<MovieApp>
Analysis and Design Document : Assignment 3

Student: <Moga Diana>
Group: 30435

Table of Contents

Contents

1. Requirements Analysis

1.1 Assignment Specification

The objective of this project is to develop an application that allows the administrator to manage and track information about movies, directors, and actors involved in film production, and allows users to register, log in, leave reviews, and view data. The application stores details about directors, actors, movies, and user reviews. It handles relationships such as many-to-many between actors and movies, one-to-many between directors and movies, and one-to-many between movies and reviews and users and reviews.

1.2 Functional Requirements

Admin Capabilities:

1. Log in using admin credentials.
2. Create new movie entries with details such as title, genre, release date, and director name.
3. Add new actors and directors to the database, including names and relevant details.
4. Assign multiple actors to a movie and 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.

User Capabilities:

4. Register a new user account.
5. Log in using valid credentials.
6. View lists of movies, actors, and directors.
7. Leave reviews on movies, including a numeric rating (e.g., 1–5 stars) and a text comment.
8. Edit or delete their own reviews.
9. Filter and sort the movie list.

1.3 Non-functional Requirements

1. **Validation**: All inputs must be validated, including user registration info, review content, and entity data (e.g., no empty fields, valid date formats, rating within allowed range).
2. **Authentication & Authorization**:
   - Users must be authenticated to leave, edit, or delete reviews.
   - Only authenticated admin can perform CRUD operations on movies, actors, and directors.
   - JWTs are created and stored

3. **Password Hashing**

   - Passwords must be hashed on the backend before being stored in the database. Plaintext passwords should never be stored.

4. **ORM Integration**: Use an Object-Relational Mapping (ORM) tool to manage entity relationships and interactions with the database.
5. **API Design**: Provide a RESTful API with clear role-based access control for CRUD operations and user actions.
6. **Database**: Use a relational database to store data about movies, actors, directors, users, and reviews.

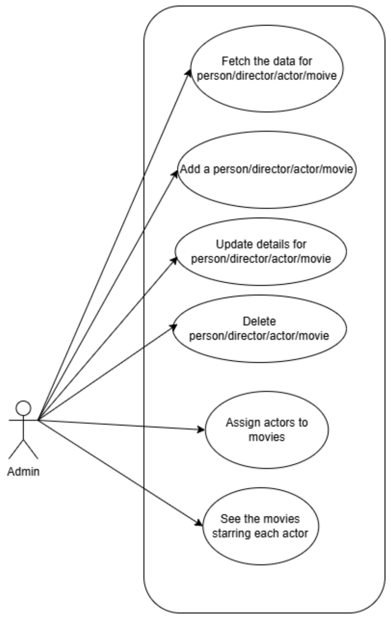
   Support:

   - Many-to-many: Actors ↔ Movies
   - One-to-many: Director → Movies
   - One-to-many: Movie → Reviews
   - One-to-many: Person → Reviews
7. **Separation of Concerns**: Follow a layered architecture with distinct responsibilities for models, controllers, services, and repositories.
8. **User Interface**:
   - Provide a clean, intuitive UI for users to browse and review movies.
   - Admins should have access to a dashboard for managing entities.

8. **Quality Assurance and Testing**:

- **Integration Testing**: Utilize the Spring Boot Test framework to perform integration tests of the controller layers in an application context. MockMvc will simulate HTTP requests and validate controller behavior, while TestPropertySource will isolate test data and environment. Fixtures (JSON files) will be used to preload test data for repeatable test scenarios.
- **Unit Testing**: Mockito will be employed to mock repository and service dependencies, ensuring isolated testing of service layer logic. JUnit 5 will be used as the main testing framework for assertions and Mockito verification to confirm interactions with dependencies.

1. Use-Case Model


Fetch the data for person/director/actor/moive
Add a person/director/actor/movie
Update details for person/director/actor/movie
Delete person/director/actor/movie
Assign actors to movies
See the movies starring each actor
Admin

-Figure 1 Use-Case Diagram: Admin-

1.1 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
Primaray actor: The admin of the application
Main succes 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 .

1.2 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.

1.3 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.

1.4 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.

