**Backend Documentation**

**Overview**

The backend of this project is developed using Spring Boot, a popular Java framework for building web applications. It provides a robust and scalable architecture for handling HTTP requests, data persistence, and business logic implementation.

Technologies Used: Java, Spring Boot, Spring Data JPA, MySQL, RESTful API

**Project Structure**

The backend code is organized into the following packages and classes:

**1.entity Package**:

Actor.java: Defines the entity class for actors, annotated with JPA annotations to map it to the database table. It represents actors with attributes such as ID, full name, and relationships with movies.

Movie.java: Defines the entity class for movies, annotated with JPA annotations to map it to the database table. It represents movies with attributes such as ID, title, release year, genre, and relationships with actors.

**2.repository Package:**

ActorRepository.java: Interface extending JpaRepository for actor entities, providing methods to interact with the database such as retrieving all actors, finding actors by ID, and more.

MovieRepository.java: Interface extending JpaRepository for movie entities, providing methods to interact with the database such as retrieving all movies, finding movies by ID, and more.

**3.service Package:**

ActorService.java: Service class containing business logic for actors, such as retrieving all actors, retrieving an actor by ID, creating a new actor, updating an existing actor, and deleting an actor.

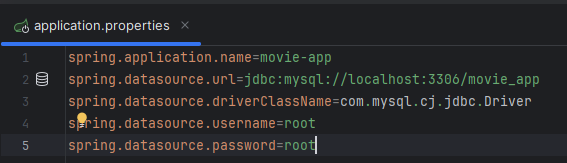
MovieService.java: Service class containing business logic for movies, such as retrieving all movies, retrieving a movie by ID, creating a new movie, updating an existing movie, and deleting a movie.

**4.controller Package:**

ActorController.java: REST controller class defining endpoints for actor-related operations, including retrieving all actors, retrieving an actor by ID, creating a new actor, updating an existing actor, and deleting an actor.

MovieController.java: REST controller class defining endpoints for movie-related operations, including retrieving all movies, retrieving a movie by ID, creating a new movie, updating an existing movie, and deleting a movie.

**6.application.properties:** Configuration file containing database connection settings, such as database URL, username, and password.



**application.properties**

This file contains configurations for the Spring Boot application.

spring.application.name: Specifies the name of the Spring Boot application, which is set to "movie-app".

spring.datasource.url: Defines the JDBC URL for connecting to the MySQL database. In this case, it connects to a database named "movie\_app" running on localhost with the default MySQL port (3306).

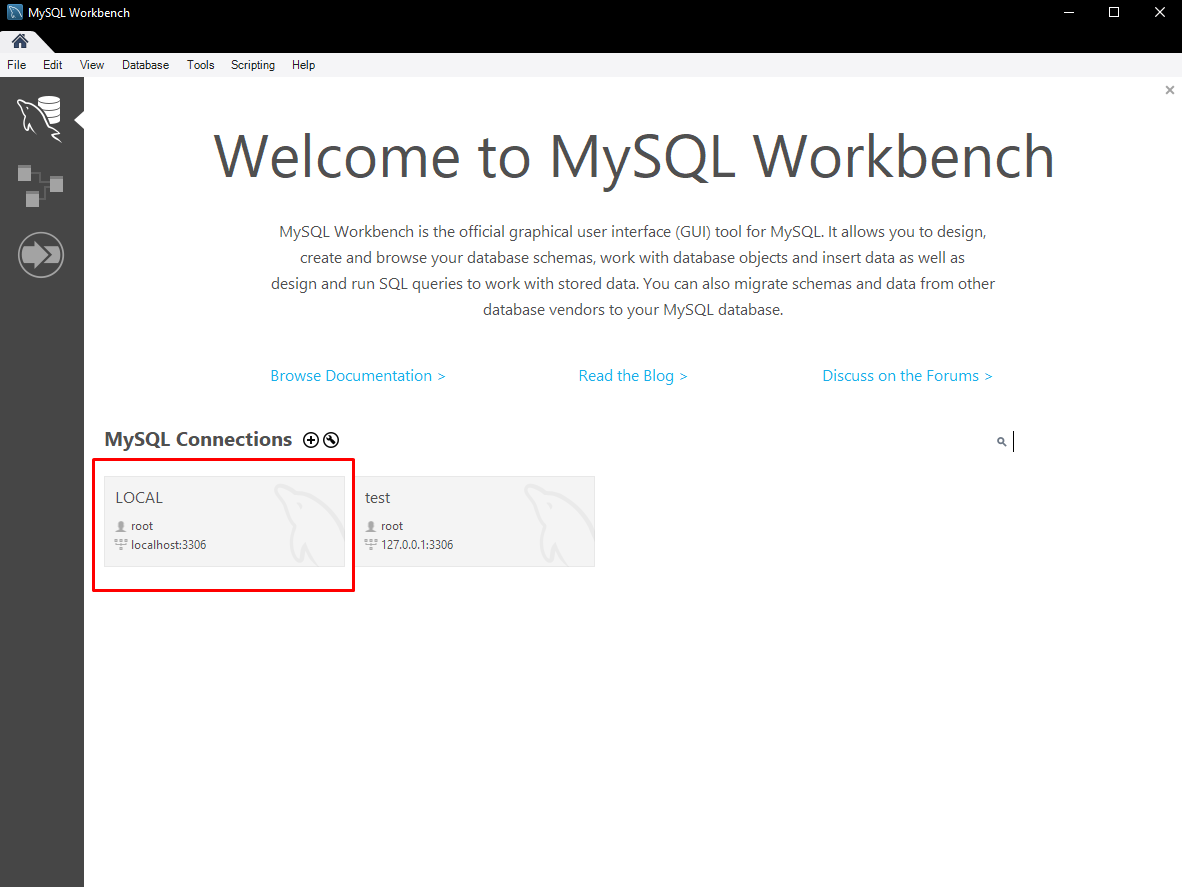
spring.datasource.driverClassName: Specifies the JDBC driver class to be used for connecting to the MySQL database. Here, it's set to "com.mysql.cj.jdbc.Driver".

