

<MovieHub>
Analysis and Design Document : Project

Student: <Moga Diana>
Group: 30435

Table of Contents

Contents

1. Requirements Analysis	4
1.1 Assignment Specification	4
1.2 Functional Requirements	4
1.1.1 Admin Capabilities:	4
1.1.2 User Capabilities:	4
1.3 Non-Functional Requirements	5
1. Validation	5
2. Authentication & Authorization	5
3. Password Security	5
4. ORM Integration	5
5. API Design	5
6. Database	5
7. Separation of Concerns	6
8. User Interface	6
9. Email Notification System	6
10. Quality Assurance and Testing	6
2. Use-Case Model	7
2.1 Description of the "Fetch the data for person/director/actor/movie" use-case:	7
2.2 Description of the "Add a person/director/actor/movie" use-case:	7
2.3 Description of the "Update details for person/director/actor/movie" use-case	8
2.4 Description of the "Delete person/director/actor/movie" use-case	8
2.5 Description of the "Assign actors to movies" use-case	8
2.6 Description of the "See the movies starring each actor" use-case	9
2.7 Description of the "Register" use-case	10
2.8 Description of the "Login" use-case	10
2.9 Description of the "Leave a review to different movies" use-case	10
2.11 Description of the "Add movie to watchlist" use-case	11
3. System Architectural Design	11
3.1 Architectural Pattern Description	11
4. Testing Layer:	13
4. Class Design	14
5. Data Model	16
6. Changes for the project	18
6.1.1 UI Design Overview	18
6.1.2 User Account Page	18

6.1.3	Watchlist Feature	18
6.1.4	Design Pattern: Observer	18

1. Requirements Analysis

1.1 Assignment Specification

The objective of this project is to develop an application that enables administrators to manage and track detailed information about movies, directors, and actors involved in film production. It also allows users to register, log in, browse movie data, leave reviews, and maintain a personalized watchlist. The system captures and stores data about directors, actors, movies, and user reviews, supporting complex relationships such as many-to-many between actors and movies, one-to-many between directors and movies, and one-to-many between movies, users, and their reviews. Additionally, users can add movies to their watchlist to keep track of titles they are interested in, enhancing their personalized experience within the app. If a movie from their watchlist is released they will receive a notification via email.

1.2 Functional Requirements

1.1.1 Admin Capabilities:

1. Log in securely using admin credentials.
2. Create new movie entries with details such as title, genre, release date, director, and synopsis.
3. Add new actors and directors to the database, including names, biographies, and other relevant details.
4. Assign multiple actors to a movie and assign a single director to a movie.
5. Edit or update movie, actor, and director details.
6. Delete movie, actor, or director records from the system.
7. Change if a movie is released or not.

1.1.2 User Capabilities:

1. Register a new user account with unique email and secure password.
2. Log in using valid credentials.
3. Browse and view lists of movies, actors, and directors with detailed information.
4. Leave reviews on movies, including numeric ratings (e.g., 1–5 stars) and text comments.
5. Edit or delete their own reviews.
6. Filter and sort movies by criteria such as genre, release year, rating, and director.
7. Add or remove movies from their personalized watchlist.
8. Receive email notifications when a movie on their watchlist is officially released.
9. View and manage their watchlist in their account dashboard.
10. Log out securely.

1.3 Non-Functional Requirements

1. Validation

- All user inputs must be thoroughly validated on both client and server sides.
 - Registration fields: no empty or invalid data, valid email format, password strength.
 - Review content: no empty text, rating within allowed range (e.g., 1 to 5).
 - Movie, actor, director data: required fields cannot be blank; date formats must be valid.

2. Authentication & Authorization

- Users must authenticate before performing actions like leaving, editing, or deleting reviews.
- Admin users require authentication for CRUD operations on movies, actors, and directors.
- Implement JWT-based authentication for session management.
- Role-based access control must strictly enforce permissions.

3. Password Security

- Passwords must be securely hashed using strong algorithms before storage.
- Plaintext passwords must never be stored or transmitted.

