richness", but correctly applying the techniques from the course: a multi-server architecture, HTTP/Axios communication, and a web client built with HTML/CSS/Bootstrap and Handlebars.

The dataset includes ~28,955 anime records (static information) and several very large dynamic datasets.In this submission, the core end-to-end pipeline is implemented and verified for the anime details dataset; dynamic collections and additional relational tables are prepared but not fully populated due to time constraints.

Distributed architecture and inter-server communication

Solution

The system follows the required 3-server pattern. The browser talks only to a central Express server (Main Server). The Main Server delegates requests via HTTP using Axios to two specialized backends:

- Express + MongoDB server (dynamic data) – port 3002
- Spring Boot + PostgreSQL server (static data) – port 8080

This keeps the Main Server lightweight (no DB queries, no data processing), which is aligned with the requirement that it must be able to serve many concurrent users.

Issues

The main integration issue was CORS when the client attempted to call backend services directly. The fix was to enforce the intended flow (client → main server) and configure CORS middleware consistently across servers for local development origins.

Requirements

This design complies with the assignment by:

- Using a central Express server that only coordinates requests.
- Using Axios/HTTP for all server-to-server communication (no direct code-level coupling).
- Including at least one additional Express server and one Spring Boot server.

Limitations

The architecture is correct but minimal: there is no caching layer and no resilience patterns (retry/circuit breaker). In production, network failures between services would require more robust handling and observability.

Data management and database split (PostgreSQL + MongoDB)

Solution

Data is split according to update rate, as required. Static/relational data (anime details and related entities) is managed in PostgreSQL via Spring Boot + JPA/Hibernate. Dynamic/user-driven data (ratings, profiles, favourites, recommendations) is intended for MongoDB via Mongoose.

In the current submission:

- PostgreSQL is populated with Details.csv (~28,955 rows) and exposed through REST endpoints.
- MongoDB collections/models are defined for the dynamic datasets, but the import scripts are not yet completed.
- Additional relational entities (characters/people) are scaffolded, but their tables are not populated.

Issues

- PostgreSQL authentication error caused by incorrect credentials in application.properties.
- CSV type mismatch (e.g., values like “17086.0” in columns initially modelled as integers). Fixed by adjusting schema types (INTEGER → NUMERIC where needed).
- File permission problems when importing CSV from a cloud-synced directory; resolved by moving files to a local folder readable by the database process.
- MongoDB connection timeouts traced to the service not running on the expected port; resolved by verifying service status and adding connection error handling.

Requirements

Compliance with requirements:

- Static data stored in Postgres; dynamic data planned for MongoDB.
- Separate servers access their own database (Spring Boot → Postgres, Express → MongoDB).
- The Main Server never queries databases directly.

Limitations

Only the Details dataset is fully loaded and queryable end-to-end. For a complete version, the MongoDB import and the remaining Postgres tables (characters/people and association tables) should be populated and exposed through APIs.

Web client (HTML/CSS/Bootstrap/Handlebars)

Solution

The client is implemented with plain HTML/CSS/JavaScript, Bootstrap for responsive layout, and Handlebars for templating (as required). Axios is used from the browser to call the Main Server. The UI provides a search box to query anime titles and renders results with basic cards including title and images.

Issues

A common frontend integration bug occurred during development: the search handler function was not available on the global scope due to a JavaScript export/syntax error. After fixing the syntax issue, the function was correctly bound and search worked as expected.

Requirements

- No framework other than Handlebars/Bootstrap is used.
- Queries are sent to the central Express server.
- Axios is used for HTTP calls.

Limitations

The UI is intentionally simple: there is no authentication, no user sessions, limited filtering, and the anime detail view could be improved (e.g., modal/page with full metadata). Pagination is also missing for large result sets.

Documentation and testing

Solution

Endpoints are documented (Swagger/OpenAPI on the Spring Boot side, and JSDoc-style comments where appropriate on the Node.js side). Basic functional testing was performed with browser DevTools and command-line requests.

Issues

The most relevant testing challenge was verifying the full HTTP chain across three processes. Monitoring the Network tab helped confirm request routing and response payloads during search and health checks.

Requirements

This submission includes a working health check and an end-to-end search flow (client → main server → Spring Boot → Postgres → back).

Limitations

There is no automated test suite yet (e.g., Jest/JUnit). Adding automated tests and CI would improve reliability and grading robustness.

Use of Generative AI

Generative AI tools were used as a support during development, mainly as a “pair programmer” for:

(i) spotting syntax mistakes,
(ii) suggesting small refactorings,
(iii) improving the wording/structure of documentation. All suggestions were reviewed and integrated manually; I tested the code and kept responsibility for architectural decisions, data split decisions, and final implementation.

I did not use AI to bypass the required Axios-based communication pattern, nor to import third-party codebases. Any generated snippets were adapted to match the assignment constraints and course materials.

Conclusions

The project meets the core technical requirements: a 3-server distributed architecture with HTTP/Axios communication, a Postgres-backed Spring Boot service for static data, an Express service intended for dynamic MongoDB data, and a Bootstrap/Handlebars client. The main remaining work is populating the MongoDB datasets and completing the additional Postgres tables for characters/people to unlock richer queries.

Division of work

This was an individual project. All parts (backend, databases, client, integration, documentation and testing) were implemented by me.