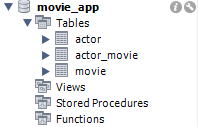
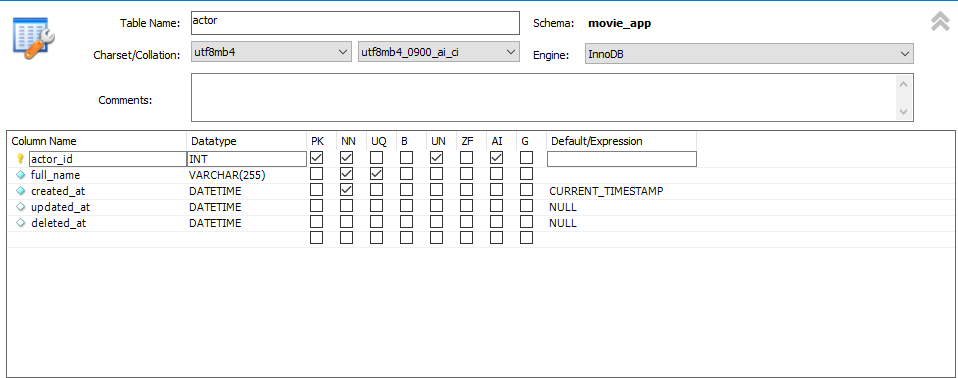
spring.datasource.username: Defines the username used to authenticate with the MySQL database server. In this example, it's set to "root".

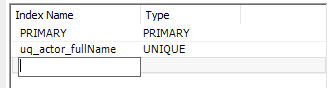
spring.datasource.password: Specifies the password used to authenticate with the MySQL database server. Here, it's set to "root".

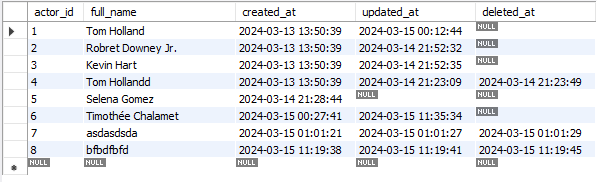
These configurations ensure that the Spring Boot application can establish a connection to the MySQL database server.