1.5 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:

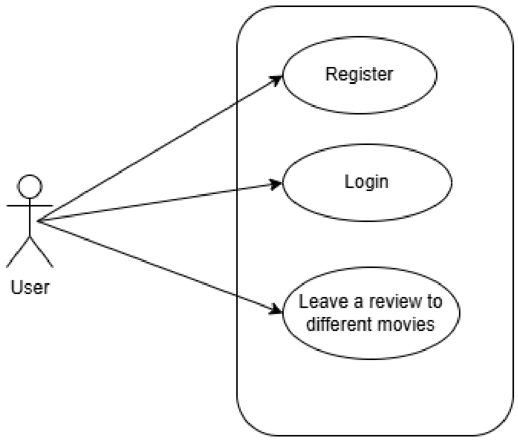
- Alternate scenario of success: The assignment succeeds, but some actors were already linked to the movie. The system prevents duplicate assignments and notifies the admin.

- 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.

1.6 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-

1.7 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.

1.8 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.

1.9 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.

1.10 Description of the Filter use-case

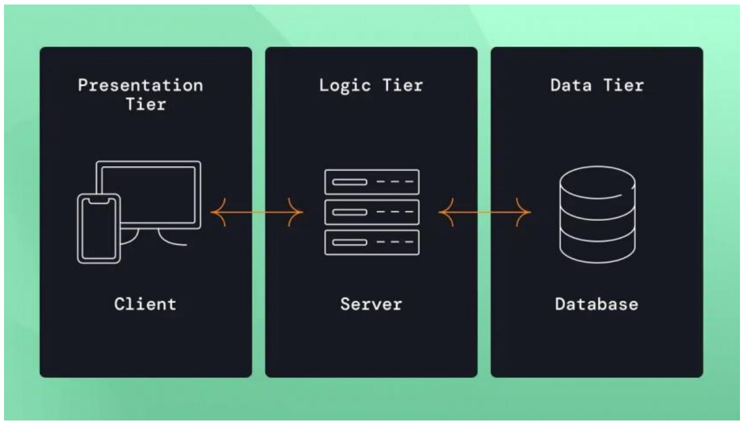
Use case goal: Allow the user to filter movies by various criteria (e.g., genre, release year) and order them (e.g., by title, rating).

Primary Actor:The user of the application.

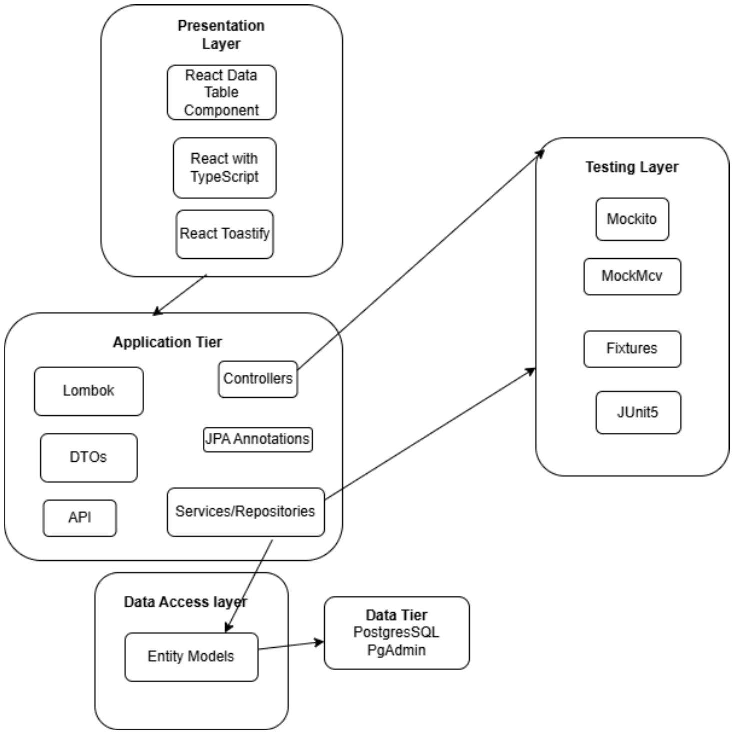
Main Success Scenario: the user accesses the movie listing page. The user selects filter criteria (e.g., genre = "Action", year > 2020). The user selects the ordering method (e.g., by title A-Z or rating high to low). The system applies the filters and ordering to the movie table. The updated, filtered,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

2. System Architectural Design

2.1 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:

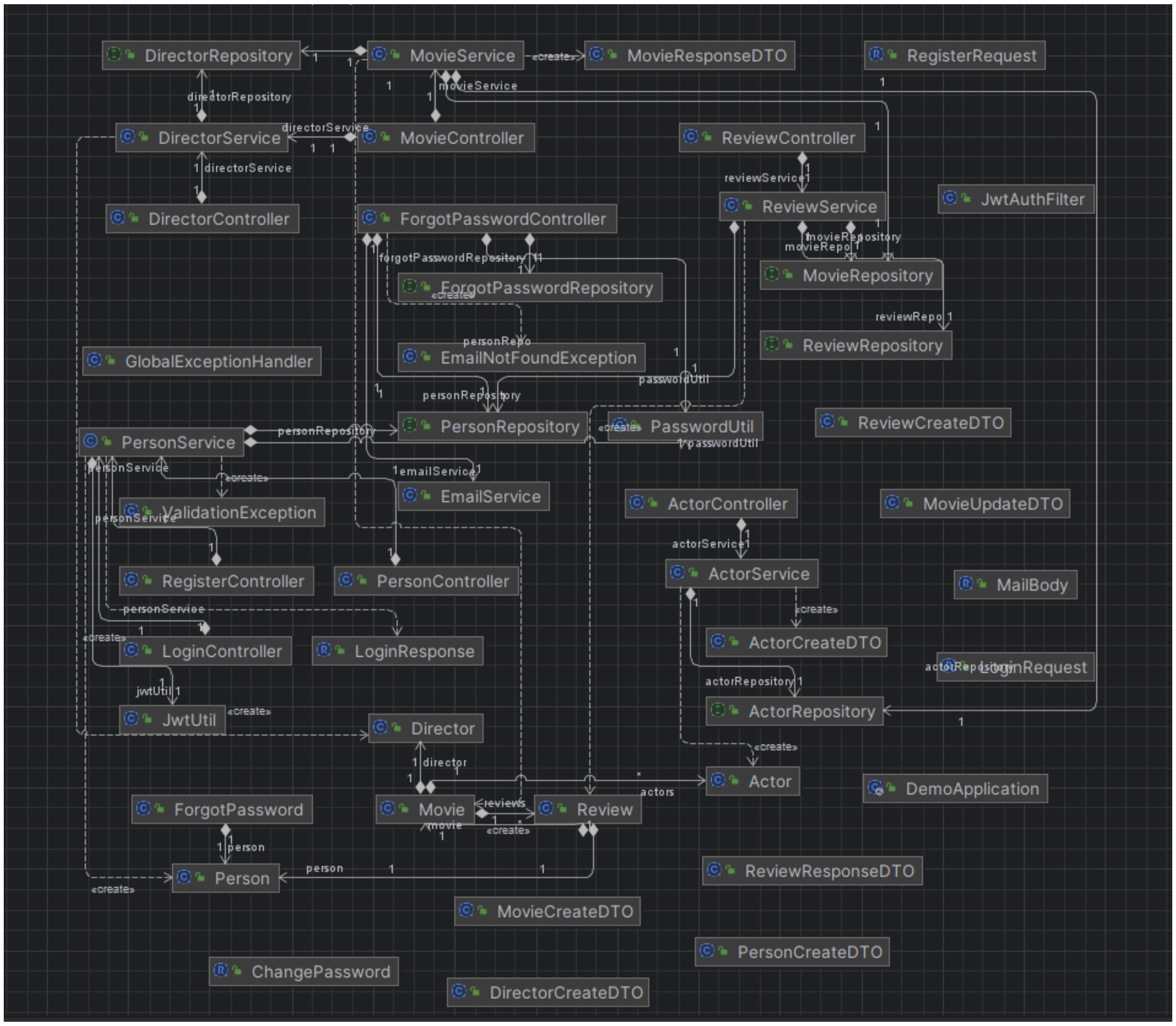
- **Spring Boot Test** framework is used to test controller layers in an application context.
- **MockMvc** is employed to simulate HTTP requests and validate controller behavior.
- **TestPropertySource** provides a separate configuration (application-test.properties) to isolate test data and environment.
- **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.