4. ORM Integration

- Utilize an Object-Relational Mapping (ORM) tool (e.g., Hibernate) to manage entity relationships.
- Support complex relationships:
 - Many-to-many between Actors and Movies.
 - One-to-many from Directors to Movies.
 - One-to-many from Movies to Reviews.
 - One-to-many from Users to Reviews.

5. API Design

- Provide a RESTful API with clear and consistent endpoints.
- Enforce role-based access control for all CRUD and user-specific actions.
- Ensure APIs follow best practices for security, statelessness, and error handling.

6. Database

- Use a relational database (e.g., PgAdmin) to store data about movies, actors, directors, users, reviews, and watchlists.
- Support all required relationships and ensure data integrity with foreign keys and constraints.

- Ensure the watchlist functionality is integrated with user accounts and movie release status.

7. Separation of Concerns

- Follow layered architecture principles:
 - Models represent data entities.
 - Controllers handle HTTP requests.
 - Services contain business logic.
 - Repositories manage database interactions.

8. User Interface

- Deliver a clean, user-friendly, and responsive UI.
- Provide a dedicated admin dashboard for managing movies, actors, directors, and users.
- Enable easy browsing, searching, and filtering of movies for users.
- Provide intuitive interfaces for managing watchlists and reviews.

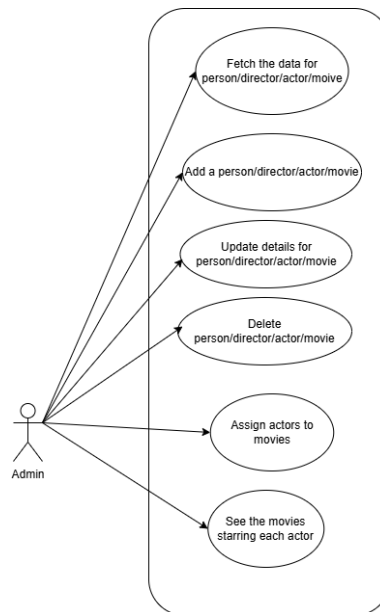
9. Email Notification System

- Automatically send email notifications to users when a movie from their watchlist is released.
- Emails should be timely, informative, and delivered reliably.

10. Quality Assurance and Testing

- **Integration Testing:**
 - Use Spring Boot Test framework to test controller layers.
 - Use MockMvc to simulate HTTP requests and verify API behaviors.
 - Use TestPropertySource to isolate environment and preload fixtures for repeatable tests.
- **Unit Testing:**
 - Use JUnit 5 for unit tests.
 - Use Mockito to mock dependencies in service and repository layers.
 - Verify service logic in isolation.

2. Use-Case Model



-Figure 1 Use-Case Diagram: Admin-

2.1 Description of the “Fetch the data for person/director/actor/movie” use-case:

Use case goal: Fetching all of the recent entries and displaying them on screen using a tables representation

Primary actor: The admin of the application

Main success scenario: Database connection succeeds, data is fetched correctly and is displayed in an intuitive manner within tables.

Extensions:

- Alternate scenario of success: Database connection succeeds, data is fetched correctly, but no entries are contained in the response due to the user never having inserted one, the home screen displays a message suggesting that user should insert his first entry
- Failure: Either the database connection fails, the request times out, the data is corrupt, the graph is unreadable or does not populate with data .

2.2 Description of the "Add a person/director/actor/movie" use-case:

Use case goal: Adding a new person, director, actor, or movie entry into the system.

Primary actor: The admin of the application.

Main success scenario: The admin enters the required details, submits the form, and the new entry is successfully stored in the database. A confirmation message is displayed.

Extensions:

- Alternate scenario of success: The entry is successfully added, but some optional fields were left blank. The system stores the provided information and allows the admin to edit it later.

- Failure: The database connection fails, the request times out, mandatory fields are missing, or the input data is invalid. The system displays an error message prompting the user to correct the issue.

2.3 Description of the "Update details for person/director/actor/movie" use-case

Use case goal: Modifying existing details of a person, director, actor, or movie entry.

Primary actor: The admin of the application.

Main success scenario: The admin selects an entry, updates its details, and submits the changes. The updated information is stored successfully in the database, and a success message is shown.