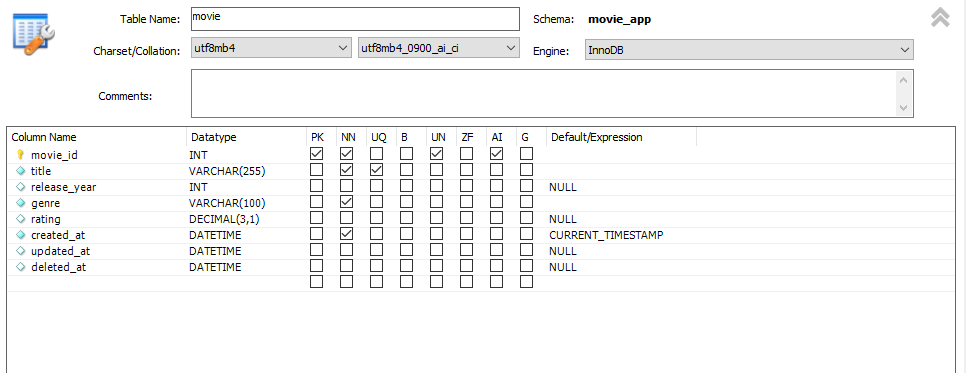
**MySQL**

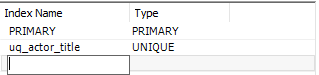


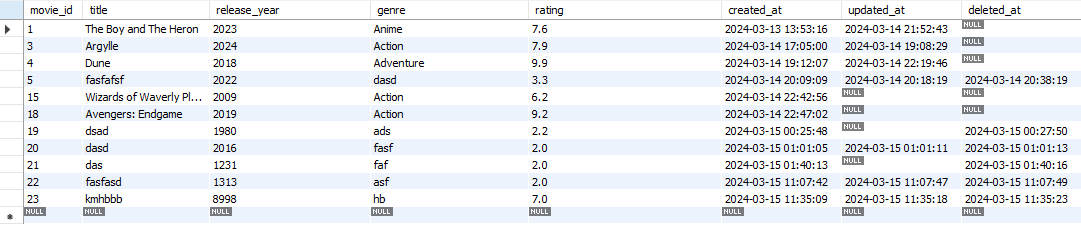
Then I made a LOCAL database in the My SQL Workbench  
 and made a schema called “movie\_app” with tables needed for the project which are actor and movie plus a many-to-many relationship with table “actor\_movie”  
  


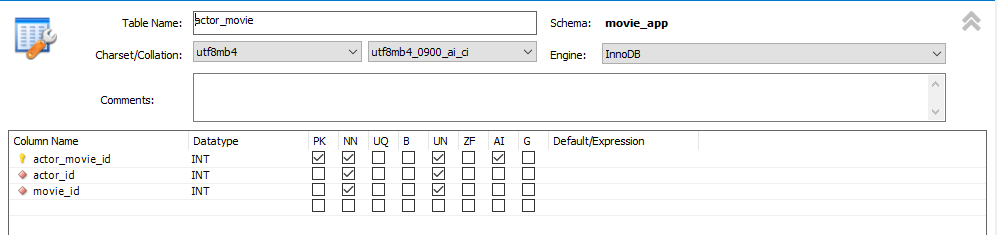


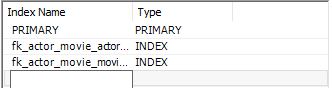
My actor database which looks like this:  


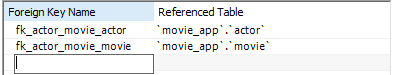


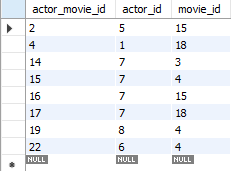


My movie database which looks like this:  






