System Design Document

Film Owl – Movie Search Engine

EECS3311

Group 17

Table of Contents

---

1. Introduction

Film Owl is a simple and efficient movie search engine that allows users to look up movies, get details, and see results quickly. The app uses the OMDB API to fetch movie data in real-time, and the backend stores these details in a database to make future searches faster. The frontend has a clean interface where users can type their queries and see results, and it's built to handle errors gracefully if something goes wrong.

We used tools like React/Next.js for the frontend, FastAPI for the backend, and PostgreSQL for the database. Everything is containerized with Docker, so it's easy to run on any platform. The goal was to build something functional, modular, and scalable while keeping the design simple and focused on the user experience.

---

2. CRC Cards

The backend and frontend were split into clear responsibilities to make development and debugging easier. For the backend, the `Movie` class is the core, defining how movie records are stored in the database with fields like title, genre, and release date. The OMDB API Client handles external requests to the OMDB API and converts the responses into usable movie objects. Service1 is the heart of the backend, managing all communication between the frontend, database, and external API.

On the frontend side, the components work together to make sure the app is user-friendly. The main component captures user input (like search queries) and sends it to Service1. Another part of the frontend is responsible for displaying search results, including movie titles, release dates, and posters. There's also error handling built into the UI to show messages if something goes wrong, like no results being found or if the API is down.

Movie	
Parent Class: SQLModel **Subclasses**: None	
Responsibilities:	***Collaborators:***
• Represent movie records in the database with attributes like title, release_date, genre, imdb_id, etc. • Serve as a data model for storing and retrieving movie information.	• Service1: Uses this class for database operations (e.g., storing and retrieving movies). • Database: Stores instances of the Movie class.

OMDB API Client	
Parent Class: None **Subclasses**: None	
Responsibilities:	***Collaborators:***
• Query the OMDB API to fetch movie search results and detailed movie information. • Handle external API requests and return parsed JSON data.	• Service1: Calls this client to retrieve movie data. • Movie: Converts JSON data into structured movie objects for database storage..

Service1	
Parent Class: None **Subclasses**: None	
Responsibilities:	***Collaborators:***
• Handle movie search functionality using the /search/{movie_title} endpoint. • Fetch detailed movie data from the OMDB API via helper functions. • Store retrieved movie data in the database, avoiding duplicates. • Provide endpoints for: • Retrieving all movies (/movies). • Creating new movie entries (/movies, POST). • Fetching specific movie details (/movies/{movie_id}). • Root endpoint for testing API availability (/).	• Movie: Used for database operations. • Database: Stores movie data and retrieves records. • OMDB API: External service for fetching movie data. • Frontend: Sends user queries to this service..

Database	
Parent Class: None **Subclasses**: None	
Responsibilities:	***Collaborators:***
• Store movie records using the Movie class. • Provide a database connection for CRUD operations: ◦ Create: Add new movie records. ◦ Read: Retrieve single or multiple movies. ◦ Update: Not currently implemented. ◦ Delete: Not currently implemented.	• Service1: Uses the database for movie storage and retrieval. • Movie: Represents data stored in the database.

Figure 2.1: CRC cards for backend

UI Homepage	
Parent Component: None	
Responsibilities:	***Collaborators:***
• Render the user interface for the movie search engine. • Capture user input through a search bar. • Display search results and movie details fetched from the backend. • Handle errors such as displaying messages for failed API requests	• Service1: Sends search queries and retrieves movie data via API calls . • Users: Interacts directly with the interface to search for movies.

Display Results	
Parent Component: Frontend (UI)	
Responsibilities:	***Collaborators:***
• Render a list of movie search results. • Display details such as title, release date, genre, and poster image. • Handle updates when new search results are retrieved.	• Service1: Supplies the list of movies through API responses. • Search Functionality: Passes data to be displayed.

Search Functionality	
Parent Component: Frontend (UI)	
Responsibilities:	***Collaborators:***
• Capture user input for movie search queries. • Send API requests to Service1 to fetch search results. • Handle loading states while waiting for responses from the backend.	• Service1: Sends search queries and retrieves movie data.

Error Handling	
Parent Component: Frontend (UI)	
Responsibilities:	***Collaborators:***
• Display error messages to users when API requests fail. • Handle cases where no search results are found. • Provide feedback for invalid inputs (e.g., empty search queries).	• Service1: Returns error responses that are displayed on the frontend.

