Movies Watchlist

(SRS Documentation)

December, 2024.

Table of Contents

Table of Figures:

01. Introduction

01. 01. About the project

The purpose of this Software Requirements Specification (SRS) document is to define and describe the requirements for the development of the **Movies Watchlist** application. The **Movies Watchlist** is designed to help users in keeping track of the movies they want to watch in one location, particularly targeting individuals who struggle to remember movies or rely on multiple methods (e.g., sticky notes, spreadsheets) to keep track of them. This application provides an organized and intuitive way for users to manage their personal movie watchlist, making it easier to categorize movies and track their watching progress.

The **Movies Watchlist** offers features that allow users to add new movies, edit their details (e.g., title, genre, description, watchlist order), assign genre recommended by AI, prioritize movies in a watchlist order (e.g., “Next Up”, “When I have time”, “Someday”), and mark movies as watched. Additionally, users can delete movies they no longer want to keep on their watchlist. When a movie is marked as watched, the system sends email notifications (if enabled) and provides AI-driven recommendations for similar movies within the same category. Users can also sort movies based on watchlist order and filter movies based on genre, watchlist order, or status (To Watch/Watched). Users can also create specific watchlists groups depending on their purpose, such as “Watch During Ski Trip” or “Watch When All Exams Finish”.

This document details the functional and non-functional requirements of the **Movies Watchlist** application, including system features, design constraints, and an overview of how the system interacts with its users. It aims to provide developers, testers, and stakeholders with a clear understanding of the system's intended functionality to ensure successful development and deployment of the application.

01. 02. Scope

The scope of the **Movies Watchlist** application is to provide a web-based platform that enables users to manage their personal movie watchlists efficiently. The application targets individuals who struggle to keep track of movies they want to watch and wish to organize them into meaningful categories. The primary goal is to simplify the process of movie management while offering features like AI-powered recommendations and email notifications to enhance user experience.

1. **Movie Management**

- **Add, Edit, and Delete Movies:**
    - Users can add new movies to their watchlist by specifying fields such as title, description, genre and watchlist order (e.g., Next Up, When I have time, Someday).
    - Users can edit existing movie details, including changing the title, description, genre, or watchlist order.
    - Users can delete movies they no longer wish to keep on their watchlist.

2. **Movie Categorization**
    - Users can assign predefined genres to movies (e.g., Action, Drama, Comedy). OpenAI integration will suggest the most suitable genre based on the movie title or description.
    - Genres will be predefined in the system for consistency and simplicity.

3. **Movie Prioritization**
    - Users can assign a custom watchlist order to movies (e.g., Next Up, When I have time, Someday) to help prioritize which movies to watch first.

4. **Completion Tracking**
    - Users can mark movies as "Watched," which updates the movie status.

- Watched movies remain accessible for future reference in the user's watchlist.

5. **Sorting**
   - Movies are automatically sorted alphabetically by title.
   - Additional sorting options include sorting by watchlist order in ascending or descending order.
6. **Filtering**
   - Users can filter movies by genre (e.g., Action, Comedy), watchlist order (e.g., Next Up, When I have time, Someday), and status (e.g., To Watch, Watched), additionally movies can be filtered based on the specific watchlist group that they belong to.
7. **Email Notifications**
   - When a user marks a movie as "Watched," the system sends an email notification (if notifications are enabled).
   - The email includes confirmation of the status change and AI-recommended movies from the same genre.
   - Users can enable or disable email notifications in their settings.
8. **AI Integration**
   - OpenAI API provides genre recommendations for movies based on the title or description during the movie addition process.
   - When a movie is marked as "Watched," email notification is sent to the user with OpenAI API suggestions for five similar movies from the same genre to the user.
9. **User Interface**
   - The application will feature a clean, easy-to-navigate web interface to ensure users can complete basic operations (e.g., adding, editing, deleting movies) with minimal effort.

- The application will adapt seamlessly to different screen sizes, including desktops, tablets, and mobile devices.

10. Data Persistence

- The system ensures that all user data (movies, genre, statuses) is stored securely in a MySQL database.

11. Web-Based Only

- The Movies Watchlist application will be exclusively web-based, accessible via modern web browsers like Google Chrome, Mozilla Firefox, and Microsoft Edge.
- There will be no standalone mobile or desktop app versions. However, the responsive web design ensures a smooth experience on mobile devices.

12. Scalability

- The backend infrastructure will be designed to handle an increasing number of users and movies without compromising performance.
- This scalability ensures that the application can support future growth and feature expansions.

13. Data Security

- User data, including email addresses, will be stored securely, and sensitive data will be encrypted, if exists. The system will follow best practices in data protection to ensure user privacy and security.

The Movies Watchlist application will focus on providing a simple, intuitive, and user-friendly platform for individuals to manage their personal movie watchlists. With features like AI-driven genre recommendations, customizable watchlist priorities, and email notifications, the application ensures users can efficiently organize and keep track of their favorite movies. By leveraging a responsive web interface, Movies Watchlist offers seamless accessibility across devices. The scope also includes planning for scalability to

support an increasing number of users and future enhancements, ensuring the application remains reliable and relevant to user needs.

01. 03. Definitions, Acronyms, and Abbreviations

In this section, key terms, acronyms, and abbreviations used throughout the **Movies Watchlist** project will be defined to ensure clarity and consistency for all stakeholders.

1. **SRS:** Software Requirements Specification - A document that outlines the functional and non-functional requirements of a system.
2. **UI:** User Interface - The part of the application through which users interact with the system.
3. **BE:** Backend - The server-side part of the application responsible for business logic, data processing, and database interactions.
4. **FE:** Frontend - The client-side part of the application responsible for the user interface and user interactions.
5. **Java Spring:** A popular Java-based framework used for building the backend of the application, providing features such as data persistence, and REST API creation.
6. **React:** A JavaScript framework used for building the frontend of the application, providing a component-based architecture to create interactive UIs.
7. **API:** Application Programming Interface - A set of protocols and tools that allow different software components to communicate with each other.
8. **CRUD:** Create, Read, Update, Delete - The four basic operations performed on data in a database or other storage system..
9. **HTTP:** Hypertext Transfer Protocol - The protocol used for communication between clients (browsers) and servers over the internet.
10. **HTTPS:** Hypertext Transfer Protocol Secure - An extension of HTTP that uses encryption (SSL/TLS) to secure data transfer between clients and servers.
11. **REST:** Representational State Transfer - An architectural style used for designing networked applications, commonly used for web APIs.

12. **AI:** Artificial Intelligence - The simulation of human intelligence processes by computer systems, used in this application for recommending movie categories and similar movies.
13. **OpenAI API:** An external service used for recommending movie categories and similar movies based on user input.
14. **Infobip API:** An external service used to send email notifications.
15. **JSON:** JavaScript Object Notation - A lightweight data-interchange format used to send data between the frontend and backend.
16. **Entity:** A single object in the database that represents a specific concept, such as a user or movie.
17. **SQL:** Structured Query Language - A language used to manage and manipulate relational databases.
18. **MySQL:** An open-source relational database management system used to store and manage the application's data.

02. Overall Product Perspective

The **Movies Watchlist** application is designed to provide users with an intuitive platform to save their favorite movies efficiently in one place. The application aims to help individuals manage their watchlist in a organized manner, enabling them to prioritize, categorize, and mark movies as watched while leveraging AI recommendations and email notifications for a seamless user experience.

02. 01. System Context

The **Movies Watchlist** application is a web-based tool comprising a **React** frontend and a **Java Spring** backend. The frontend provides an interactive user interface for managing the movie watchlist, while the backend handles data persistence, business logic, and external API integrations. The system interacts with two major third-party APIs to enhance functionality:

1. **Infobip API:**
   The Infobip API manages email notifications. It allows the system to send confirmation emails when users mark a movie as "Watched" and includes AI-driven recommendations for similar movies.
2. **OpenAI API:**
   The OpenAI API provides AI-driven suggestions for movie genres based on the title and generates recommendations for similar movies to be sent in an email notification when a user marks a movie as "Watched."

02. 02. Dependencies

Java Spring Framework

- The backend is built using the Java Spring framework, which provides features for developing REST APIs, implementing business logic, and managing data persistence efficiently.

React Framework

- The frontend is developed using React, a JavaScript framework that offers a component-based architecture to build a dynamic, responsive, and user-friendly interface.

Infobip API

- This API is used to manage email notifications. It enables the application to send confirmation emails with similar movies recommendations whenever a user marks a movie as "Watched."

OpenAI API

- The OpenAI API enhances the application's functionalities by:
    - Suggesting the most suitable genre for a movie during its addition to the watchlist.
    - Recommending similar movies when a movie is marked as "Watched," helping users discover more content in their preferred genres.

MySQL Database

- A relational database used to store and manage essential application data, including user and movie details, and genre information.

Web Browser

- The application is accessible through modern web browsers, allowing users to interact with their movie watchlists seamlessly on any device, including desktops, tablets, and smartphones.

02. 03. Key Product Functions

1. **Movie Management:** Users can add new movies with fields such as title, description, genre (recommended by AI), and watchlist order (e.g., "Next Up," "When I have time," "Someday"). Users can edit details like title, description, genre, and watchlist order for existing movies. Users can delete movies from their watchlist when they are no longer needed.
2. **Movie Categorization:** Users can assign genres manually or use AI-driven suggestions provided by the OpenAI API. This feature helps users organize their movies effectively.
3. **Completion Tracking:** Users can mark movies as "Watched". When a movie is marked as watched, then the status is updated in the system. An email notification is sent (if enabled) with AI-recommended movies from the same genre.
4. **Sorting:** Movies are sorted alphabetically by title by default. Additional sorting options include sorting by watchlist order (ascending or descending).
5. **Filtering:** Users can filter movies based on genre, watchlist order, status (To Watch/Watched) and/or specific watchlist grouping.
6. **Email Notifications:** Notifications are sent via the Infobip API when a movie is marked as "Watched", unless they are manually disabled in the application. Emails include AI-generated recommendations for movies in the same genre.
7. **AI Integration:** OpenAI API recommends a genre for the movie based on its title. When marking a movie as "Watched," the system uses the OpenAI API to suggest five similar movie.
8. **Responsive User Interface:** The application is designed to work seamlessly on various devices, including desktops, tablets, and smartphones, through a responsive React-based frontend.
9. **Data Persistence and Security:** All user data, including movies, genres, watchlist order, and statuses, are securely stored in a MySQL database.
10. **Scalability:** The backend is designed to handle a growing number of users and movies without compromising performance.

By using Infobip for email notifications and OpenAI for AI-driven recommendations, the Movies Watchlist application offers a simple and effective way to manage movie watchlists. It helps users stay organized and prioritize their favorite movies while providing a smooth and enjoyable experience.

03. Project Requirements

03. 01. Functional Requirements

01. Add a New Movie

As a user, I should be able to add a new movie to my watchlist so that I can keep track of the movies I want to watch.

Acceptance Criteria:

1. Users should be able to add a movie by providing the following fields:
    - Title (required)
    - Description
    - Watchlist Category
    - Genre (e.g., Comedy, Thriller), which can be suggested by AI but editable by the user.
    - Watchlist order ("Next Up," "When I have time," "Someday").
2. The system should confirm the successful addition of a movie.
3. The new movie should immediately appear in the user's watchlist after creation.

02. Edit an Existing Movie

As a user, I should be able to edit a movie in my watchlist so that I can update any details if my preferences or plans change.

Acceptance Criteria:

1. Users should be able to select a movie and modify its details, including:
    - Title
    - Description
    - Watchlist Category
    - Genre
    - Watchlist order

2. The system should confirm the successful update of a movie.
3. The updated movie details should be reflected in real-time in the user's watchlist.

03. Delete a Movie

As a user, I should be able to delete a movie from my watchlist so that I can remove movies I no longer wish to track.

Acceptance Criteria:

1. Users should be able to delete a movie with a single action (e.g., clicking a delete button).
2. The system should prompt the user for confirmation before permanently deleting the movie.
3. The deleted movie should no longer appear in the watchlist after deletion.

04. Mark a Movie as Watched

As a user, I should be able to mark a movie as "Watched" so that I can keep track of the movies I've already seen.

Acceptance Criteria:

1. Users should be able to mark a movie as watched with a single action (e.g., clicking a "Mark as Watched" button).
2. The movie's status should update to "Watched" in the system.
3. If notifications are enabled, the system should send an email confirming the update and include five similar movies recommended by AI.
4. Watched movies should remain accessible in the user's list.

05. Filter Movies

As a user, I should be able to filter movies in my watchlist so that I can view specific movies based on my preferences.