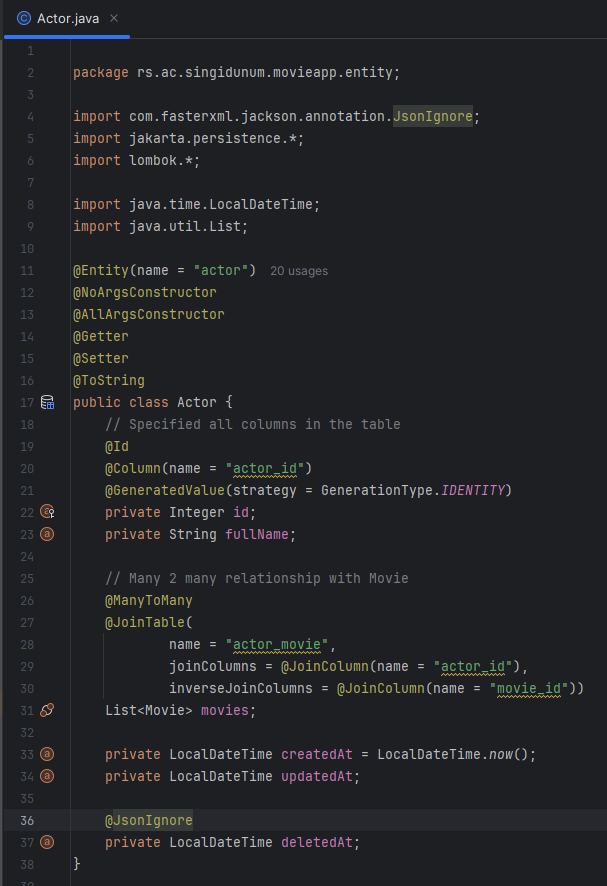


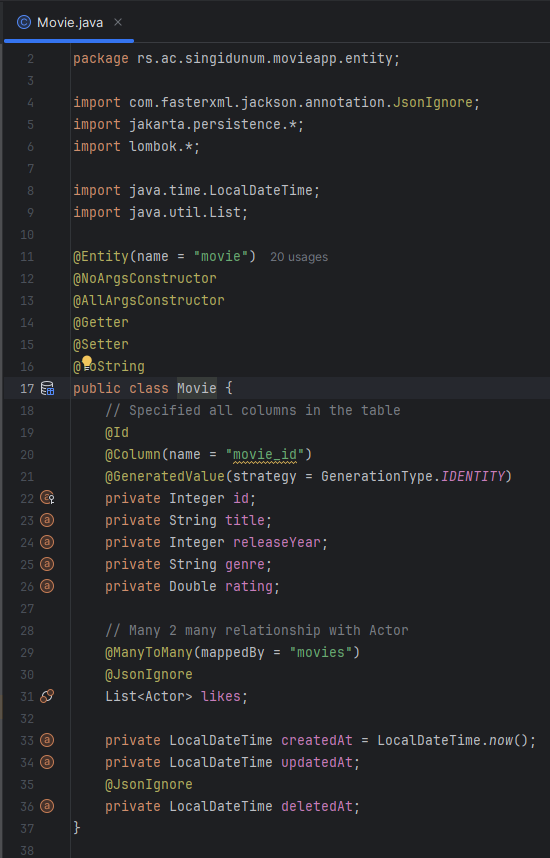
My actor\_movie database which looks like this:  
Which are only ids in it for the actor and movie the connect

**BACKEND**

Now in the IntelliJ program we have to make a controller, entity, repository and service for both Actor and Movie

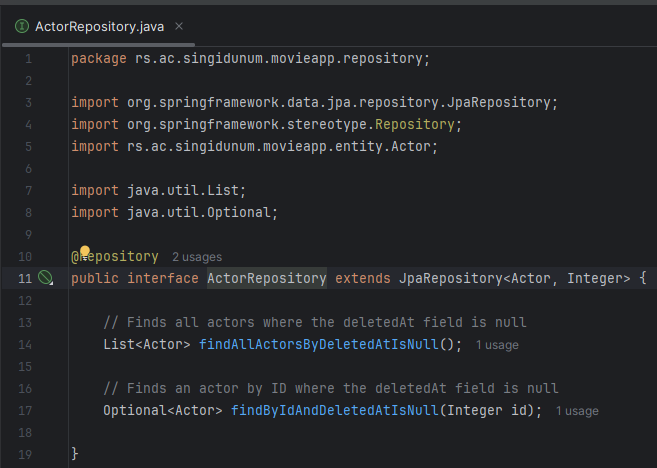
**Entity**

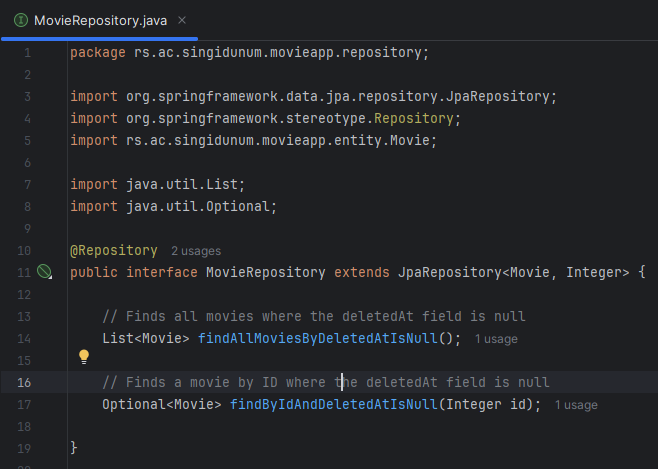




These classes represent the entity for actors and movies in the movie application. It is annotated with JPA annotations to map it to a corresponding database table.

