Postgres and Chill: Smarter Movie Recommendations

CS 51550-001 Database Systems Fall 2025

1st Panagiotis Papadopoulos
Purdue University Northwest CS dept.
Hammond, IN, USA
papadop@pnw.edu

***Abstract*—This project presents Postgres and Chill, a fully Dockerized movie recommendation application built using a PostgreSQL database like the name suggests, a spring boot/REST API backend, that implements the recommendation engine and performs the basic functions of the app, and lastly a HTML/CSS/JS lightweight frontend. The system loads movie data from the MovieLens dataset and generates user personalized suggestions. The user can also browse through popular and all movies and can rate any movie. Lastly the whole app, as mentioned before, is deployed using Docker Compose to ensure easier portability. This paper highlights and demonstrates the system and its components and evaluates if it satisfies the requirements set in the original proposal.**

***Index Terms*—movies, recommendations, ratings, PostgreSQL**

I. Introduction

What is the worst time of the weekend? Having free time to watch a movie and relax but not knowing what exactly to watch. Of course, streaming platforms provide you with suggestions and personalized recommendations, but only for the films they have in their catalog. We need to think BIGGER! We introduce Postgress and Chill, a movie recommendation platform that returns personalized suggestions from a broader movie database.

The System loads movie data and metadata (users and ratings) from the MovieLens dataset [1] into a PostgreSQL database [3]. Then the HTML/CSS/JS frontend that connects with the Postgres/Spring Boot backend handles the production and delivery of ranked, personalized suggestions for each user, popular movies, and a generic all movies search. Users with no historical data are cold-started with a "rate a few movies" section during the registration process, along with popular movies for their initial recommendations.

The primary goals set in the original project proposal were to provide a PostgreSQL database filled with Movie data(movies, rankings, users), a working backend that returns a list of all movies, returns personalized recommendation data for each user, cold-starts new users, and sorts movies with different filters, a simple frontend web UI where users sign in/up, view the top N recommended movies, and browse popular movies, a Dockerized environment where all the components, front/backend, and DB can run with a simple command, and lastly documentation and demo, setup manual, and examples. All of the elements described in the project proposal have been developed in full and will be explained in greater detail in this report.

II. METHODOLOGY

In this section, we will discuss the overall design, architecture, and logic behind Postgres and Chill. This is a full-stack app that follows a multilayered architecture that we'll discuss in detail. Our methodology was to keep the system as much componenized as possible. Components that are clean and managable and have a distinct role to play. In the followingg subsections we will see each layer and major component of our software.

A. SYSTEM OVERVIEW

Postgres and Chill is made up of three core architectural components, three different layers of our system. The First layer is the one that the user interacts with, the frontend. Then we have the main backend layer, which is, let's say, the brain of our platform, executing the recommendation algorithm, and handling communication between the other two layers. Lastly, we have the database layer, which stores all the movies, users, and their metadata. The whole platform is Dockerized to allow easy deployment and consistent use across different devices, since the ideal use of a full server logic was simply not an option. This architecture is clearly visualized in the following diagram "Fig. 1".

The frontend communicates with the backend through HTTP REST requests, while the backend interacts with the Postgres database via JPA and JDBC. Docker Compose ties all the layers together by managing and starting the 2 containers and providing a shared network. We will discuss all the components in detail. This is a very high-level, quick view of our System.

B. FRONTEND

The Frontend of Postgres and Chill is a fairly simple interface made out of several static HTML pages, each serving

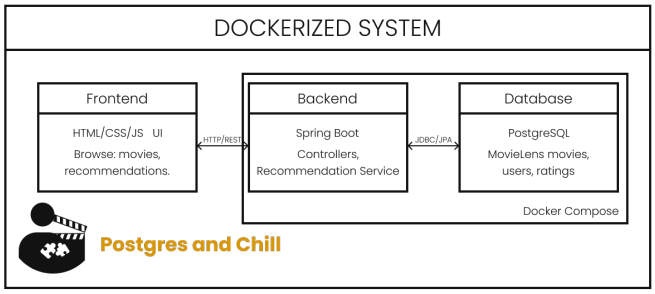

Fig. 1. System Overview

a purpose. The UI is inspired by several platforms that work with movies, mainly Netflix and IMDB, and the colors I chose are inspired by our own PNW logo. The files that make up the frontend are the following. index.html: The home page where users can browse popular and all movies, with no login required. dashboard.html: A user personalized page showing movie suggestions, popular movies, all movies, and past ratings of the user. The user can choose to rate any movie shown here. login.html/register.html: These pages allow users to sign in or create a new account. coldstart.html: A page for new users to rate a few movies for the system to begin personalization. system-down.html/loading.html: Pages for when the system is either down or starting up, these are fallback pages for when the backend is unavailable. These are the main files that make up the frontend and provide the structural layout of the application.