Acceptance Criteria:

1. Users should be able to filter movies based on:
    - Genre (e.g., Comedy, Thriller).
    - Watchlist order ("Next Up," "When I have time," "Someday").
    - Status (To Watch/Watched).
2. The filtered results should be loaded to the page.
3. Users should be able to clear filters to view the full watchlist again.

06. Sort Movies

As a user, I should be able to sort movies in my watchlist so that I can organize them based on specific criteria.

Acceptance Criteria:

1. Movies should be sorted alphabetically by title (ascending) by default.
2. Users should have the option to sort movies by watchlist order (ascending or descending).
3. The list should be updated according to the chosen sorting criteria.

07. AI-Driven Genre Suggestions

As a user, I should receive AI-driven suggestions for movie genres so that I can quickly assign the most suitable genre to a movie.

Acceptance Criteria:

1. The system should automatically suggest a genre for the movie title using the OpenAI API when adding a new movie.
2. Users should have the option to accept or reject the suggested genre.
3. The assigned genre should be displayed as part of the movie details in the watchlist.

08. AI-Driven Movie Recommendations

As a user, I should receive AI-generated recommendations for similar movies when I mark a movie as "Watched" so that I can discover similar movies.

Acceptance Criteria:

1. The system should use the OpenAI API to recommend five similar movies from the same genre.
2. The recommendations should be included in the email notification sent to the user (if notifications are enabled).

09. Email Notifications

As a user, I should receive email notifications when I mark a movie as "Watched" so that I can keep track of my progress and discover new movie recommendations.

Acceptance Criteria:

1. Notifications should be sent via the Infobip API when a movie is marked as "Watched."
2. Emails should include confirmation of the watched status and five other AI-recommended movies.
3. Users should have the option to enable or disable email notifications in their settings.
4. If notifications are disabled, no emails should be sent when marking a movie as watched.

10. Add a New Category

As a user, I should be able to add a new category to my watchlist so that I can organize my movies into meaningful groups.

Acceptance Criteria:

- Users should be able to add a new category by entering a category name.
- Upon successful addition, the system should confirm the creation of the category.
- The new category should immediately appear in the list of available categories.

11. Edit an Existing Category

As a user, I should be able to edit the name of an existing category in my watchlist so that I can rename it if needed.

Acceptance Criteria:

- Users should be able to select a category and change its name.
- Upon successful renaming, the system should confirm the update of the category.
- The updated category name should immediately be reflected in the list of available categories and any movies associated with the category.

12. Delete a Category

As a user, I should be able to delete a category from my watchlist so that I can remove groups that are no longer needed.

Acceptance Criteria:

- Users should be able to delete a category with a single action (e.g., clicking a delete button).
- The system should prompt the user for confirmation before permanently deleting the category.
- Users should have the option to delete the category **only** or delete the category along with all movies in it.
- Upon successful deletion, the system should confirm the deletion of the category.
- If the category is deleted **only**, movies associated with it should remain in the watchlist without an assigned category.

- If the category is deleted **with movies**, those movies should no longer appear in the user's watchlist.

03. 02. Non-Functional Requirements

01. Scalability

The system should be designed to efficiently handle an increasing number of users and movies without compromising performance.

Acceptance Criteria:

1. The layered architecture and design patterns should enable the backend to support a growing user base while maintaining consistent response times.
2. The system infrastructure should accommodate an increase in the number of movies and users.

02. Performance

The system should provide a fast and responsive user experience, ensuring minimal load times for all operations.

Acceptance Criteria:

1. The application should load within 5 seconds for most users.
2. Movie-related operations (e.g., adding, editing, deleting) should complete within 5 seconds under normal load conditions.

03. Security

Description: The system should ensure the security of user data, following best practices for data protection.

Acceptance Criteria:

1. If there is sensitive user data, it should be encrypted both in transit and at rest.

2. User inputs should be validated and sanitized to prevent common security vulnerabilities such as SQL injection.

04. Usability

The system should have an intuitive user interface that is easy to learn and use.

Acceptance Criteria:

1. Users should be able to perform core tasks (e.g., adding, editing, deleting movies) with minimal effort.
2. The UI should provide clear feedback to users for all actions (e.g., success or error messages).

05. Reliability

The system should be reliable and available for users at all times, minimizing downtime.

Acceptance Criteria:

1. The system should have an uptime of 99% or higher.

06. Maintainability

The system should be easy to maintain and update to accommodate future changes and improvements.

Acceptance Criteria:

1. The codebase should follow best practices for readability, including proper documentation and comments.
2. The system should be modular, allowing individual components to be updated or replaced without impacting the rest of the application.

07. Portability

The system should be portable and able to run on different environments without major modifications.

Acceptance Criteria:

1. The application should run on different operating systems (e.g., Windows, Linux, macOS) without requiring major changes.

04. UML Diagrams

01. Use Case Diagram

Actors

1. **User**: Represents both new and returning users of the Movies Watchlist system.
2. **System**: Represents the Movies Watchlist application itself.
3. **Infobip API**: An external service for sending email notifications.
4. **OpenAI API**: An external service for intelligent genre suggestions and movie recommendations.

Use Cases

1. **Add a Movie**: The user can add a new movie by providing its title, description, genre, and watchlist order (e.g., "Next Up," "When I have time," "Someday"). Includes interaction with the OpenAI API for suggesting the genre.
2. **Edit Task**: The user can edit existing movies to update details such as title, description, genre, and watchlist order.
3. **Delete Movie**: The user can delete movies that are no longer needed from their watchlist.
4. **Add Category:** The user should be able to create a new watchlist category, which can later be used to organize movies.
5. **Edit Category Name:** The user should be able to edit the name of an existing category to better organize their watchlist.
6. **Delete Category:** The user should be able to delete a category. The user should also have the option to delete all movies associated with the category or retain them without a category.
7. **Mark Movie As Watched:** The user can mark a movie as "Watched." This includes interaction with the Infobip API to send a notification (if enabled) with AI-recommended similar movies from the same genre.

8. **Filter Movies**: The user can filter movies based on genre, watchlist order, group or status (To Watch/Watched). The system processes the filtering criteria and displays results.
9. **Sort Movies**: The user can besides default alphabetical sorting of movies by title, also sort movies by watchlist order (ascending/descending). The system reorders movies based on the selected sorting parameter.
10. **Send Notifications:** The system sends email notifications to users (if notifications are enabled) when a movie is marked as "Watched," with OpenAI API driven recommendation for similar movies using the Infobip API.
11. **Disable Notifications:** The user can choose to enable or disable email notifications for their account.
12. **View Watchlist:** The user can view their complete watchlist, independent of category, watchlist order or status.
13. **Create AI-Driven Recommendations**: The system recommend user personalized genre suggestions when adding movies and movie recommendations in email notification after marking movies as "Watched," powered by the OpenAI API.
14. **Access Application via Responsive UI**: The user can access the Movies Watchlist on various devices (e.g., desktops, tablets, smartphones) through a responsive web interface.

Relationships

- **Include**:
    - **Add Movie** includes interaction with the OpenAI API for genre suggestions.
    - **Mark Movie as Watched** includes interaction with the Infobip API for notifications.
    - **Create AI-Driven Recommendations** includes interaction with the OpenAI API for genre and movie recommendations.
    - The **Manage Categories** use case should be linked to the **Add Movie** use case, since users must assign movies to categories.
- **Extend**:

- **Mark Movie as Watched** extends **Edit Movie**, as marking a movie as watched is a form of editing.
- The **Manage Categories** use case can extend the **Filter Movies** and **Sort Movies** use cases, as users can filter or sort movies based on categories.

Description of Use Case Diagram Components

2. **User**:
    - Adds, edits, and deletes movies.
    - Adds, edits, and deletes categories.
    - Marks movies as watched.
    - Views and filters their watchlist.
    - Sorts movies.
    - Configures notification preferences.
    - Receives AI-driven recommendations for genres and similar movies.
    - Accesses the application through a responsive web interface.
3. **System**:
    - Suggests genres for movies via the OpenAI API.
    - Sends notifications with OPENAI API movie-recommendations via the Infobip API.
    - Dynamically filters and sorts movies based on user-defined criteria in real-time.
    - Ensures a responsive and seamless user experience across devices.

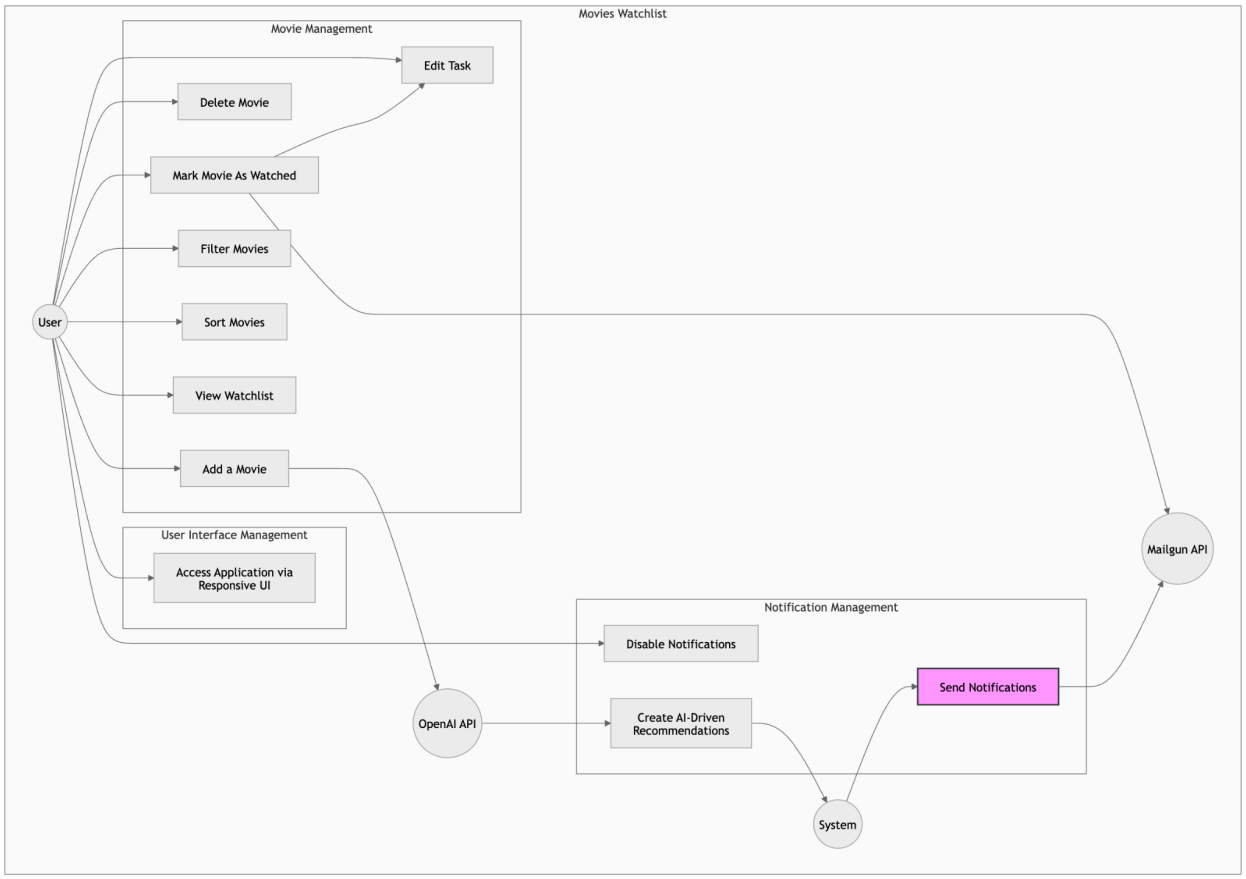

Figure 1. Use case diagram of the Movies Watchlist application

02. Activity Diagrams

1. Activity Diagram for Basic CRUD Operations on the Movie

This diagram focuses on Create, Read, Update, and Delete (CRUD) operations that a user can perform on movies, incorporating AI genre suggestions.

Steps:

01. Start

- User opens the Movies Watchlist application in a web browser.

02. Main Menu

- User navigates to the movie management section.

03. Decision: Choose Operation

- User selects an operation:
    - Add Movie
    - View Movie Details
    - Edit Movie
    - Delete Movie

4. Add Movie Flow

- **Add Movie:**
    - User selects "Add Movie."
    - User provides details such as title, description, genre, watchlist group and watchlist order.
    - The system requests a genre suggestion from the OpenAI API.
    - The suggested genre is displayed and can be accepted or modified by the user.
    - System confirms successful movie creation.

- The new movie appears on the user's watchlist.

5. View Movie Flow

- **View Movie:**
  - User selects "View Movie Details."
  - System displays movie details, including title, description, genre, watchlist order, group and status.