**Repository**

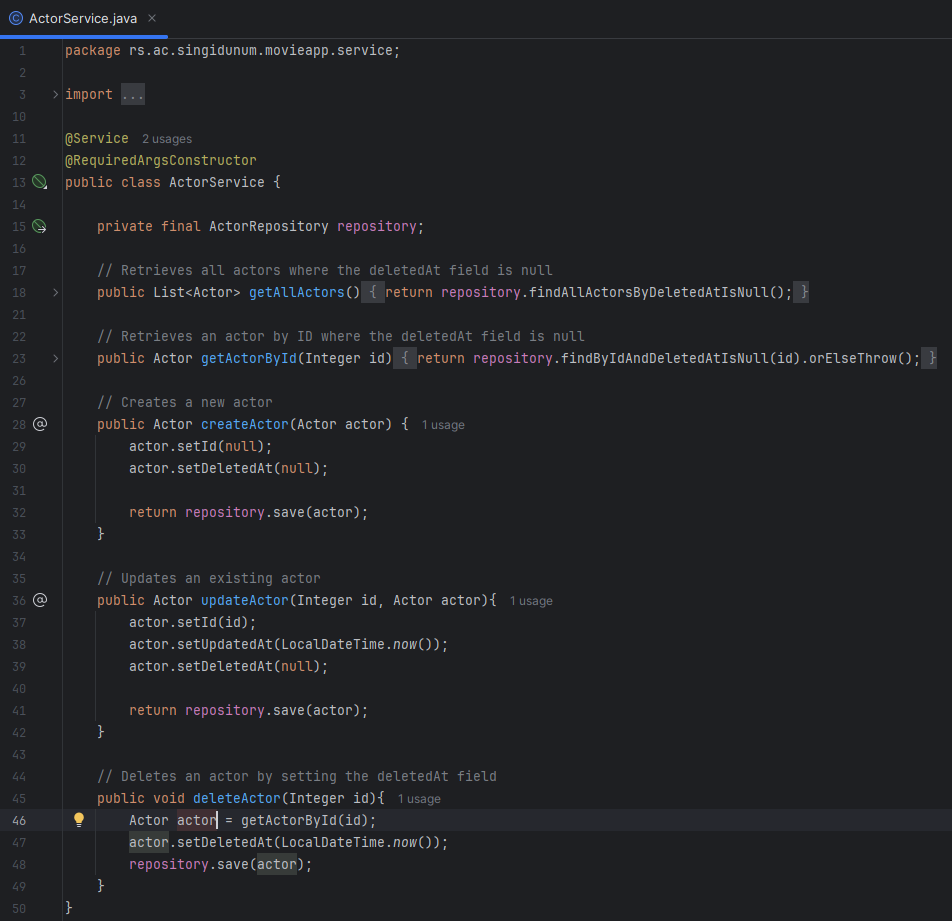


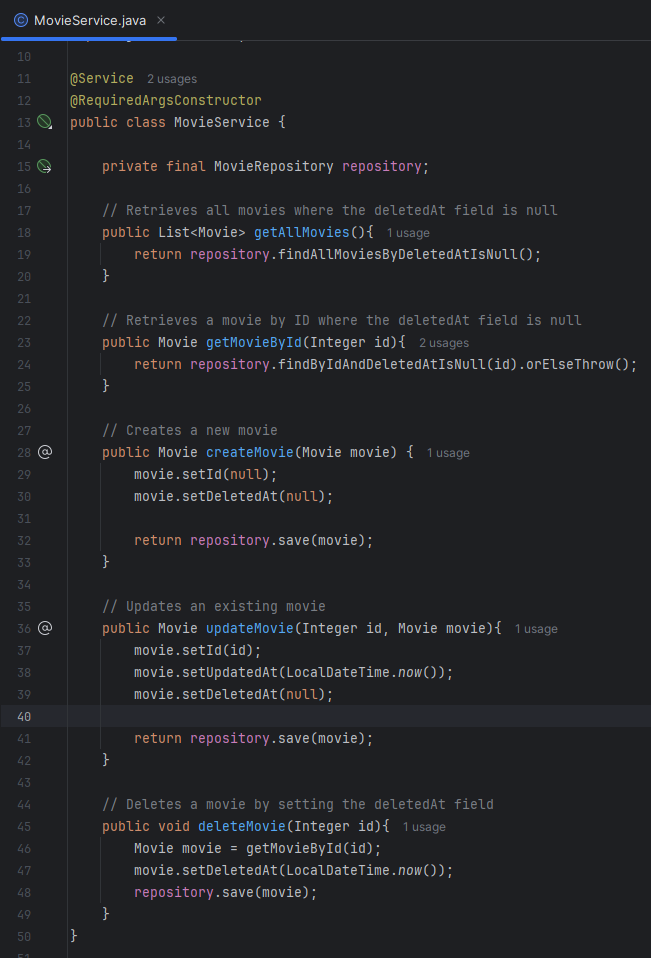


This interface extends **JpaRepository** and provides methods for interacting with the **Actor** entity in the database. And the same for **Movie** entity.

These repository uses Spring Data JPA's powerful features to provide CRUD (Create, Read, Update, Delete) operations for the corresponding entities.