Extensions:

- Alternate scenario of success: The update is applied, but some optional fields remain unchanged. The system updates only the modified fields and retains previous data for the others.
- Failure: The database connection fails, the request times out, invalid data is entered, or the entry does not exist anymore. The user is notified with an error message.

2.4 Description of the "Delete person/director/actor/movie" use-case

Use case goal: Removing an existing person, director, actor, or movie entry from the system.

Primary actor: The admin of the application.

Main success scenario: The admin selects an entry and confirms its deletion. The system removes the entry from the database and provides confirmation.

Extensions:

- Alternate scenario of success: The deletion succeeds, but the admin had previously assigned the entity to other records (e.g., a director to a movie). The system informs the user and provides options to modify related records.
- Failure: The database connection fails, the request times out, the entry does not exist, or dependencies prevent deletion (e.g., trying to delete a movie with assigned actors). The system notifies the admin of the issue.

2.5 Description of the "Assign actors to movies" use-case

Use case goal: Assigning one or more actors to a movie record.

Primary actor: The admin of the application.

Main success scenario: The admin selects a movie, chooses one or more actors, and confirms the assignment. The database is updated, and a success message appears.

Extensions:

- Alternate scenario of success: The assignment succeeds, but some actors were already linked to the movie. The system prevents duplicate assignments and notifies the admin.

- Failure: The database connection fails, the request times out, the movie or actors do not exist, or the assignment violates system constraints. The system informs the admin of the issue.

2.6 Description of the "See the movies starring each actor" use-case

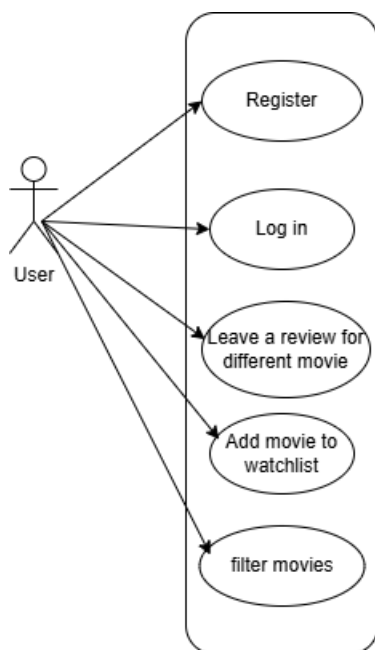
Use case goal: Displaying all movies associated with a selected actor.

Primary actor: The admin of the application.

Main success scenario: The admin selects an actor, and the system fetches and displays a list of movies featuring that actor.

Extensions:

- Alternate scenario of success: The actor exists but is not assigned to any movies. The system displays a message stating that the actor has no associated films.
- Failure: The database connection fails, the request times out, or the actor does not exist. The system displays an error message.



-Figure 2 Use-Case diagram: User-

2.7 Description of the “Register” use-case

Use case goal: Allowing a new user to create an account in the system.

Primary actor: The user of the application.

Main success scenario: The user fills in the required registration details (e.g., username, email, password, age, password) and submits the form. The data is validated and stored successfully.

Extensions:

- Failure: Registration fails due to invalid inputs, missing mandatory fields, password do not match confirm password, or a server/database error. An appropriate error message is displayed to the user.

2.8 Description of the “Login” use-case

Use case goal: Authenticate a registered user to access the application features.

Primary actor: The user of the application.

Main success scenario: The user enters correct login credentials and is authenticated. The system grants access to the user dashboard or home screen.

Extensions:

- Failure: The system detects incorrect credentials, account does not exist, or there's a server issue. An error message is shown.

2.9 Description of the “Leave a review to different movies” use-case

Use case goal: Enable a logged-in user to leave a review or rating for one or more movies.

Primary actor: The user of the application.

Main success scenario: The user selects a movie, writes a review and rating, and submits it. The system validates and saves the review, confirming submission and updating the movie's review section.

2.10 Description of the Filter use-case

Use case goal: Allow the user to filter movies by various criteria (e.g., genre, release year) and order them (e.g., by title, rating).

Primary Actor: The user of the application.