6. Edit Movie Flow

- **Edit Movie:**
  - User selects a movie to edit.
  - User modifies movie details.
  - System confirms the successful update.
  - Movie watchlist is updated to reflect changes.

7. Delete Movie Flow

- **Delete Movie:**
  - User selects a movie to delete.
  - System prompts for confirmation.
  - If confirmed, the system deletes the movie.
  - The movie is removed from the watchlist.

8. End

- User completes the desired CRUD operation and can choose to perform another action or exit the application.

Flow Description:

- **Start Node**: Represents the beginning of the CRUD operations.
- **Decision Node**: Indicates the operation selection—add, view, edit, or delete.

- **Activity Nodes**: Include actions such as "Add Movie," "Edit Movie," "View Movie Details," and "Delete Movie."
- **Flow Arrows**: Connect all the activities, representing the flow of each operation.
- **Merge node:** Second decision nodes where flows of all activities will merge into one flow
- **End Node**: Represents the end of the CRUD operations.

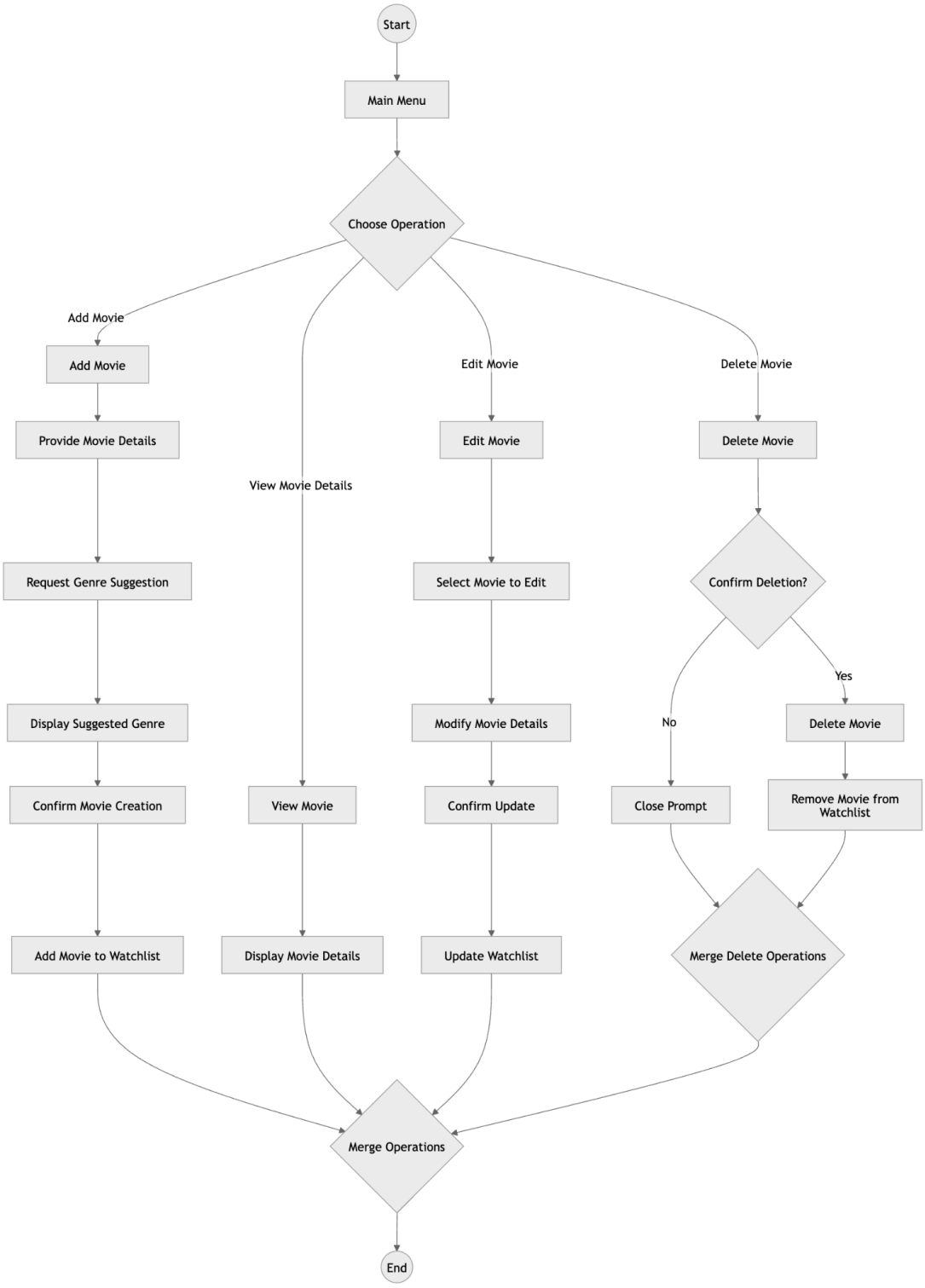

Figure 2. Activity Diagram for Basic CRUD Operations

2. Activity Diagram for Marking Movies as Watched, Filtering, and Sorting

This diagram focuses on Marking Movies as Watched, Filtering Movies, and Sorting Movies, incorporating AI-driven recommendations and notifications.

Steps:

1. Start

- User opens the Movies Watchlist application.

2. Main Menu

- User navigates to the movie management page.

3. Decision: Choose Action

- User selects an action:
    - Mark Movie as Watched
    - Filter Movies
    - Sort Movies

4. Mark Movie as Watched Flow

- **Mark Movie as Watched:**
    - User selects a movie from the watchlist.
    - User marks the movie as "Watched."
    - System updates the status of the movie to "Watched."
    - If notifications are enabled:
        - The Infobip API sends an email notification with AI-recommended similar movies from the same genre.
    - The movie remains accessible in the "Watched" section of the watchlist.

5. Filter Movies Flow:

- **Filter Movies:**
    - User selects filter criteria (e.g., genre, watchlist order, status, watchlist group).
    - System applies the filter and displays the filtered list of movies.
    - User can remove filters to view the full watchlist again.

7. Sort Movies Flow:

- **Sort Movies:**
    - The system automatically displays the movies alphabetically by title as the default sorting.
    - User selects other sorting criteria by watchlist order, either ascending or descending.
    - Once a user selects a different sorting criterion, the system sorts the movies and updates the list in real-time.

8. End:

- User completes the desired action and can choose to perform another or exit the application.

Flow Description:

- **Start Node:** Represents the beginning of activities related to marking movies as watched, filtering, and sorting.
- **Decision Node:** User decides whether to mark a movie as watched, filter movies, or sort movies.
- **Activity Nodes:** Represent actions such as "Mark Movie as Watched," "Filter Movies," and "Sort Movies."
- **Merge Node:** Combines activities back into a common flow for further actions.
- **Flow Arrows:** Represent the transitions between activities.
- **End Node:** Represents the end of the activity flow.

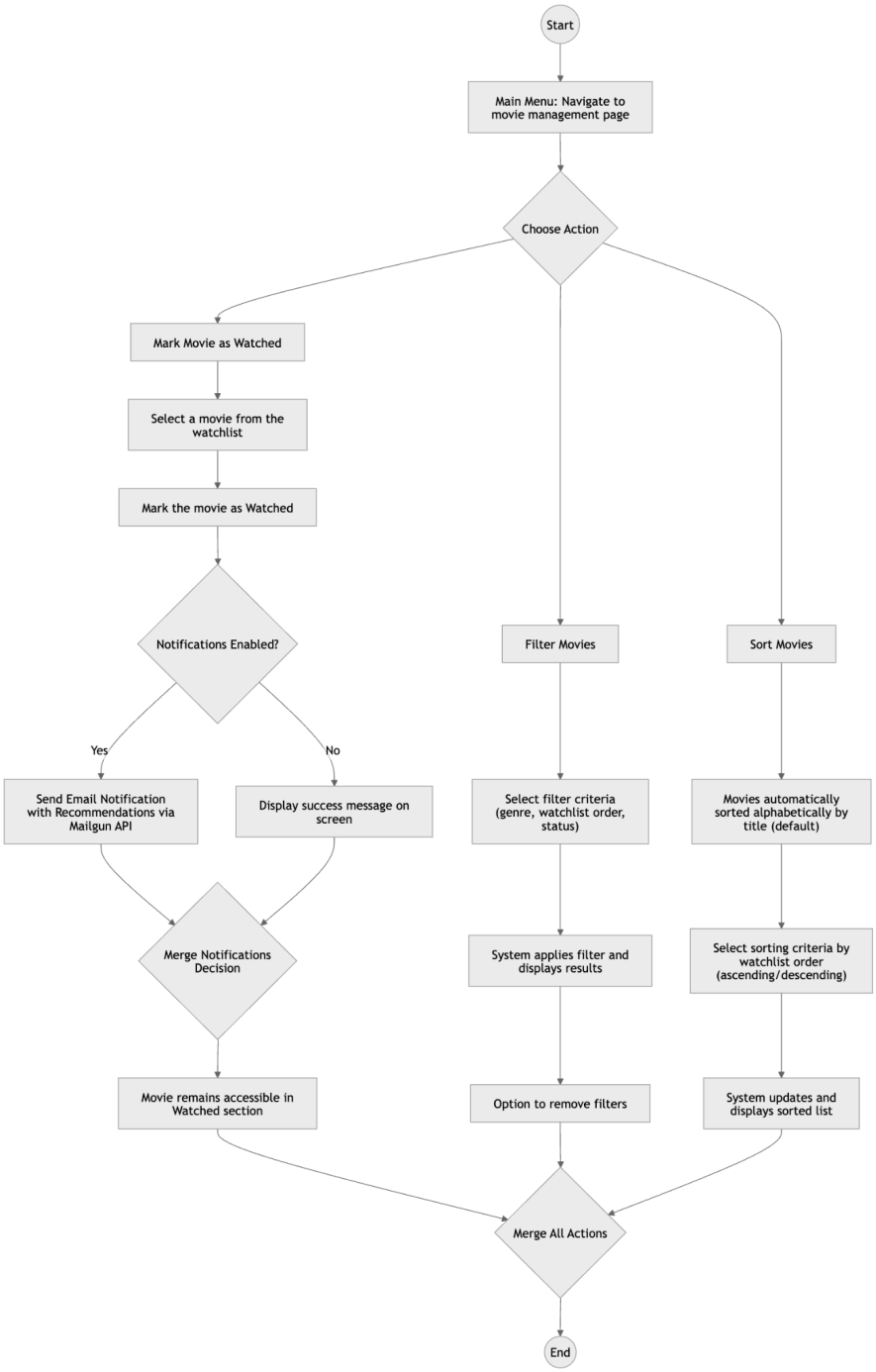
Start
Main Menu: Navigate to movie management page
Choose Action
Mark Movie as Watched
Select a movie from the watchlist
Mark the movie as Watched
Notifications Enabled?
Yes
No
Send Email Notification with Recommendations via Mailgun API
Display success message on screen
Merge Notifications Decision
Movie remains accessible in Watched section
Filter Movies
Select filter criteria (genre, watchlist order, status)
System applies filter and displays results
Option to remove filters
Sort Movies
Movies automatically sorted alphabetically by title (default)
Select sorting criteria by watchlist order (ascending/descending)
System updates and displays sorted list
Merge All Actions
End

Figure 3. Activity Diagram for Task Completion, Filtering, and Sorting

3. Activity Diagram for Basic CRUD Operations on Watchlist Categories

This diagram focuses on Create, Update, and Delete (CRUD) operations that a user can perform on watchlist categories, ensuring flexibility in category management for movies.

Steps:

1. Start

- The user opens the Movies Watchlist application in a web browser.

2. Main Menu

- The user navigates to the category management section.

3. Decision: Choose Category Operation

- The user selects one of the available operations:
  - Add Category
  - Edit Category Name
  - Delete Category

Add Category Flow

- **Add Category**

  The user selects "Add Category."
- **Provide Category Details**

  The user enters the details for the new category (e.g., category name).
- **Confirm Category Creation**

  The system confirms that the category has been successfully created.
- **Add Category to Watchlist**

  The new category is added to the user's list of available watchlist categories.

- **Merge Operations**

  The flow merges into the main flow, and the user can choose to perform another action or exit.

Edit Category Flow

- **Edit Category Name**

  The user selects "Edit Category Name."
- **Select Category to Edit**

  The user selects an existing category to modify.
- **Modify Category Name**

  The user updates the category name.
- **Confirm Update**

  The system confirms the successful update of the category name.
- **Update Category in Watchlist**

  The updated category name is reflected in the user's watchlist.
- **Merge Operations**

  The flow merges into the main flow, and the user can choose to perform another action or exit.