Figure 2.2: CRC cards for frontend

3. System Architecture

The architecture of the app is pretty straightforward. The frontend is the user interface where users type their movie queries, and it's hosted on port 3000. It sends API requests to Service1, which is the main backend component. Service1 handles all the heavy lifting—it fetches data from the OMDB API, processes it, and saves it in the PostgreSQL database. It also retrieves stored data for future queries to avoid hitting the API unnecessarily.

The PostgreSQL database is where all the movie records are saved. It's reliable and ensures that there are no duplicate entries by using `imdb_id` as the unique key. Adminer is also included in the system for database management and debugging during development. Service2 is just a placeholder for now, but we plan to use it for future features, like personalized movie recommendations or user accounts. Finally, the OMDB API is the external service that provides the actual movie data.

The architecture diagram shows how everything connects: the frontend communicates with Service1, which talks to both the database and the OMDB API. Adminer connects directly to the database for management purposes.

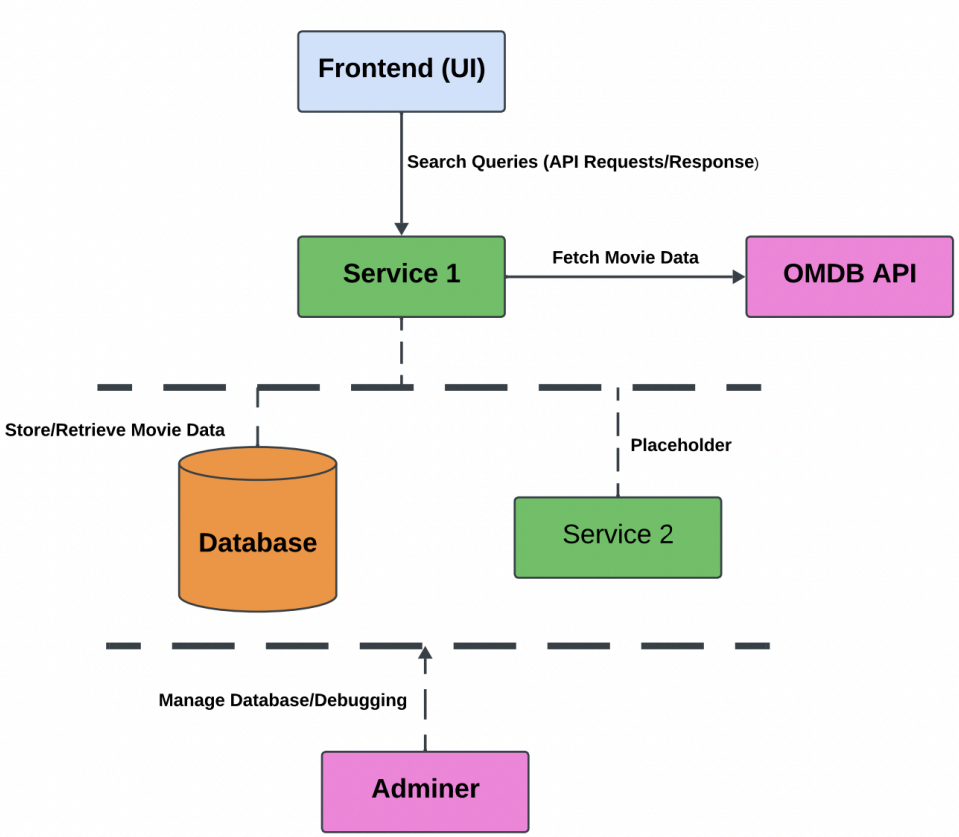

Figure 3.1: Software architecture diagram

---

4. Error Handling Strategy

Error handling was a focus for this project because we wanted the app to be as smooth as possible for users. On the frontend, we handle errors like failed API requests or invalid inputs by showing clear error messages. For example, if no movies are found for a query, the app will let the user know instead of just staying blank.

The backend is designed to handle problems too. It validates all user inputs before processing and logs errors so we can debug easily. If the OMDB API is down or unreachable, the backend sends a meaningful error response back to the frontend. For the database, we set up unique constraints to prevent duplicate records and handle connection failures gracefully.

By layering error handling across the frontend, backend, and database, we made sure the system stays reliable even if something goes wrong at any point.

---

5. Conclusion

Film Owl is a simple but effective movie search engine that balances user experience and functionality. The system is modular, so it’s easy to maintain and add new features in the future. The CRC Cards helped us define the responsibilities of each component, and the architecture diagram makes it clear how everything fits together.

While this version of Film Owl focuses on basic search functionality, we’ve set it up to expand in the future. With modern tools, the system is scalable, fast, and reliable. Overall, we’re happy with the current state of the project, and we’re excited about adding more features in the next sprint.