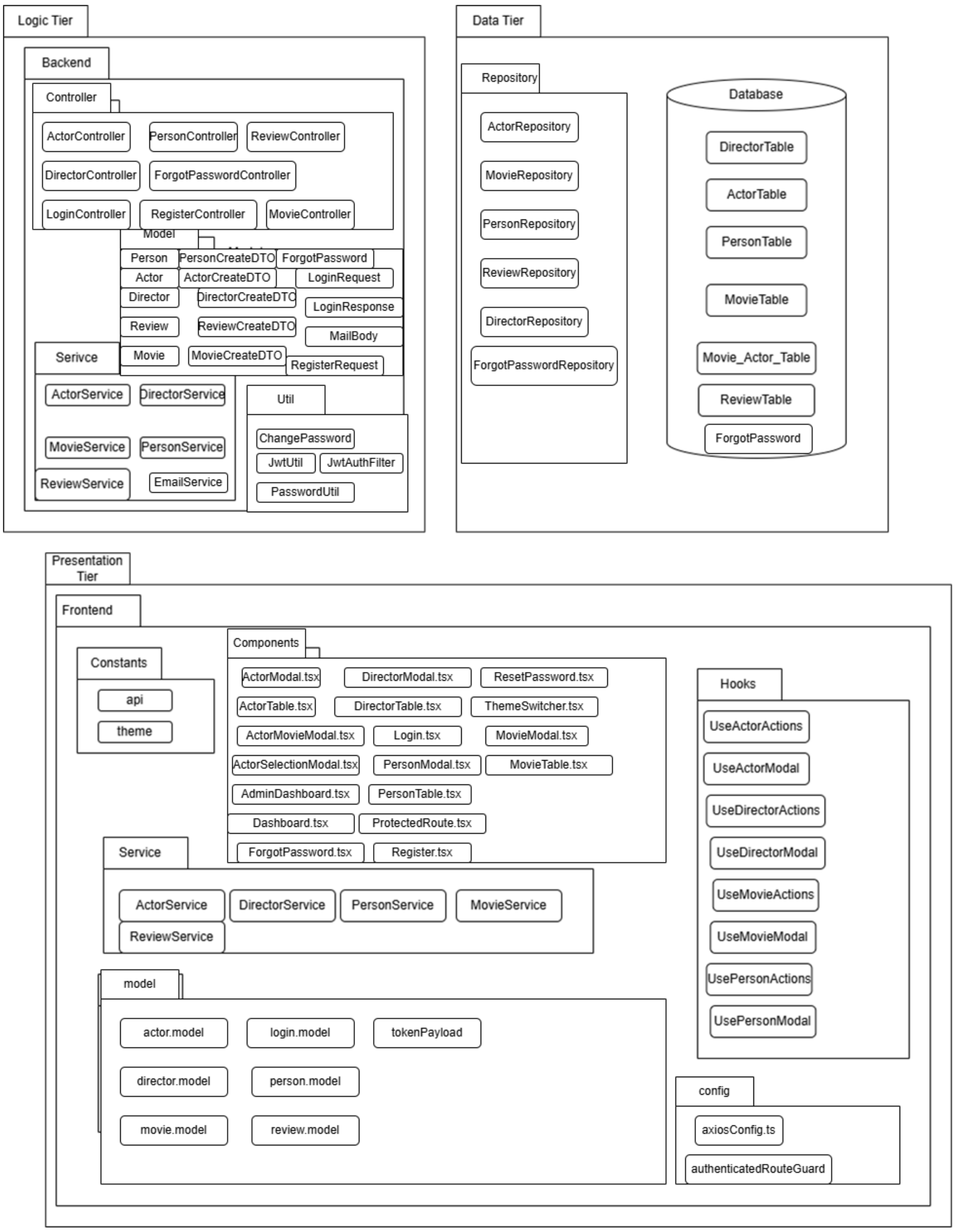
3. Class Design

4.1 Class diagram backend:



-Figure 5 Backend classes-

4.2 Package and class Diagram:



-Figure 6 Package and Class Diagram-

4. Data Model

4.1.1.1 Entities & Relationships

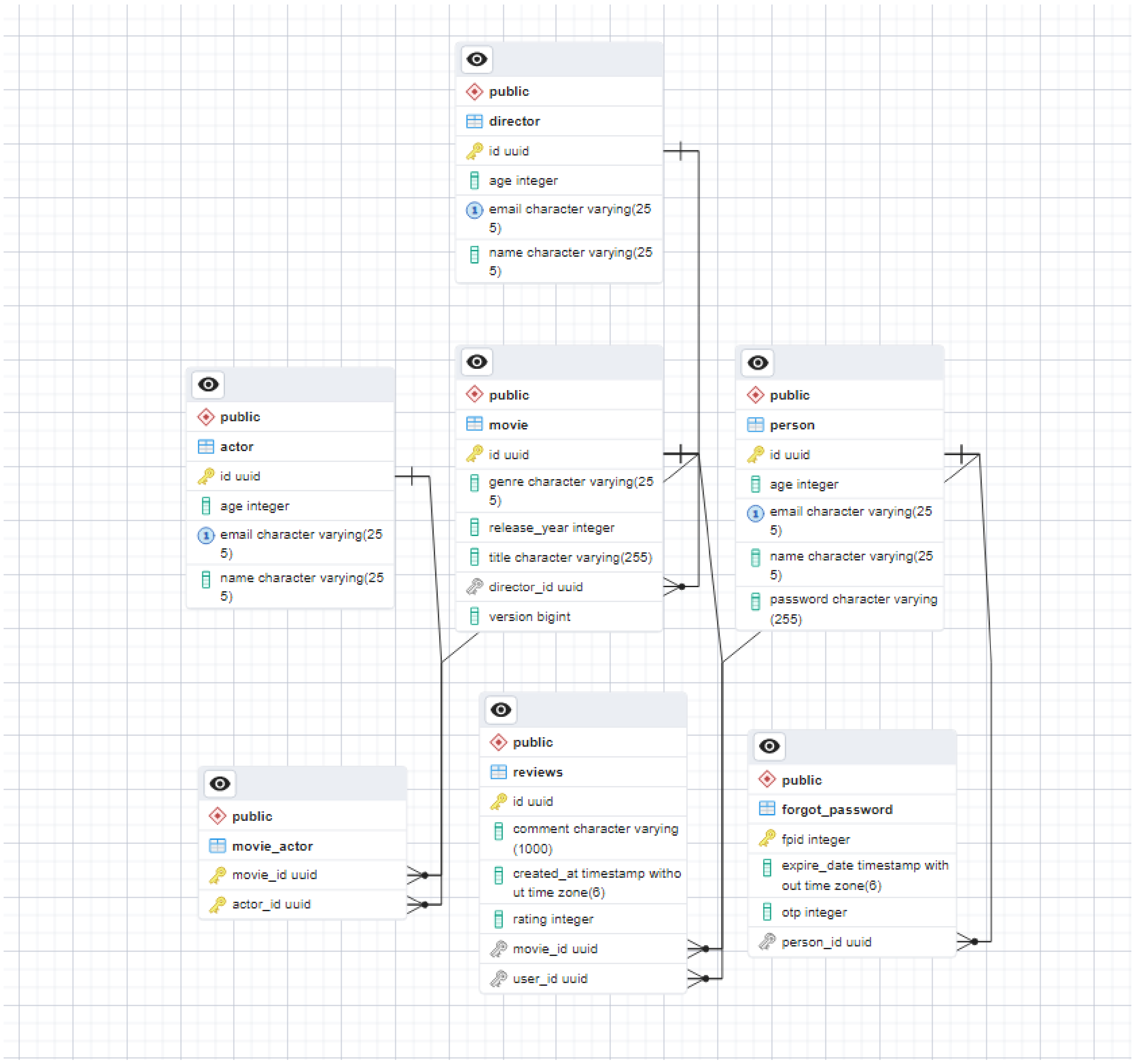
My database follows a relational model and consists of the following main entities:

- **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)

4.1.1.2 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)


public
director
id uuid
age integer
email character varying(255)
name character varying(255)
public
actor
id uuid
age integer
email character varying(255)
name character varying(255)
public
movie
id uuid
genre character varying(255)
release_year integer
title character varying(255)
director_id uuid
version bigint
public
person
id uuid
age integer
email character varying(255)
name character varying(255)
password character varying (255)
public
movie_actor
movie_id uuid
actor_id uuid
public
reviews
id uuid
comment character varying (1000)
created_at timestamp without time zone(6)
rating integer
movie_id uuid
user_id uuid
public
forgot_password
fpid integer
expire_date timestamp without time zone(6)
otp integer
person_id uuid

-Figure 7 Diagram ERD-