Delete Category Flow

- **Delete Category**

  The user selects "Delete Category."
- **Confirm Deletion**

  The system prompts the user for confirmation before proceeding with the deletion.
- **Decision: Delete Category with Movies?**

  The user decides whether to delete the category along with its associated movies or to delete only the category:

- **Delete Category and Movies**:
  The user confirms deletion of both the category and its associated movies.
  The system removes the category and all movies associated with it from the watchlist.
- **Remove Category Only**:
  The user confirms deletion of the category without deleting the associated movies.
  The system removes only the category while retaining the associated movies in the watchlist.
- **Merge Delete Results**
  Both delete flows merge into a single flow.
- **Close Prompt**
  If the user cancels the deletion, the system closes the prompt without performing any action.
- **Merge Close Prompt**
  The flow of deleting categories merges with the main flow, and the user can choose to perform another action or exit.

End

The user completes the desired category management operation and can choose to perform another action or exit the application.

Flow Description:

1. **Start Node**
   Represents the beginning of category CRUD operations.
2. **Decision Node**
   Indicates the operation selection—Add Category, Edit Category, or Delete Category.

3. **Activity Nodes**

   Include actions such as "Add Category," "Edit Category Name," and "Delete Category."

4. **Decision Node: Delete Category with Movies?**

   A key decision point where the user decides whether to delete the category with or without its associated movies.

5. **Flow Arrows**

   Connect all the activities, representing the flow of each operation.

6. **Merge Nodes**

   Ensure that all activity flows merge into a single flow before proceeding to the next steps or completing the operation.

7. **End Node**

   Represents the end of the category CRUD operations.

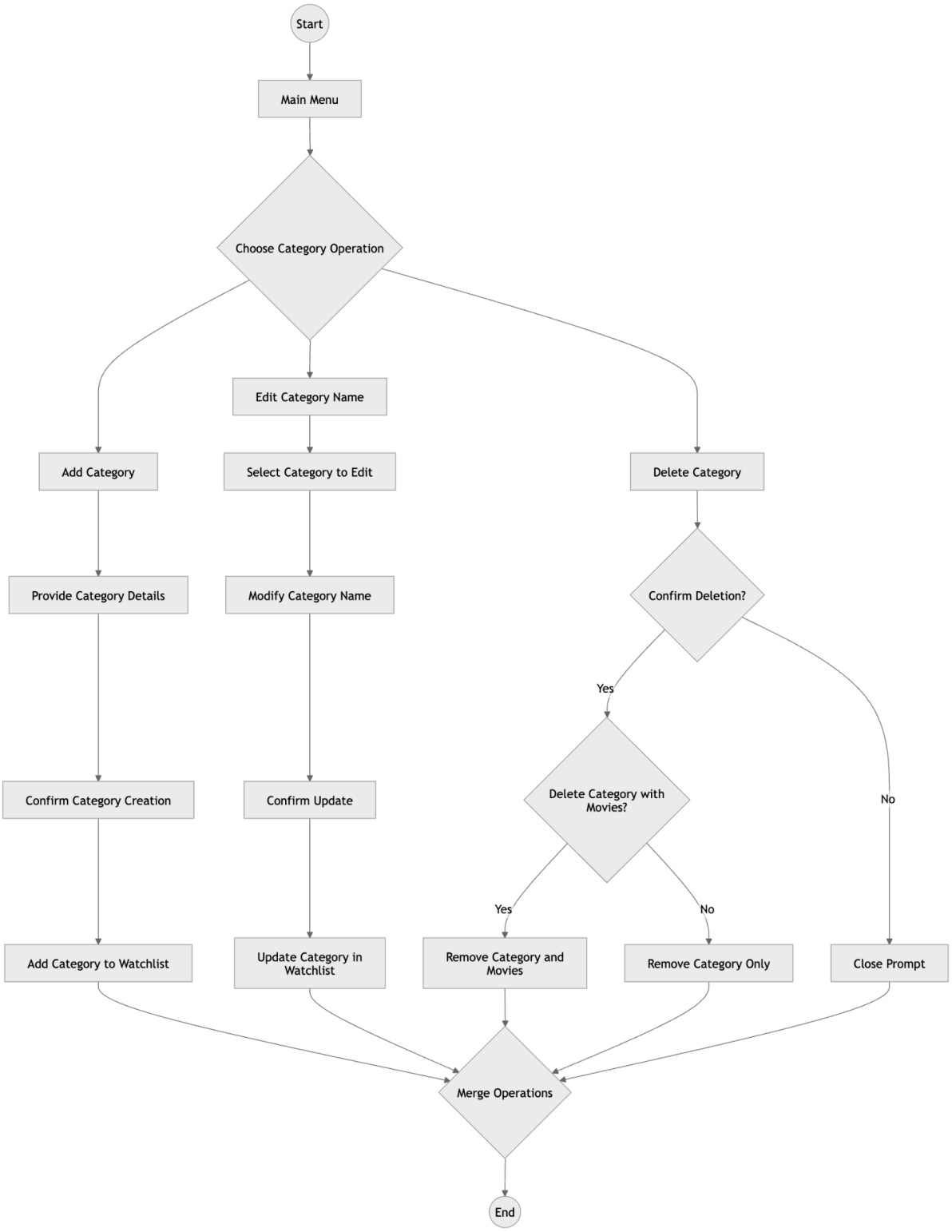
Start
Main Menu
Choose Category Operation
Edit Category Name
Add Category
Select Category to Edit
Delete Category
Provide Category Details
Modify Category Name
Confirm Deletion?
Yes
Confirm Category Creation
Confirm Update
Delete Category with Movies?
No
Yes
No
Add Category to Watchlist
Update Category in Watchlist
Remove Category and Movies
Remove Category Only
Close Prompt
Merge Operations
End

Figure 4. Activity Diagram for Basic CRUD Operations on Watchlist Categories

03. Sequence Diagram

1. Sequence Diagram for the Basic CRUD Operations on the Movie

Sequence Diagram Components

- **Actors:**
  - User: Represents the individual interacting with the application.
  - Frontend (React): Represents the user interface for managing movies.
  - Backend (Java Spring): Handles the server-side logic for movies.
  - Database (MySQL): Stores all movie-related data persistently.
  - OpenAI API: Provides intelligent genre suggestions.
- **Use Cases Covered:**
  - Add Movie
  - View Movie Details
  - Edit Movie
  - Delete Movie

Description

1. Add Movie Sequence:

1. **User:** Enters details such as title, description, group and watchlist order via the Frontend.
2. **Frontend:** Sends an Add Movie Request to the Backend.
3. **Backend:** Processes the request and calls the OpenAI API to fetch a genre suggestion based on the title.
4. **OpenAI API:** Returns the suggested genre to the Backend.
5. **Backend:** Stores the movie details, including the genre, in the Database.
6. **Database:** Confirms successful insertion to the Backend.
7. **Backend:** Sends an Add Movie Success Response to the Frontend.

8. **Frontend:** Updates the watchlist with the newly added movie and displays it to the User.

2. View Movie Details Sequence:

1. **User:** Selects a movie from the watchlist via the Frontend.
2. **Frontend:** Sends a View Movie Details Request to the Backend.
3. **Backend:** Queries the database for the selected movie's details.
4. **Database:** Returns the movie details to the Backend.
5. **Backend:** Sends the Movie Details Response to the Frontend.
6. **Frontend:** Displays the movie details (title, description, genre, watchlist order, status, group) to the User.

3. Edit Movie Sequence:

1. **User:** Modifies movie details (e.g., title, description, genre, group or watchlist order) via the Frontend.
2. **Frontend:** Sends an Edit Movie Request to the Backend with the updated details.
3. **Backend:** Updates the movie details in the Database.
4. **Database:** Confirms the update to the Backend.
5. **Backend:** Sends an Edit Movie Success Response to the Frontend.
6. **Frontend:** Updates the watchlist to reflect the changes and displays the updated details to the User.

4. Delete Movie Sequence:

1. **User:** Selects a movie to delete via the Frontend.
2. **Frontend:** Sends a Delete Movie Request to the Backend.
3. **Backend:** Deletes the movie from the Database.
4. **Database:** Confirms the deletion to the Backend.
5. **Backend:** Sends a Delete Movie Success Response to the Frontend.
6. **Frontend:** Removes the movie from the watchlist and updates the display for the User.

Detailed Sequence Flow

Add Movie:

1. User → Frontend: Enters movie details → Add Movie Request.
2. Frontend → Backend: Send movie details to add.
3. Backend → OpenAI API: Fetch genre suggestion.
4. OpenAI API → Backend: Return suggested genre.
5. Backend → Database: Save movie details with genre.
6. Database → Backend: Confirm movie addition.
7. Backend → Frontend: Movie added successfully.
8. Frontend → User: Display newly added movie.

View Movie Details:

1. User → Frontend: Select a movie → View Details Request.
2. Frontend → Backend: Request movie details.
3. Backend → Database: Query for movie details.
4. Database → Backend: Return movie details.
5. Backend → Frontend: Send movie details.
6. Frontend → User: Display movie details.

Edit Movie:

1. User → Frontend: Modify movie details → Edit Movie Request.
2. Frontend → Backend: Send updated details.
3. Backend → Database: Update movie details.
4. Database → Backend: Confirm movie update.
5. Backend → Frontend: Movie updated successfully.
6. Frontend → User: Display updated movie details.

Delete Movie:

1. User → Frontend: Select a movie → Delete Movie Request.

2. Frontend → Backend: Send delete request.
3. Backend → Database: Remove movie from database.
4. Database → Backend: Confirm deletion.
5. Backend → Frontend: Movie deleted successfully.
6. Frontend → User: Update watchlist to reflect deletion.

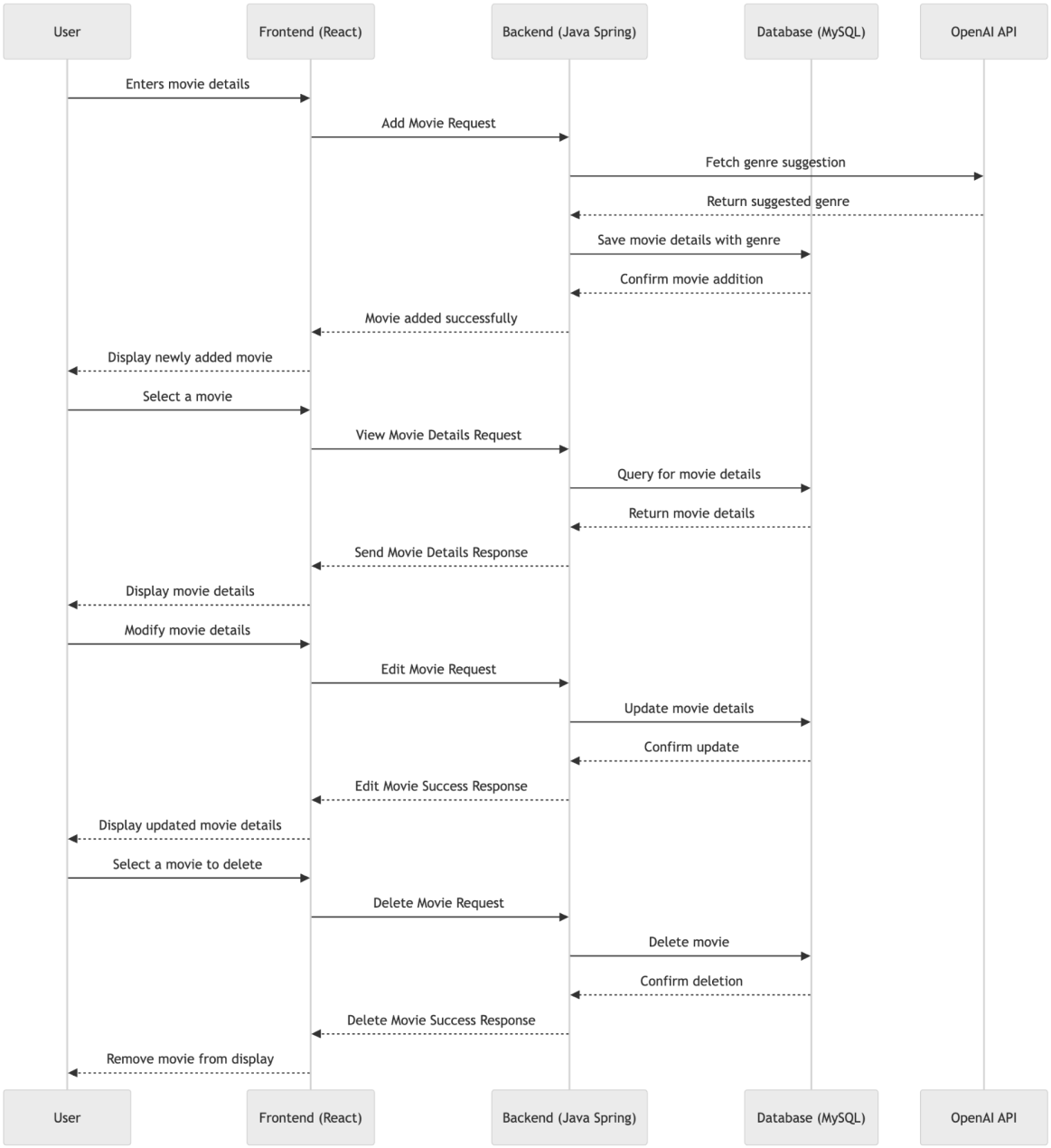

Figure 5. Sequence Diagram for the Basic CRUD Operations on the Movie

2. Sequence Diagram for Marking Movies as Watched, Filtering, and Sorting

Sequence Diagram Components

Actors:

1. **User**: Represents the individual interacting with the application.
2. **Frontend (React)**: Represents the user interface for managing movies.
3. **Backend (Java Spring)**: Handles the server-side logic for movie management.
4. **Database (MySQL)**: Stores all movie-related data persistently.
5. **Infobip API**: Sends email notifications for watched movies.
6. **OpenAI API**: Provides movie recommendations for watched movies.