Now that we have mostly seen the CSS part of the files through screenshots, let's discuss a little bit beyond presentation. JavaScript plays a central role in enabling interaction with our backend. The frontend sends and receives JSON data using the Fetch API for all the main processes of the app, like fetching movies, recommendations, sending ratings, and others. There's also a script located in health-check.js, which checks backend availability and sends to the fallback page if the backend is unavailable, ensuring a stable experience. If the backend is down, we don't want the user to perform actions.

Overall, the frontend is intentionally kept simple and clean. The environment is not completely strange to new users, with no major external dependencies, and with fast-loading times.

C. BACKEND

The backend of our application is implemented using Spring Boot [2], which easily allowed us to produce modular REST APIs. We designed and developed in a mostly clean and layered way with clear separation of responsibilities. The distinct parts of our backend are the following. First, we have the controllers, which are the Java classes that expose REST API endpoints and also send and receive requests from the frontend. These endpoints are /api/movies, which returns all movies with details, and /api/genres, which gives us all genres. Then we have /api/ratings, which is used to add or edit a rating for a movie from a user. Also, there's /api/users that handles user operations like looking up a user. Last api/recommendations that returns a list of suggested movies for a specific user. The controllers mainly process the request and usually call a service and return the result.

The next component, which was just mentioned, is the Services. We have two of them, the RecommendationService, which implements the algorithm that matches movies with users, we will discuss that later on. And the Second one, which is MovieService, acts as the middleman between the controllers and the database. It transforms the data through SQL queries set in the repositories and also converts movies to MovieDTOs.

Next, we move to Repositories, which are the 3 components, and they work as Spring Data JPA interfaces [2] used to communicate with the Postgres DB either by custom SQL queries or by ready functions like findAll(). One of those custom queries is used to produce average and weighted ratings for movies. For the weighted score, which is more accurate because it takes into account how many people ranked this movie, we used a very simple Bayesian weighted formula, as IMDB does [5].

Moving on, there's the Models component where we map directly classes to our database tables.

And last, we have the DTOs that were mentioned earlier, Data Transfer Objects, which are used to structure responses and requests that the API will provide.

This is what constitutes our backend, this clear separation allows for clear and maintainable code with classes that have very specific tasks. Overall, Spring Boot is the brain of our App that coordinates everything, from user actions, to database operations, endpoint exposure, and most importantly, recommendations calculations. Authors and Affiliations

D. DATABASE

Our database layer is a very simple PostgreSQL that stores movies, users, ratings, and genres extracted from the MovieLens dataset, specifically the Latest Small: 100,000 ratings, 9,000 movies, 600 users, providing a realistic foundation for our app. Two files handle the setup of our database, the schema.sql which like its name suggests defines the relational schema. And the load_data.sql the loads the data from the movielens csv files to the database tables. Here's the Entity Relationship Diagram to help us understand the schema better "Fig. 2".

E. RECOMMENDATION ENGINE

The most essential part of our application is its recommendation engine, which is responsible for generating personalized movie suggestions for each user. The logic behind it is fairly simple, used in real-world applications like Netflix. It applies a simple form of collaborative filtering [6], instead of relying on movie metadata alone, it uses rating patterns from different users to find similarity between movies("other users also liked") and predict users' possible preferences.

In short, how this works is the system gathers all ratings done by a user (this is the reason we want to cold-start new users) and treats this collection as a vector of that user's

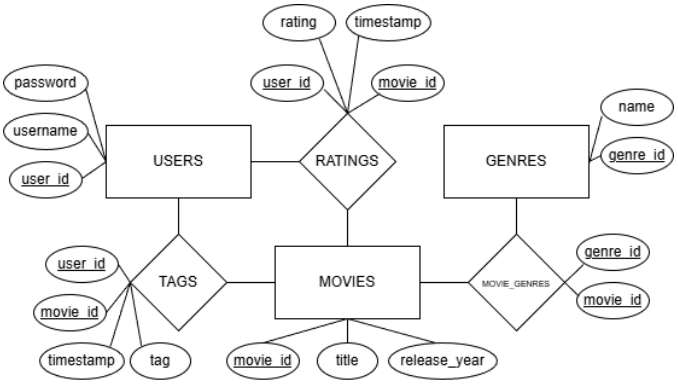

Fig. 2. ER Diagram

preferences. Then, for every movie that hasn't been rated by him, the system checks across other users for ratings patterns, for example, many users also liked. After that, the system goes and for every movie that the user has not yet rated and compares it to every other movie. For our implementation, we used a weighted correlation for that comparison. When that similarity score is produced, it is compared to the movies that the user has rated if it is close to his like (over user's own avg rating) then it gets high score, if close to his dislike (below user's own avg rating) it gets a low score, and these movies are put in a ranked list that can then be displayed as top N at the user's for you section.