Main Success Scenario: the user accesses the movie listing page. The user selects filter criteria (e.g., genre = "Action", year > 2020). The user selects the ordering method (e.g., by title A-Z or rating high to low). The system applies the filters and ordering to the movie table. The updated, filtered, and sorted list is displayed to the user.

2.11 Description of the “Add movie to watchlist” use-case

Use case goal: Enable a logged-in user to add a movie to their personalized watchlist for future tracking.

Primary actor: The user of the application.

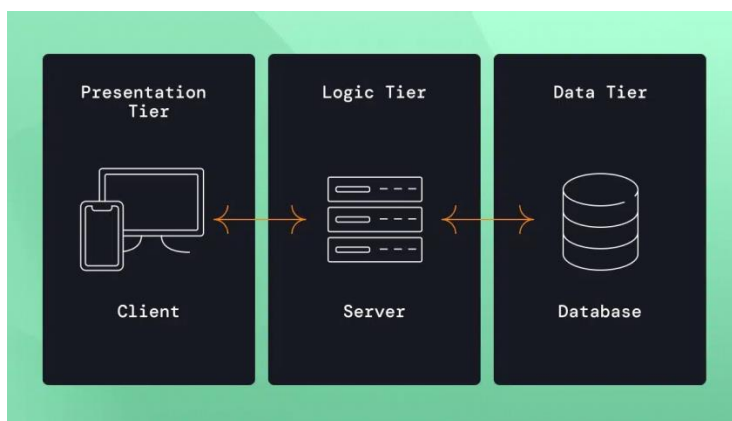
Main success scenario: The user selects a movie from the catalog and clicks the “Add to Watchlist” button. The system confirms the action and stores the movie in the user’s watchlist.

Extensions:

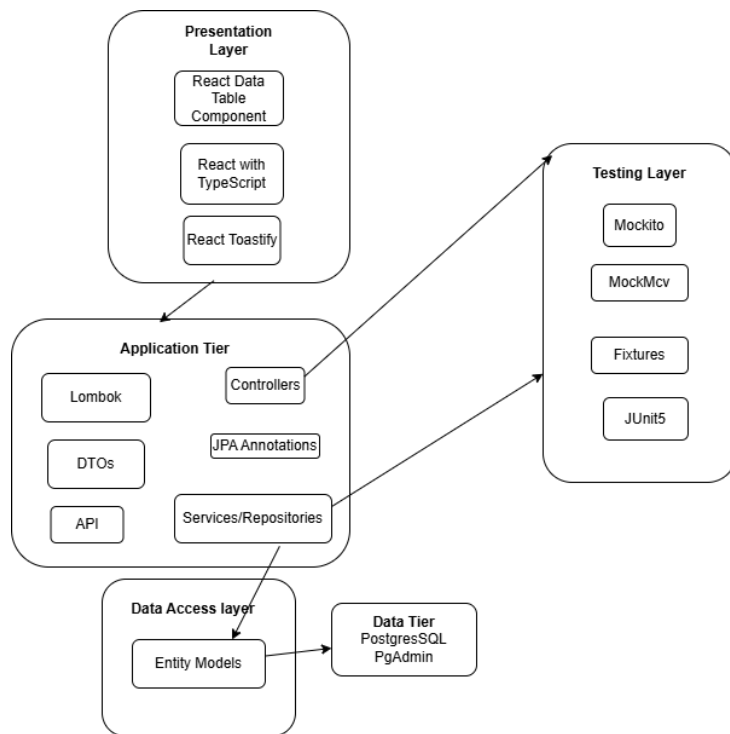
- Failure: The system fails to add the movie due to a session timeout, server error, or database issue. An appropriate error message is displayed to the user.

3. System Architectural Design

3.1 Architectural Pattern Description



-Figure 3 General Architectural Diagram-



-Figure 4 Archtectural Description-

Architectural Pattern Description MovieApp follows a four-tier architecture:

1. Presentation Tier (Frontend):

- React with TypeScript for component-based UI development and type safety.
- React Data Table Component for dynamic table rendering and data handling.
- React-Toastify for user-friendly notifications and alerts.