Use Cases Covered:

1. Mark Movie as Watched
2. Filter Movies
3. Sort Movies

Description

1. Mark Movie as Watched Sequence:

1. **User**: Selects a movie to mark as "Watched" via the Frontend.
2. **Frontend**: Sends a **Mark as Watched Request** to the Backend.
3. **Backend**: Updates the movie's status in the Database to "Watched."
4. **Database**: Confirms the update to the Backend.
5. **Backend**: Requests movie recommendations from the OpenAI API.
6. **OpenAI API**: Returns similar movies from the same genre to the Backend.
7. **Backend**: If notifications are enabled, sends a request to the Infobip API to send an email with the recommendations.

8. **Infobip API**: Sends an email notification to the user.
9. **Backend**: Sends a **Mark as Watched Success Response** to the Frontend.
10. **Frontend**: Updates the movie's status to "Watched" in the watchlist.

2. Filter Movies Sequence:

1. **User**: Selects filter criteria (e.g., genre, watchlist order, or status) via the Frontend.
2. **Frontend**: Sends a **Filter Request** to the Backend with the selected criteria.
3. **Backend**: Queries the Database to retrieve movies matching the filter criteria.
4. **Database**: Returns the filtered movie list to the Backend.
5. **Backend**: Sends the **Filtered Movie List** to the Frontend.
6. **Frontend**: Displays the filtered movies to the User.

3. Sort Movies Sequence:

1. **User**: Selects sorting criteria (e.g., by watchlist order: ascending or descending) via the Frontend.
2. **Frontend**: Sends a **Sort Request** to the Backend with the sorting criteria.
3. **Backend**: Queries the Database to retrieve and sort movies based on the criteria.
4. **Database**: Returns the sorted movie list to the Backend.
5. **Backend**: Sends the **Sorted Movie List** to the Frontend.
6. **Frontend**: Displays the sorted movies to the User. The default sorting (alphabetical by title) is automatically applied when no additional sorting criteria are selected.

Detailed Sequence Flow

Mark Movie as Watched:

1. **User** → **Frontend**: Select a movie → Mark as Watched.
2. **Frontend** → **Backend**: Send Mark as Watched Request.
3. **Backend** → **Database**: Update movie status to Watched.
4. **Database** → **Backend**: Confirm update.
5. **Backend** → **OpenAI API**: Request similar movie recommendations.

6. **OpenAI API** → **Backend**: Return recommended movies.
7. **Backend** → **Infobip API**: Send email notification with recommendations (if notifications are enabled).
8. **Infobip API** → **User**: Send email with recommendations.
9. **Backend** → **Frontend**: Send Mark as Watched Success Response.
10. **Frontend** → **User**: Update movie status to Watched in the watchlist.

Filter Movies:

1. **User** → **Frontend**: Select filter criteria.
2. **Frontend** → **Backend**: Send Filter Request with selected criteria.
3. **Backend** → **Database**: Query for movies matching filter criteria.
4. **Database** → **Backend**: Return filtered movie list.
5. **Backend** → **Frontend**: Send Filtered Movie List.
6. **Frontend** → **User**: Display filtered movies.

Sort Movies:

7. **User** → **Frontend**: Select sorting criteria.
8. **Frontend** → **Backend**: Send Sort Request with sorting criteria.
9. **Backend** → **Database**: Query for sorted movies.
10. **Database** → **Backend**: Return sorted movie list.
11. **Backend** → **Frontend**: Send Sorted Movie List.
12. **Frontend** → **User**: Display sorted movies.

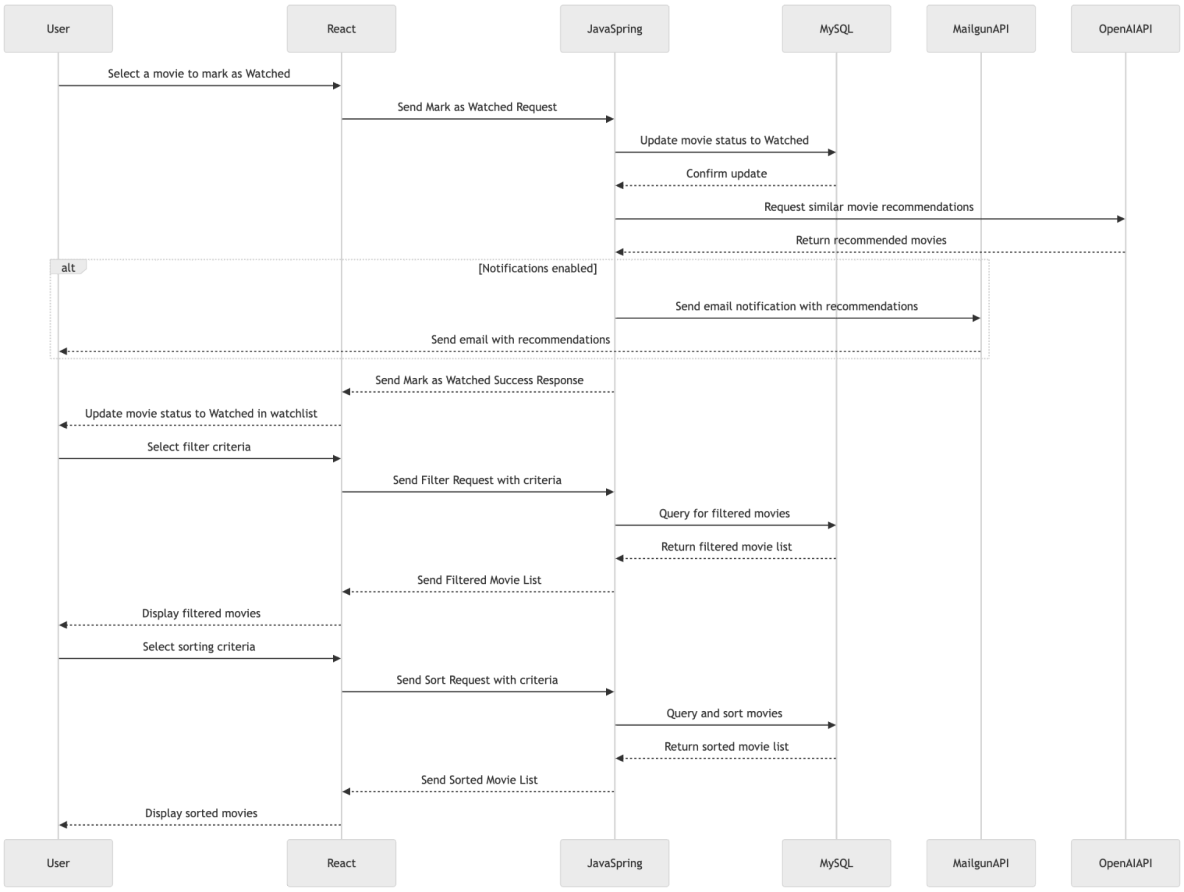

Figure 6. Sequence Diagram for Marking Movies as Watched, Filtering, and Sorting

3. Sequence Diagram for Basic CRUD Operations on Watchlist Categories

Sequence Diagram Components

Actors:

- User: Represents the individual interacting with the application.
- Frontend (React): Represents the user interface for managing categories.
- Backend (Java Spring): Handles the server-side logic for categories.
- Database (MySQL): Stores all category-related data persistently.

Use Cases Covered:

1. Add Category

2. Edit Category Name
3. Delete Category

Description

1. Add Category Sequence:

1. User: Enters a new category name via the Frontend.
2. Frontend: Sends an Add Category Request to the Backend.
3. Backend: Processes the request and stores the category details in the Database.
4. Database: Confirms successful insertion to the Backend.
5. Backend: Sends an Add Category Success Response to the Frontend.
6. Frontend: Updates the watchlist categories and displays the newly added category to the User.

2. Edit Category Name Sequence:

1. User: Selects a category to edit and enters a new name via the Frontend.
2. Frontend: Sends an Edit Category Request to the Backend with the updated details.
3. Backend: Updates the category name in the Database.
4. Database: Confirms the update to the Backend.
5. Backend: Sends an Edit Category Success Response to the Frontend.
6. Frontend: Updates the watchlist categories to reflect the changes and displays the updated name to the User.

3. Delete Category Sequence:

1. User: Selects a category to delete via the Frontend.
2. Frontend: Sends a Delete Category Request to the Backend.
3. Backend: Deletes the category from the Database.
4. Database: Confirms the deletion to the Backend.
5. Backend: Sends a Delete Category Success Response to the Frontend.

6. Frontend: Removes the category from the watchlist and updates the display for the User.

Detailed Sequence Flow

Add Category

1. User → Frontend: Enters category name → Add Category Request.
2. Frontend → Backend: Send category details to add.
3. Backend → Database: Save category details.
4. Database → Backend: Confirm category addition.
5. Backend → Frontend: Category added successfully.
6. Frontend → User: Display newly added category.

Edit Category Name

1. User → Frontend: Modify category name → Edit Category Request.
2. Frontend → Backend: Send updated category name.
3. Backend → Database: Update category name.
4. Database → Backend: Confirm category update.
5. Backend → Frontend: Category updated successfully.
6. Frontend → User: Display updated category name.

Delete Category

1. User → Frontend: Select a category → Delete Category Request.
2. Frontend → Backend: Send delete request.
3. Backend → Database: Remove category from database.
4. Database → Backend: Confirm deletion.
5. Backend → Frontend: Category deleted successfully.
6. Frontend → User: Update category list to reflect deletion.