F. DEPLOYMENT

Postgres and Chill is deployed using Docker [4] and Docker Compose. This ensures that the app runs more easily and more consistently on different devices without having to download all the needed programs except Docker itself. The platform is orchestrated through the docker-compose.yml. As we saw in the project overview, there are two containers. The PostgreSQL Container starts up and loads the database. Then the Backend Container runs the Java app, connects to postgres container, and serves the REST API routes under /api. Lastly, the fronted does not require a dedicated container because it is served as static HTML. To run our app, a user has to download our project folder and simply double-click the start.bat file. This will start the containers and will trigger the frontend too, with the user not having to deal with any terminal commands. There are more options, like clean-start, which resets the database to the original, opens the console to logs, and of course, the stop.bat, which, as its name suggests, stops the Docker. Thus, having a fully bundled software application that can run on any device.

III. RESULTS AND DISCUSSION

A. TESTING, EVALUATION, AND RESULTS

To ensure that Postgres and Chill worked correctly as a complete end-to-end recommendation platform, we conducted several functional tests, validation of system behavior, and evaluation of the final application. Testing focused on both backend correctness and frontend usability, while evaluation highlighted the system's ability to provide movie suggestions and deliver a stable UX. More specific testing included **User Authentication** (Login/Register), **Movie browsing** (movie lists, popular movies and filtering/searching of those lists), **Ratings** (submission/update), **Database integrity** (preservation of data after down/up of docker), and last the produced **Recommendations** (how good and true are the results).All testing was mostly conducted manually through interactions with the UI, while at the same time checking the REST API endpoits. The final product achieved all set objectives, resulting in a fully functional, containerized recommendation platform. The system serves personalized suggestions based on real movie data and provides users with a clean, intuitive interface inspired by modern streaming and rating platforms. The collaborative filtering algorithm performed effectively across different user profiles, producing meaningful recommendation lists that reflected user rating patterns.

The project successfully demonstrated:

- **A complete recommendation system** integrating frontend, backend, and database layers.
- **A visually appealing and responsive UI** modeled mostly after IMDB and Netflix.
- **Real personalized recommendation** powered by ratings from a real-world data.
- **A fully containerized, reproducible architecture** enabled by Docker Compose.
- **A clean and modular backend design** using Spring Boot, JPA repositories, and DTO-based communication.

Overall, the system demonstrates the feasibility of building a lightweight but realistic recommendation platform using modern development tools, sound database design, and classical collaborative filtering methods.

B. LIMITATIONS AND FUTURE WORK

Although our application successfully implements and satisfies all the requirements set in the proposal, some limitations still remain and set the route for future development and improvement.

To begin, the most notable limitation is the recommendation engine itself. We discussed that it currently relies on a lightweight similarity-based collaborative filtering, but we would like in the future to update that to a smarter engine that incorporates some kind of neural recommender. Next, we have the user authentication system, which, although working, doesn't implement any modern security practices. We'd like to see that shift to something like a JWT authentication system. Next, the User interface could do with some updates too, structural and functional. One such update would be to modernize the framework and switch to something like React. Lastly, the logic behind our app could be expanded to incorporate tags which are currently stored in our database but not used, add functions like custom movie lists (eg, watch later), and an already watched button.

These improvements would enrich the whole user experience and furthermore expand the capabilities of our platform.

IV. CONCLUSION

Postgres and Chill is a project that was meant to demonstrate the implementation of a movie recommendation platform. The final application successfully satisfied the initial requirements outlined in the project proposal. The end result is fully dockerized platform that integrates a postgreSQL database with a spring boot backend and a static frontend, forming a complete full-stack application. Throughout the development a range of skills were applied, and a lot of hands on experience was gained, across database design, backend engineering, docker, and integration. The project also offered valuable insight into the modern recommender systems even if our platform doesn't apply these state of the art models, we got to study them during the planning and design of our app.

REFERENCES

[1] F. M. Harper and J. A. Konstan, "The MovieLens Datasets: History and Context," ACM Transactions on Interactive Intelligent Systems (TiiS), vol. 5, no. 4, 2015.

[2] Pivotal Software, "Spring Boot Documentation," 2023. [Online]. Available: https://spring.io/projects/spring-boot

[3] PostgreSQL Global Development Group, "PostgreSQL Documentation," 2023. [Online]. Available: https://www.postgresql.org/

[4] Docker, Inc., "Docker Documentation," 2023. [Online]. Available: https://docs.docker.com/

[5] IMDb, "IMDb Weighted Rating FAQ," IMDb Help Center. [Online]. Available: https://help.imdb.com (Used as reference for the Bayesian weighted rating formula.)

[6] J. Herlocker et al., "An Algorithmic Framework for Collaborative Filtering," Proceedings of the ACM SIGIR Conference, 1999. (Referenced for standard similarity-based recommender systems.)