2. Application Tier (Backend):

- Spring Boot for modular, scalable API development.
- Spring Security with JWT for authentication and authorization.
- DTOs (Data Transfer Objects) for structured data exchange.
- Lombok to reduce boilerplate code in models and services.
- Spring Data JPA with Hibernate for database access and persistence.
- JPA (Jakarta Persistence API) Annotations: these annotations define how the class is mapped to a database table.
- The application includes an email service for sending notifications. This is implemented using Spring Mail, with JavaMailSender to send SimpleMailMessage objects. The EmailService component handles the process of constructing and sending emails (e.g., for password reset). The MailBody record encapsulates the email content (recipient, subject, and text).

3. Data Tier (Database):

- PostgreSQL for relational data storage.
- pgAdmin for database management and administration

4. Testing Layer:

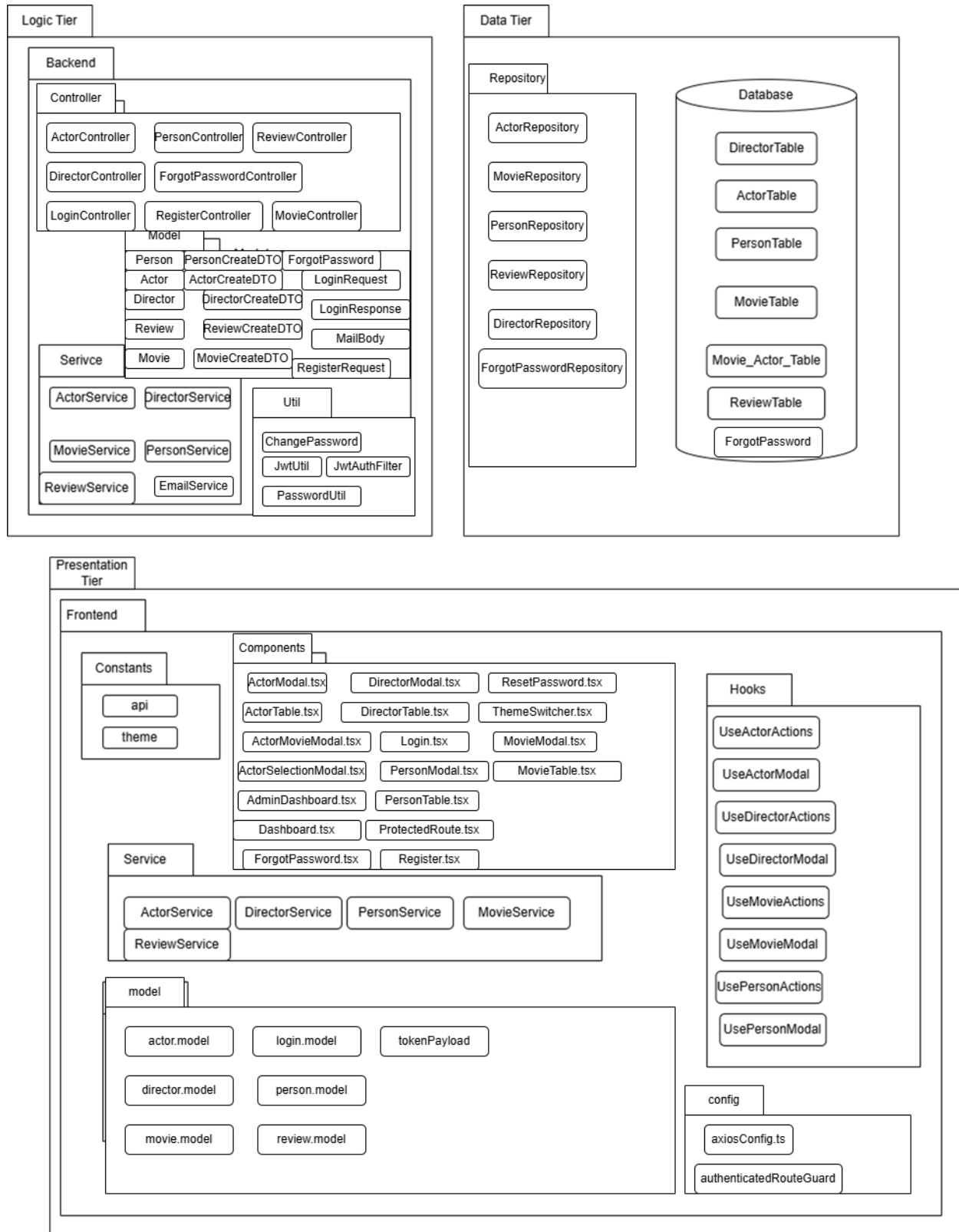
Integration Testing:

- **Spring Boot Test** framework is used to test controller layers in an application context.
- **MockMvc** is employed to simulate HTTP requests and validate controller behavior.
- **TestPropertySource** provides a separate configuration (application-test.properties) to isolate test data and environment.
- **Fixtures** (JSON files) are used to preload test data for consistent and repeatable test scenarios.
- **Transactional tests** ensure that database state is rolled back after execution, maintaining isolation.

Unit Testing:

- **Mockito** is used to mock repository and service dependencies for isolated testing of service layer logic.
- **JUnit 5** is used as the main testing framework.
- **Assertions** validate expected outcomes, while **Mockito verification** ensures interactions with dependencies occur as intended.

4.2 Package and class Diagram:



-Figure 6 Package and Class Diagram-

5. Data Model

5.1.1.1 Entities & Relationships

My database follows a relational model and consists of the following main entities:

- **Person** (Represents the users of the application.)
- **Movie** (Stores information about movies)
- **Actor** (Links actors to movies)
- **Director** (Links directors to the movies they directed)
- **Review** (Links reviews to users and the respective movie they review)
- **ForgotPassword** (Links person with an generated otp)

5.1.1.2 Entity Descriptions

1. Person Table

- id (Primary Key, UUID)
- name (VARCHAR, Not Null)
- email (VARCHAR, Unique, Not Null)

2. Movie Table

- id (Primary Key, UUID)
- title (VARCHAR, Not Null)
- release_year (INTEGER, Not Null)
- genre (VARCHAR)
- director_id (Foreign Key -> Director)

3. Actor Table

- id (Primary Key, UUID)
- movie_id (Foreign Key → Movie)
- name (VARCHAR, Not Null)
- email (VARCHAR, Unique, Not Null)

4. Director Table (One-to-Many Relationship with Movie Table)

- id (Primary Key, UUID)
- name (VARCHAR, Not Null)
- email (VARCHAR, Unique, Not Null)