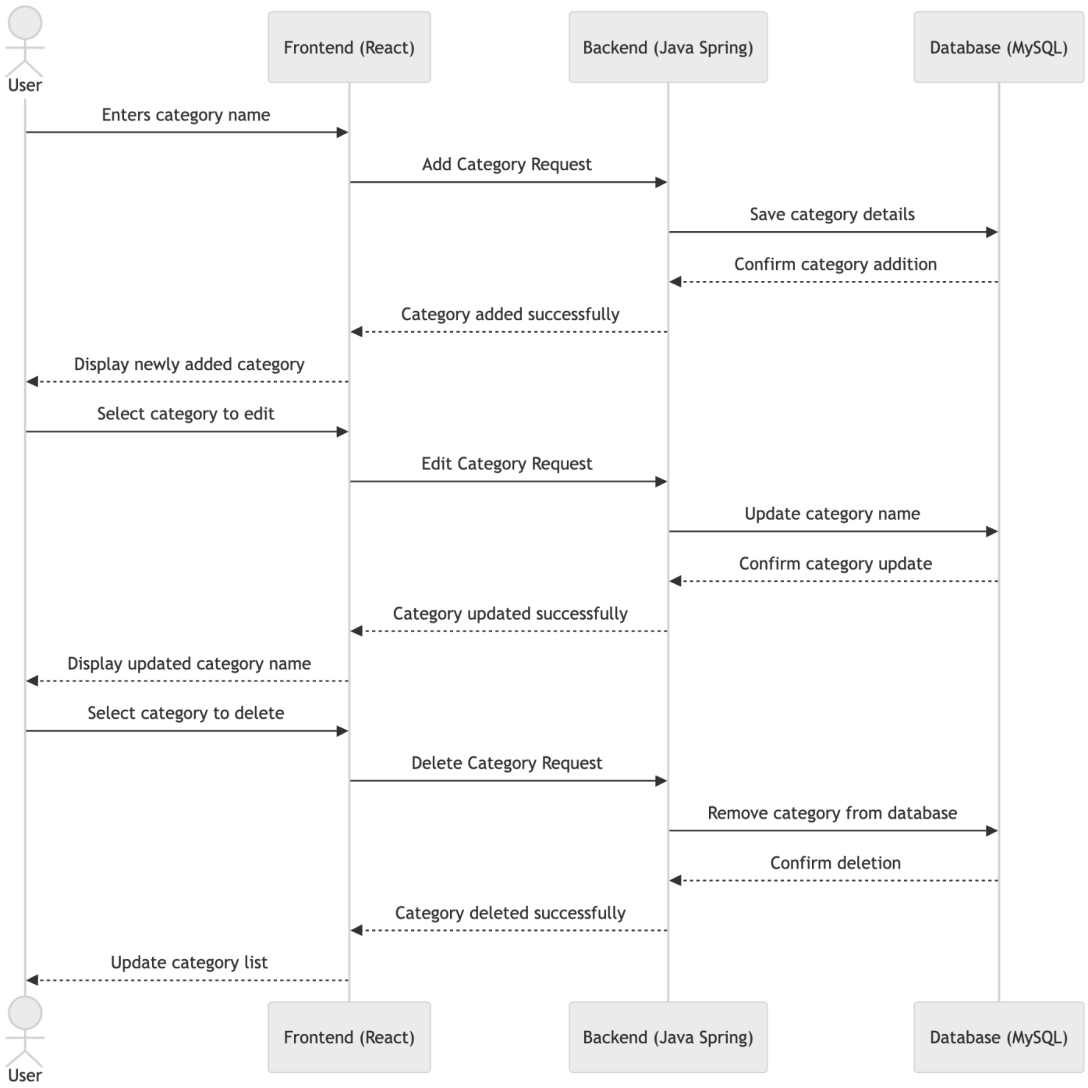

Figure 7. Sequence Diagram for Basic CRUD Operations on Watchlist Categories

04. Class Diagram

The class diagram for the Movies Watchlist application involves the following classes, attributes, methods, and their relationships:

Classes

User

1. **Attributes:**
   a. userId: Integer - Represents the unique identifier for each user

b. email: String - Represents the user email address used for sign up and sign in.
c. emailEnabled: Boolean - Represents whether the user has enabled or disabled notifications.

2. **Methods**:
   a. addMovie(movie: Movie): void - Allows the user to add a new movie to the watchlist.
   b. editMovie(movieId: Integer, updatedMovie: Movie): void - Allows the user to edit an existing movie by specifying the movie ID and updated details.
   c. deleteMovie(movieId: Integer): void - Allows the user to delete a movie by specifying the movie ID.
   d. markAsWatched(movieId: Integer): void - Allows the user to mark movies as watched.
   e. disableNotifications(userId: Integer): void - Allows the user to disable receiving email notifications.
   f. filterMovies(criteria: String): List - Allows the user to filter movies based on certain criteria.
   g. sortMovies(criteria: String): List - Allows the user to sort movies based on criteria such as watchlist order.

Movie

1. **Attributes**:
   a. movieId: Integer - Represents the unique identifier for each movie.
   b. title: String - Represents the title of the movie.
   c. description: String - Represents the description of the movie.
   d. genre: Genre - Represents the genre assigned to the movie.
   e. watchlistOrder: String - Represents the priority of the watching (e.g., Next Up, When I have time, Someday)
   f. status: String Represents the status of the movie (e.g., To Watch/Watched).

2. **Methods**:
    a. assignGenre(genre: Genre): void Assigns a genre to the task.
    b. generateRecommendations(): List<Movie> - Calls the OpenAI API to fetch similar movies.

Genre

1. **Attributes:**
    a. genreId: Integer - Represents the unique identifier for each category.
    b. name: String - Represents the name of a genre.
2. **Methods:**
    a. createGenre(name: String): void - Allows the user to create a new category.
    b. getMoviesByGenre(): List<Movie> - Fetches movies associated with this genre.

WatchlistGroup

1. **Attributes**:
    a. groupId: Integer – Unique identifier for each watchlist group.
    b. name: String – Name of the watchlist group.
2. **Methods**:
    a. addCategory(name: String): void – Adds a new category to the watchlist group.
    b. editCategory(groupId: Integer, newName: String): void – Edits the name of an existing category.
    c. deleteCategory(groupId: Integer, deleteMovies: Boolean): void – Deletes a category, optionally deleting associated movies.
    d. getMoviesPerGroup(): List<Movie> – Retrieves all movies associated with this group.

WatchlistEntry

1. **Attributes**:

   a. entryId: Integer – Unique identifier for each watchlist entry.
   b. movieId: Integer – Identifier for the associated movie.
   c. groupId: Integer – Identifier for the associated watchlist group.
2. **Methods**:
   a. addToGroup(movie: Movie, group: WatchlistGroup): void – Adds a movie to a specific group.
   b. removeFromGroup(entryId: Integer): void – Removes a movie from the group.

Relationships

1. **User - Movie:**
   - Association: A user can add, edit, delete, and manage multiple movies.
   - Multiplicity: One User can have multiple Movies (1..*).
2. **Movie - Genre:**
   - Association: Each Movie is assigned one Genre.
   - Multiplicity: One Genre can be assigned to multiple Movies (1..*).
3. **WatchlistGroup – WatchlistEntry**
   - Association: A watchlist group contains multiple watchlist entries.
   - Multiplicity: One watchlist group can have multiple entries (1..*).
4. **WatchlistEntry – Movie**
   - Association: A watchlist entry references a single movie.
   - Multiplicity: One watchlist entry corresponds to one movie (1..1).

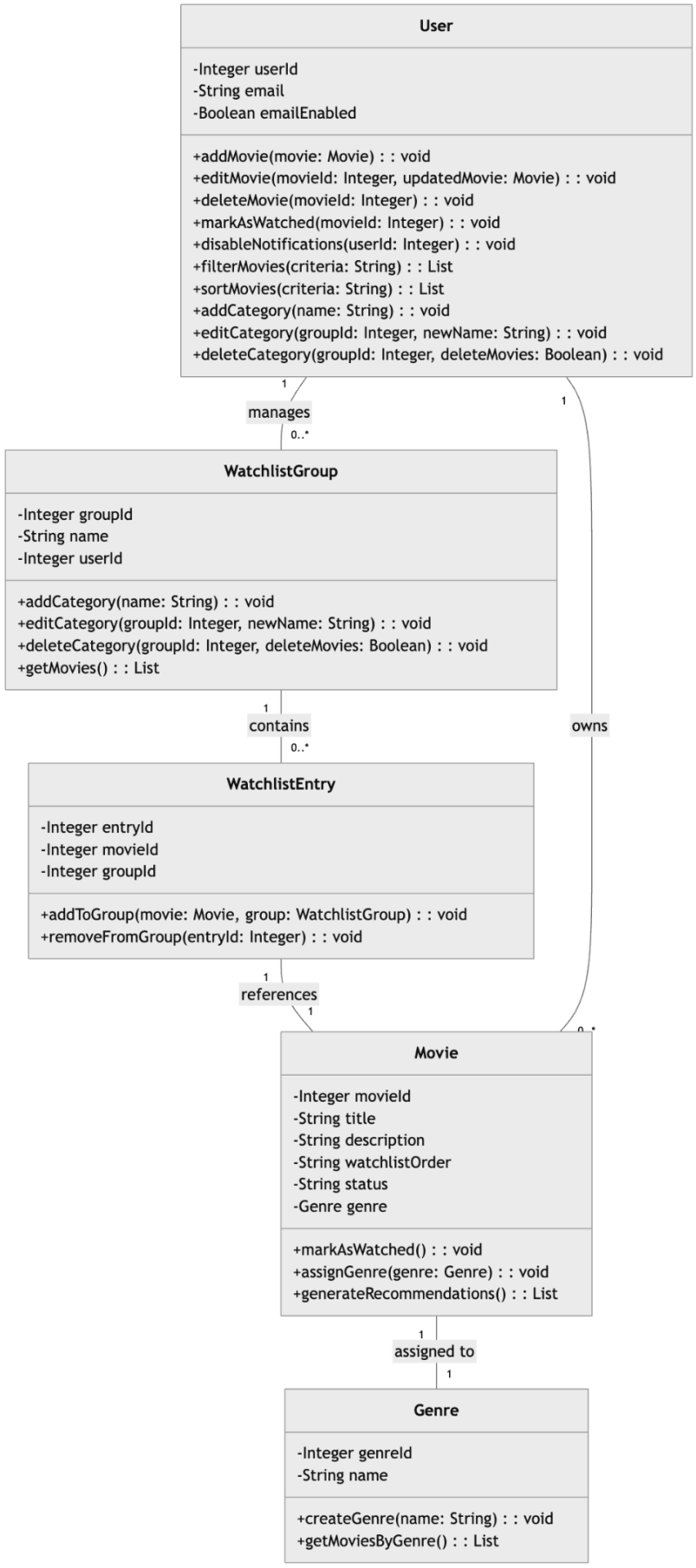
User
-Integer userId
-String email
-Boolean emailEnabled
+addMovie(movie: Movie) : : void
+editMovie(movieId: Integer, updatedMovie: Movie) : : void
+deleteMovie(movieId: Integer) : : void
+markAsWatched(movieId: Integer) : : void
+disableNotifications(userId: Integer) : : void
+filterMovies(criteria: String) : : List
+sortMovies(criteria: String) : : List
+addCategory(name: String) : : void
+editCategory(groupId: Integer, newName: String) : : void
+deleteCategory(groupId: Integer, deleteMovies: Boolean) : : void
1
manages
0..*
1
owns
WatchlistGroup
-Integer groupId
-String name
-Integer userId
+addCategory(name: String) : : void
+editCategory(groupId: Integer, newName: String) : : void
+deleteCategory(groupId: Integer, deleteMovies: Boolean) : : void
+getMovies() : : List
1
contains
0..*
WatchlistEntry
-Integer entryId
-Integer movieId
-Integer groupId
+addToGroup(movie: Movie, group: WatchlistGroup) : : void
+removeFromGroup(entryId: Integer) : : void
1
references
1
Movie
-Integer movieId
-String title
-String description
-String watchlistOrder
-String status
-Genre genre
+markAsWatched() : : void
+assignGenre(genre: Genre) : : void
+generateRecommendations() : : List
1
assigned to
1
Genre
-Integer genreId
-String name
+createGenre(name: String) : : void
+getMoviesByGenre() : : List

Figure 8. Class diagram for Movies Watchlist application

05. Entity Relationship Diagram

This ERD will help represent the data model, showing the key entities, their attributes, and the relationships between them.

Entities and Attributes

1. **User**
    - userId: Integer - Unique identifier for each user (Primary Key).
    - email: String - Represents the email address of the user.
    - emailEnabled: Boolean - Indicates whether the user has enabled email notifications.
2. **Movie**
    - movieId: Integer - Unique identifier for each movie (Primary Key).
    - title: String - Represents the title of the movie.
    - description: String - Represents the description of the movie.
    - genreId: Integer - Represents the genre assigned to the movie (Foreign Key).
    - userId: Integer - Represents the user who owns the movie (Foreign Key).
    - watchlistOrder: String - Represents the priority of the watching (e.g., "Next Up," "When I have time," "Someday").
    - status: String - Represents the status of the movie (e.g., To Watch/Watched).
3. **Genre**
    - genreId: Integer - Unique identifier for each genre (Primary Key).
    - name: String - Represents the name of a genre (e.g., Comedy, Thriller).
4. **Watchlist Group**
    - groupId: Integer - Unique identifier for each watchlist group (Primary Key).
    - name: String - Represents the name of the watchlist group (e.g., Favorites, Must Watch).
5. **Watchlist Entry**
    - entryId: Integer - Unique identifier for each watchlist entry (Primary Key).

- movieId: Integer - Represents the movie added to the watchlist (Foreign Key).
- groupId: Integer - Represents the watchlist group to which the movie belongs (Foreign Key).

Relationships

1. **User - Movie**
   - Relationship: One user can add multiple movies to their watchlist.
   - Cardinality: One-to-Many (User.userId → Movie.userId).
2. **Movie - Genre**
   - Relationship: Each Movie is assigned one genre, but one Genre can be associated with multiple movies.
   - Cardinality: One-to-Many (Genre.genreId → Movie.genreId).
3. **Watchlist Group - Watchlist Entry**
   - Relationship: Each watchlist group can have multiple entries (movies).
   - Cardinality: One-to-Many (WatchlistGroup.groupId → WatchlistEntry.groupId).
4. **Movie - Watchlist Entry**
   - Relationship: Each movie can be added to multiple watchlist groups via entries.
   - Cardinality: One-to-Many (Movie.movieId → WatchlistEntry.movieId).