**Service**

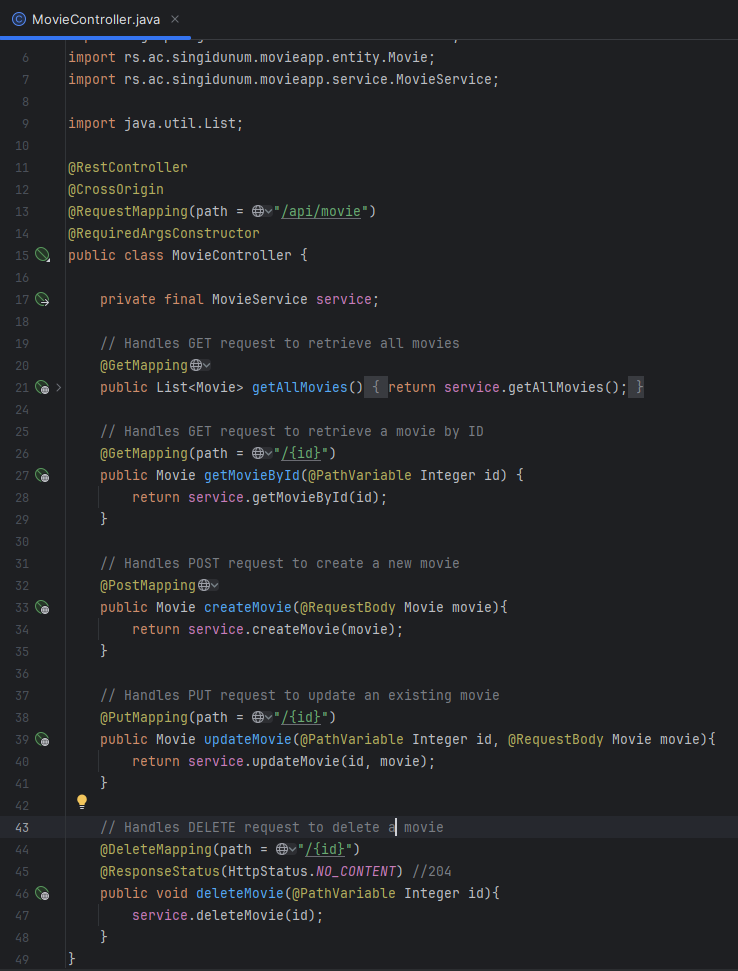




This service class provides methods to handle business logic related to actors and movies in the movie application. These service classes encapsulate the business logic for actors and movies, providing a clean separation between the controller layer and the data access layer. They facilitate CRUD operations on actors and movies while handling the underlying repository interactions.