5. Movie_Actor Table (Many-to-Many Relationship)

- actor_id
- movie_id

6. Review Table (One- to-Many Relationship with Movie Table and with Person Table)

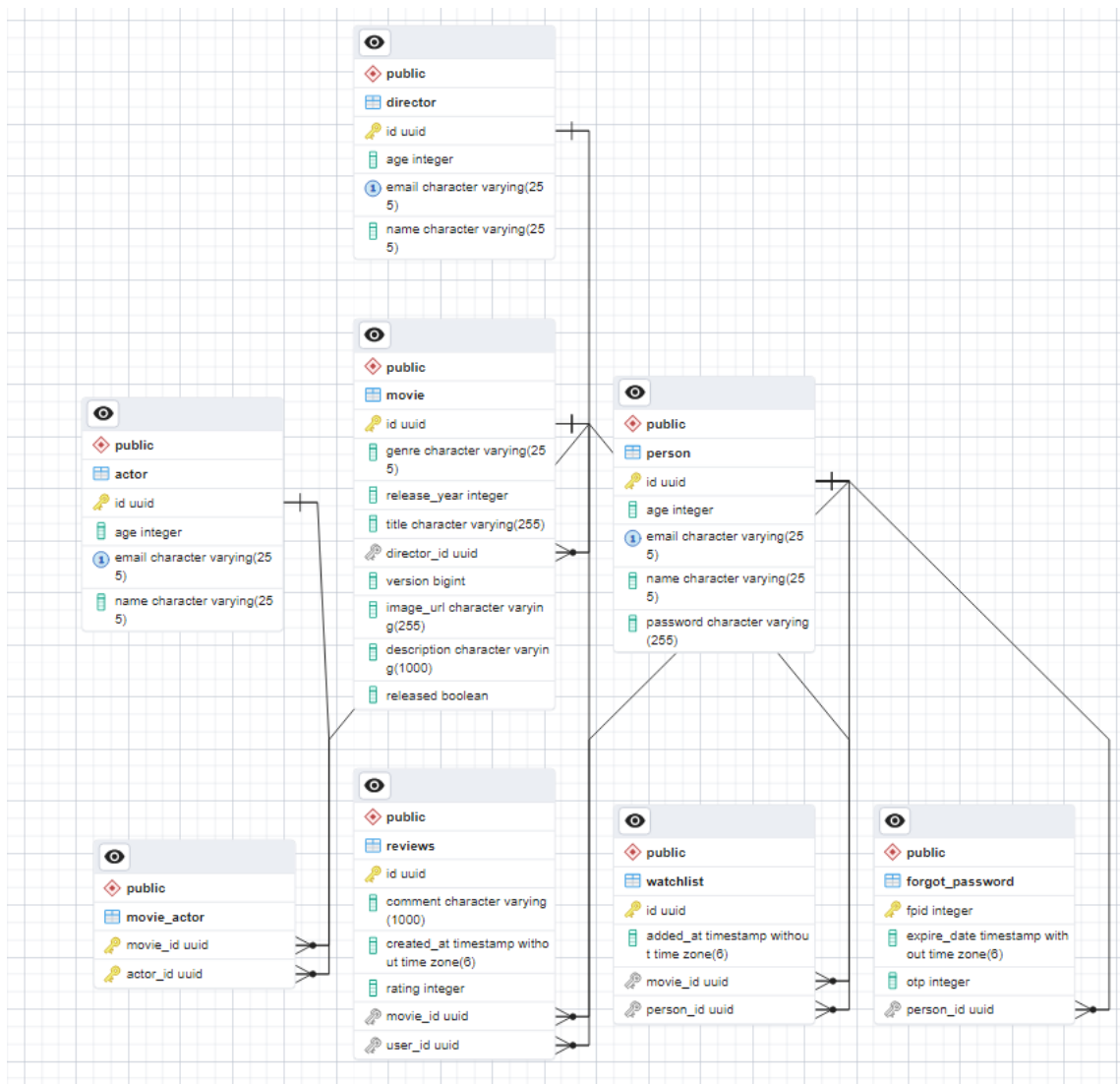
- id (Primary Key, UUID)
- comment (VARCHAR, Not Null)
- created_at (timestamp)
- rating (INTEGER, Not Null)
- movie_id
- user_id

7. ForgotPassword Table(One-To-One Relationship with Person Table)

- Fpid(Primary Key, Integer)
- Expire_date(timestamp without timezone)
- Otp(Integer)
- Person_id(Uuid)

8. Watchlist Table(Many-to-One Relationship with Person Table and with Movie Table)

- Id(Primary Key, uuid)
- Added_at(timestamp without timezone)
- Movie_id(uuid)
- Person_id(uuid)



-Figure 7 Diagram ERD-

6. Changes for the project

6.1.1 UI Design Overview

The user interface is designed to be simple, clean, and easy to use. It includes:

- **Consistent Style:** Uses a clear layout with matching colors, rounded corners, and modern fonts.
- **Custom Backgrounds:** Each page has styled backgrounds and areas for media content.

6.1.2 User Account Page

The **User Account Page** is a central hub for managing user-specific settings. Features include:

- Editable user information (name, email, age)
- See user watchlist

6.1.3 Watchlist Feature

Users can add movies to their personal **Watchlist**, which offers:

- Easy tracking of upcoming and released titles
- Real-time updates on movie release status

This system leverages the **Observer Design Pattern** to handle dynamic updates. Each user acts as an observer subscribed to movie release subjects. When a movie in the watchlist is released, a notification email is triggered and sent to the user automatically using a decoupled, event-driven mechanism.

6.1.4 Design Pattern: Observer

To implement the notification system, the **Observer Design Pattern** was used.

Each Movie object maintains a list of observers (users). Upon status change (e.g., from "not released" to "released"), it notifies all subscribed observers. This results in modular, maintainable code that cleanly separates core logic from user interaction handling.