ERD Components Description

User Entity

- The User entity represents the users of the Movies Watchlist application.
- Each User can manage multiple movies in their watchlist.
- The emailEnabled attribute allows users to toggle email notifications.

Movie Entity

- The Movie entity stores detailed information about each movie, including its title, description, genre, and status.
- Each Movie is linked to a specific User via the userId attribute, ensuring only the owner can manage their watchlist.

- The genreId attribute links each movie to a genre in the Genre table.

Genre Entity

- The Genre entity represents various genres available for classification, such as Comedy, Thriller, and Drama.
- Each Genre can have multiple movies associated with it.

Watchlist Group Entity

- The Watchlist Group entity represents user-created categories for organizing movies into meaningful groups.
- The name attribute stores the title of the group.

Watchlist Entry Entity

- The Watchlist Entry entity represents the association between movies and watchlist groups.
- Each entry links a specific movie to a specific watchlist group, allowing users to assign movies to multiple groups.
- The movieId attribute establishes the link to the corresponding movie, and the groupId attribute links the entry to a specific watchlist group.

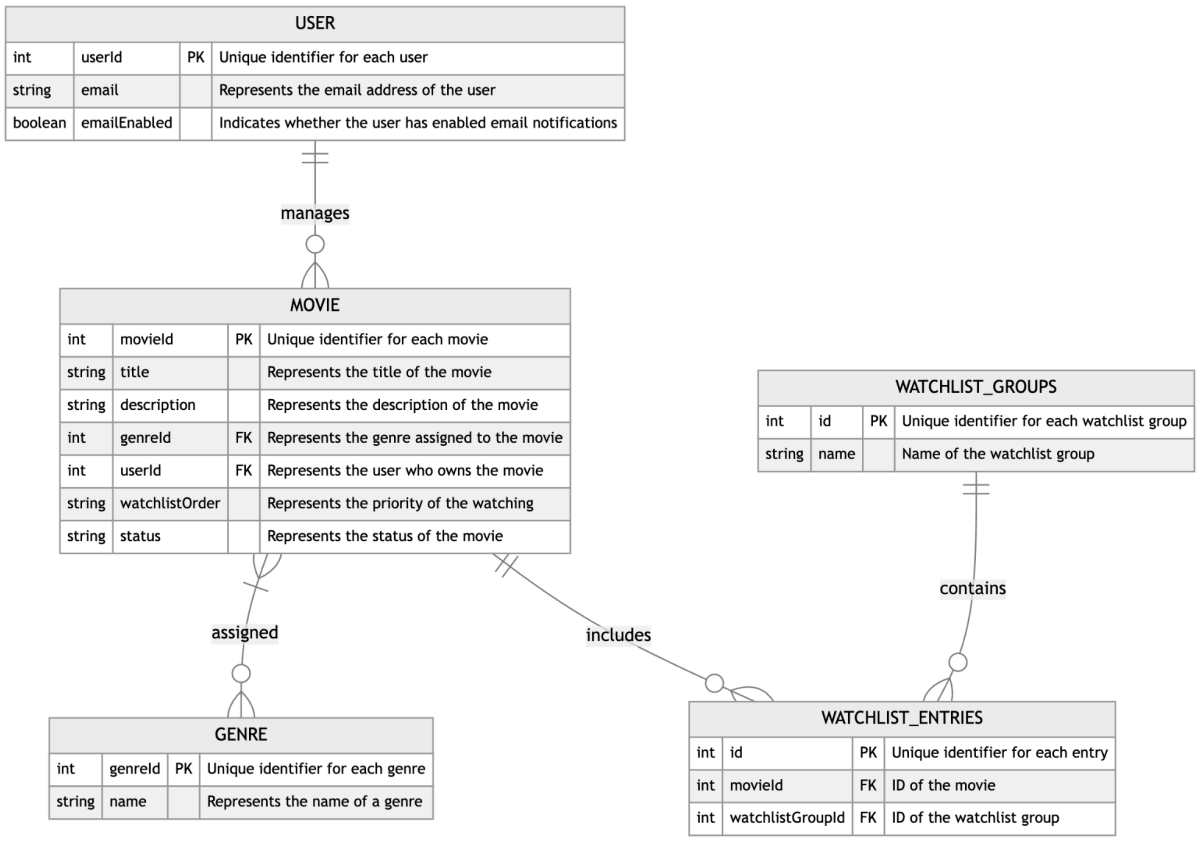

Figure 9. ER Diagram of the Movies Watchlist

06. Design Patterns

In the **Movies Watchlist** application, two essential design patterns are utilized to enhance maintainability, scalability, and efficient resource usage: the **Builder Design Pattern** and the **Singleton Design Pattern**. Below is an explanation of why these patterns are used and their roles in the project.

01. Builder Design Pattern

The **Builder Design Pattern** is used to create complex objects in a step-by-step and convenient way. In the context of the Movies Watchlist application, creating objects like Movie can involve multiple attributes, such as the title, description, genre, watchlist order, and status. Using a builder makes this process more manageable, particularly when not all attributes are required or need to be initialized in a specific order.

Reasons for using Builder Design Pattern

- **Simplifies Object Creation:** The Movie object may have several optional attributes. The Builder Pattern allows the creation of these objects with only the attributes that are required while maintaining flexibility for additional attributes if needed.
- **Improves Readability and Maintainability**: By using a builder, the process of setting attributes for complex objects becomes more readable and less error-prone. This reduces the need for a lengthy constructor with numerous parameters, making the code easier to maintain and extend.
- **Customization**: The Builder Pattern provides a way to easily add new attributes or modify existing ones without affecting the existing code, which makes customization and scaling more straightforward.

02. Singleton Design Pattern

The **Singleton Design Pattern** ensures that a class has only one instance and provides a global point of access to that instance. In the Movies Watchlist application, the Singleton Pattern is used to manage the database connection.

Reasons for using Singleton Design Pattern

- **Efficient Resource Management:** By ensuring that there is only one instance of the DatabaseHandler class, the Singleton Pattern prevents the creation of multiple database connections, which can lead to resource exhaustion and inconsistent states.
- **Avoids Redundant Connections**: Creating multiple connections to the database can be inefficient and can introduce synchronization issues. The Singleton Pattern ensures that a single, shared connection is used throughout the application, reducing overhead and preventing potential issues like data inconsistency.
- **Centralized Access**: The Singleton Pattern provides a single, centralized point of access to the database connection, making it easier to debug, maintain, and ensure the reliability of database operations.

Summary

The Builder Design Pattern is utilized to streamline and simplify the creation of complex objects such as Movie, while the Singleton Design Pattern is used to efficiently manage the database connection, preventing redundancy and ensuring optimal resource usage. Both patterns contribute to a more maintainable, scalable, and efficient application architecture.

07. Interface Requirements

01. GUI The user interface

The user interface for the Movies Watchlist application will be a web-based graphical user interface (GUI). The application is accessible through modern web browsers and provides users with forms for adding, editing, deleting, and managing movies. Features include filtering and sorting movies and toggling notification settings. The interface is designed to be user-friendly, intuitive, and responsive across various devices, including desktops, tablets, and smartphones.

02. CLI

There is no command-line interface for the Movies Watchlist application.

03. REST API Interfaces

- **MailClient Interface**
- The MailClient is an interface that defines methods for sending email notifications. This interface abstracts the implementation of email services, allowing different clients to be used interchangeably. Method defined by this interface is:
    - sendEmail(to: String, subject: String, body: String): void - Sends an email to the specified recipient with the provided subject and body.
    - **Implementation Example**: The InfobipClient class implements the MailClient interface, providing the actual implementation for interacting with the Infobip API. This design allows the email service to be easily replaced with another provider (e.g., SendGridClient) by implementing the same MailClient interface.

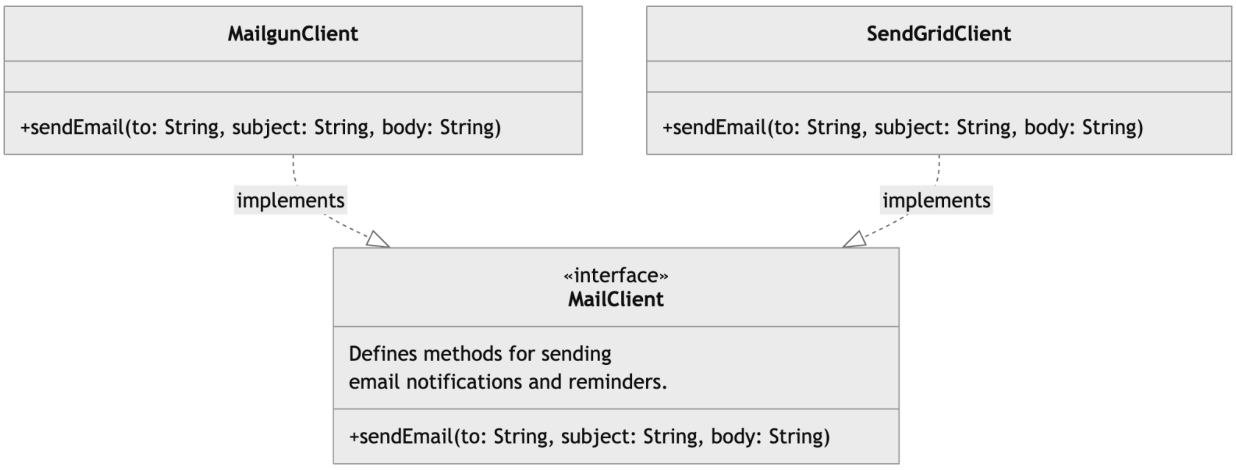

Figure 10. Interface requirements for the MailClient implementation

- **AIRecommendationClient Interface**
- The AIRecommendationClient is an interface that defines methods for fetching AI-driven recommendations. This interface abstracts the implementation of AI-based services, allowing different AI clients to be used interchangeably. Methods defined by this interface are:
    - getGenreSuggestion(movieTitle: String): String - Fetches a genre suggestion for a given movie title using the AI service.
    - getSimilarMovies(movieTitle: String): List<Movie> - Provides a list of similar movies based on the specified movie title.
    - **Implementation Example:** The OpenAIClient class implements the AIRecommendationClient interface, providing the actual implementation for interacting with the OpenAI API. This design allows the AI recommendation service to be easily replaced with another provider (e.g., AWS AI Services) by implementing the same AIRecommendationClient interface.

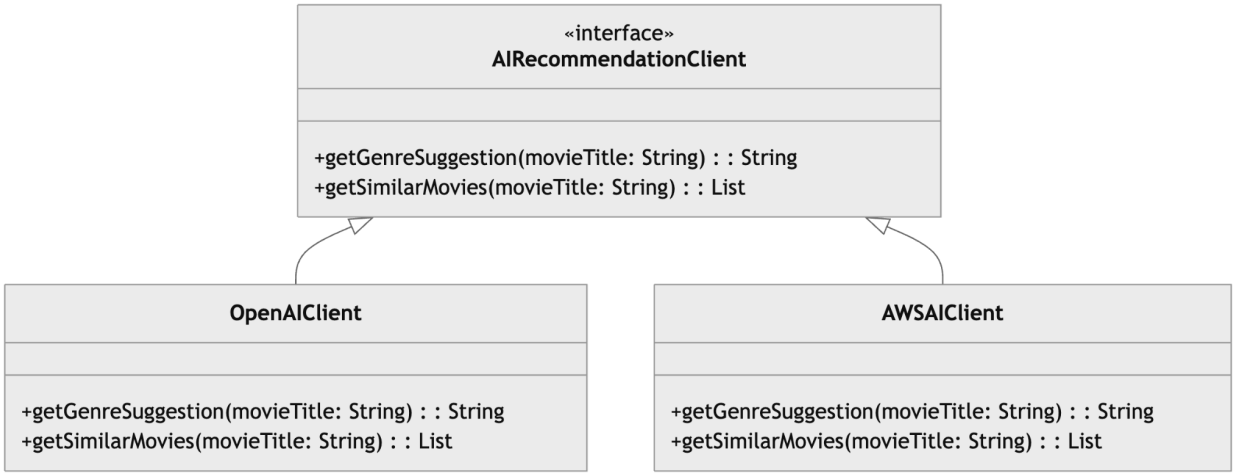

Figure 11. Interface requirements for the AIRecommendationClient implementation

Summary

The interface requirements for the Movies Watchlist application focus on the use of abstract interfaces within the REST API to provide flexibility and extensibility. By using interfaces such as MailClient for email notifications and AIRecommendationClient for AI-driven recommendations, the application can easily swap out different implementations, supporting polymorphic behavior and making the system more maintainable and adaptable to future requirements.

05. Testing

Unit Testing

In the **Movies Watchlist** application, unit tests are essential to validate the core functionalities and ensure the correctness of individual components. Below are simple test cases that can be used to verify proper functionality of individual components.

01. Add a Movie

- **Test Case**: Add a new movie with all required attributes (title, description, genre, watchlist order).

- **Expected Result**: The movie should be successfully added to the watchlist, and all attributes should match the input provided.

02. Edit Movie

- **Test Case**: Edit the genre and description of an existing movie.
- **Expected Result**: The movie's genre and description should be updated correctly without affecting other attributes.

03. Delete a Movie

- **Test Case**: Delete an existing movie from the watchlist.
- **Expected Result**: The movie should be removed from the database and no longer appear in the watchlist.

04. Mark Movie as Completed

- **Test Case**: Mark a movie as "Watched."
- **Expected Result**: The movie's status should change to "Watched," and an email notification (if enabled) should be triggered and sent.

05. Filter Movies by Genre

- **Test Case**: Filter movies by a specific genre.
- **Expected Result**: Only movies belonging to the specified genre should be displayed in the results.

06. Sort Movies by Watchlist Order

- **Test Case**: Sort movies by watchlist order (e.g., "Next Up," "When I have time," "Someday") ascending or descending.
- **Expected Result**: The movies should be sorted based on the specified watchlist order.

07. Singleton Database Connection

- **Test Case**: Verify that the database connection is a singleton (i.e., only one instance is created).
- **Expected Result**: All requests for a database connection should return the same instance.

08. Filter Movies by Status

- **Test Case**: Filter movies by a specific status (To Watch/ Watched).
- **Expected Result**: Only movies that have this status should be displayed in the results.

09. Send Email Notifications

- **Test Case**: Trigger an email notification when a movie is marked as "Watched" and notifications are enabled.
- **Expected Result**: An email should be sent to the user with a list of generated AI-recommended movies.

10. Disable Notifications

- **Test Case**: Disable email notifications for a user.
- **Expected Result**: No email notifications should be sent when the user marks a movie as "Watched."

Summary

The above unit test cases cover the essential functionality of the **Movies Watchlist** application. These tests validate critical operations, such as adding, editing, deleting, and marking movies as watched, along with filtering, sorting, and email notification settings. By implementing these tests, the reliability and correctness of the core features can be ensured while maintaining a simple and beginner-friendly approach to testing.

06. Architectural Design

Create me Architectural Design for this application.

The **Movies Watchlist** application follows a structured architecture for both the backend and frontend components to ensure maintainability, scalability, and ease of use. Below is an overview of the architectural design of the application, including the backend, frontend, and database design.

1. Backend Architecture (Java Spring)

The backend is developed using Java Spring, adhering to a Layered Architecture that promotes separation of concerns. The key layers are:

- **Entity Layer:** Contains entity classes representing the data models (e.g., User, Movie, Genre). These entities are mapped to the database tables using JPA annotations.
- **Repository Layer:** Handles data persistence using Spring Data JPA repositories. Provides built-in CRUD operations without requiring custom SQL queries. Examples: UserRepository, MovieRepository, GenreRepository.
- **Service Layer:** Implements business logic and serves as a mediator between controllers and repositories. Examples: MovieService handles movie-related logic, while UserService manages user-specific operations.
- **Controller Layer:** Provides REST API endpoints to handle HTTP requests and return responses. Examples: MovieController, UserController.

2. Frontend Architecture (React)

The frontend is built using **React** and follows a **component-based architecture**:

- **Components**: Reusable building blocks representing UI elements. Examples: MovieForm, MovieList.

- **Pages**: Main views composed of multiple components. Examples: Dashboard, MovieDetails, SignUp.
- **State Management:** Uses React hooks (e.g., useState, useEffect).
- **API Integration**: Interacts with the backend via **Axios** or **fetch** to make HTTP requests.
- **Styling**: Uses CSS-in-JS libraries or frameworks like Material-UI for responsive and visually appealing designs.

3. Database Layer (MySQL)

The MySQL database serves as the persistence layer, storing all application data. Key tables include:

- **User Table:** Stores user-specific details such as userId, email and emailEnabled.
- **Movie Table:** Stores movie details, including movieId, title, description, genreId, watchlistOrder, and status. userID and genreId are foreign keys in this table.
- **Genre Table:** Stores predefined genres such as Comedy, Thriller, etc. together with their ids.

4. Integration and Data Flow

1. **Frontend to Backend**: The React frontend sends HTTP requests to the Java Spring backend via REST API endpoints. Examples include requests to add, edit, delete, or retrieve movies from the user's watchlist.
2. **Backend to Database**: The backend uses Spring Data JPA to interact with the MySQL database. Data is retrieved, persisted, or updated through repository interfaces.
3. **Backend to External APIs**:
    - **Infobip API**: Used for sending email notifications when a movie is marked as "Watched."
    - **OpenAI API**: Provides AI-driven genre suggestions when adding a new movie and recommendations when marking a movie as watched.

4. **Database to Frontend**:
    - Data from the MySQL database is sent back in JSON format to update the UI dynamically.

Summary

The **Movies Watchlist** application uses a layered backend architecture with Java Spring, a component-based frontend architecture with React, and MySQL as the database layer. This design ensures scalability, maintainability, and a seamless user experience. The integration with external APIs like Infobip and OpenAI further enhances the application's functionality, making it intelligent and user-friendly.

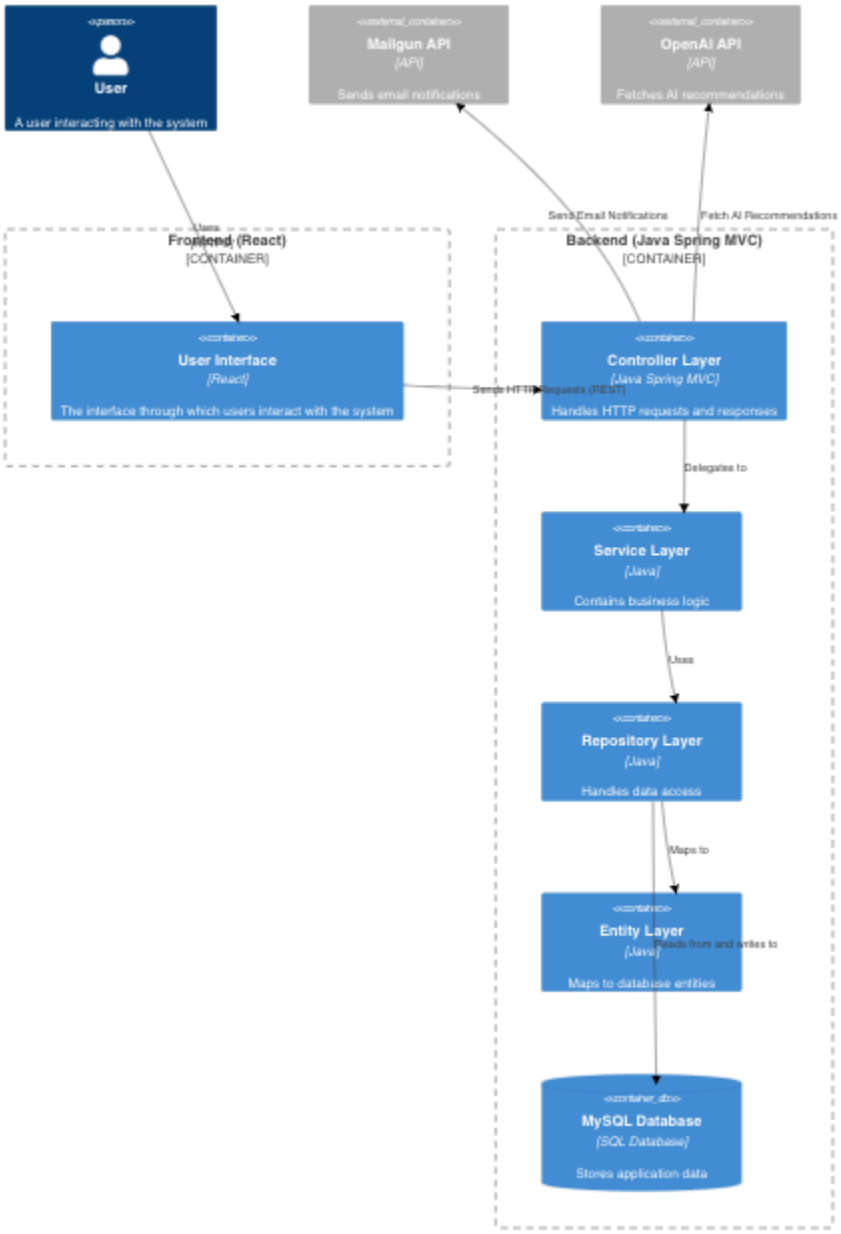

Figure 12. Architectural Design of the Movie Watchlist application

07. Schedule of Implementation

The development of the Movies Watchlist application will follow a phased approach:

Phase 1: Requirements Analysis and Design (Weeks 1-2)

- **Week 1**: Gather and document detailed requirements, including functional and non-functional requirements.
- **Week 2**: Design the architecture, including database schema, system components, and REST API interfaces.

Phase 2: Frontend and Backend Setup (Weeks 3-4)

- **Week 3:** Set up the backend using Java Spring, including database configuration and implementing core services such as MovieService and UserService.
- **Week 4**: Set up the frontend using React, create the basic UI components, and integrate the frontend with the backend API.

Phase 3: Core Feature Implementation (Weeks 5-8)

- **Weeks 5-6**: Implement movie management features, including adding, editing, deleting, and categorizing movies.
- **Weeks 7-8**: Implement movie filtering, sorting, and Infobip integration for email notifications.

Phase 4: Integration and Testing (Weeks 9-10)

- **Week 9**: Integrate the OpenAI API for genre suggestions and complete the integration of all components.
- **Week 10**: Conduct unit testing to ensure system functionalities work properly.

Phase 5: Deployment and Documentation (Weeks 11-12)

- **Week 11**: Deploy the application, set up the database, and ensure access to the API.
- **Week 12**: Finalize user documentation by complete the SRS document.

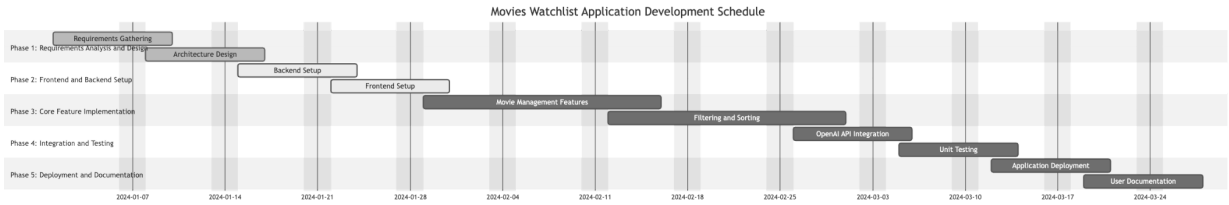

Figure 13. Project Schedule Overview

Summary

The development schedule ensures a structured approach to building the **Movies Watchlist** application, focusing on both functionality and user experience. By breaking down the implementation into phases, the team can deliver a quality product on time, while ensuring that all requirements are met and potential risks are mitigated. This schedule will serve as the roadmap for the successful development, deployment, and delivery of the Movies Watchlist application.

08. Conclusion

The Movies Watchlist application is designed to offer users a convenient and user-friendly platform for managing their personal movie watchlists. By leveraging modern technologies like Java Spring for the backend, React for the frontend, and MySQL for data persistence, the application ensures a seamless and efficient experience. Integration with third-party services such as Infobip for email notifications and OpenAI for AI-driven recommendations enhances its functionality, providing users with personalized genre suggestions and similar movie recommendations.

The application follows a layered architecture to promote scalability, maintainability, and separation of concerns. Design patterns such as the Builder for complex object creation and Singleton for efficient resource management ensure that the system is robust, efficient, and easy to maintain. With features like customizable watchlist order, AI-driven recommendations, filtering, sorting, and the ability to toggle email notifications, the Movies Watchlist application aims to cater to the diverse needs of its users.

The development process is structured into well-defined phases, from requirement gathering to deployment, ensuring timely delivery and adherence to quality standards. By addressing both functional and non-functional requirements, the Movies Watchlist application provides a reliable, scalable, and intuitive solution for users to keep track of their favorite movies and discover new ones with ease. This project stands as an innovative and practical tool for enhancing the way users manage their movie preferences.