**Controller**

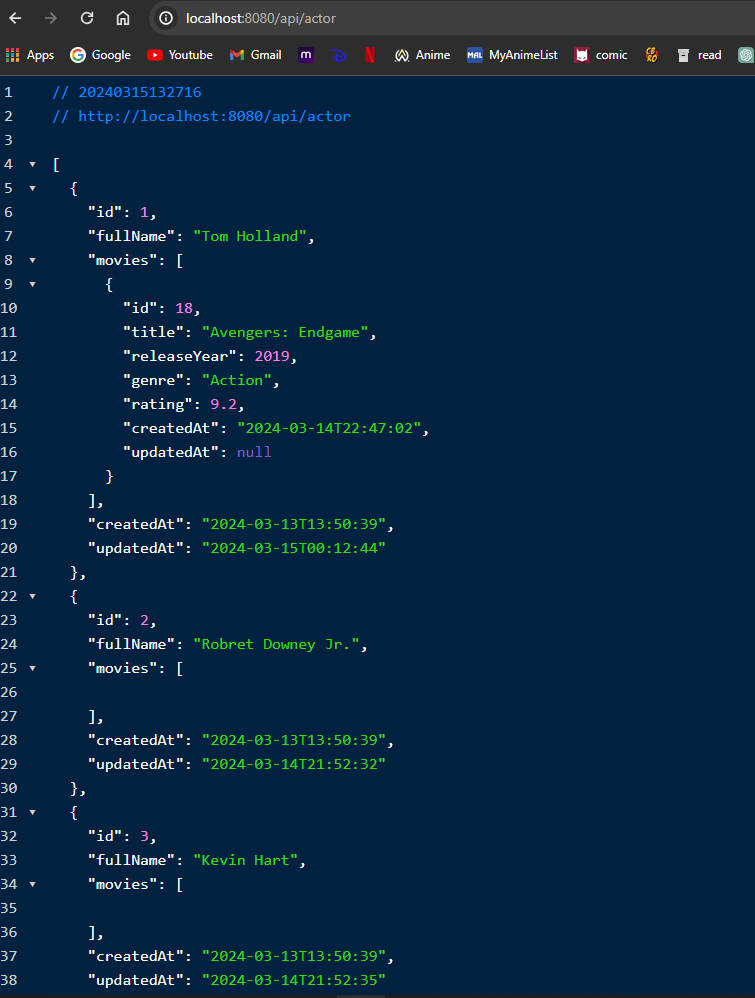


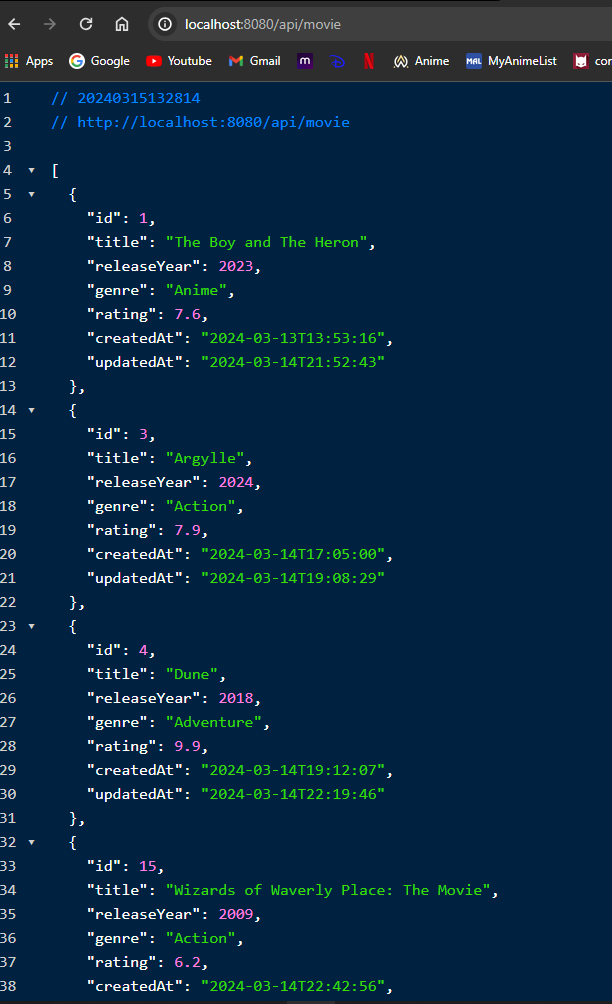


This controller class handles HTTP requests related to actors and movies in the movie application. It utilizes the **ActorService** and **MovieService** to process these requests.

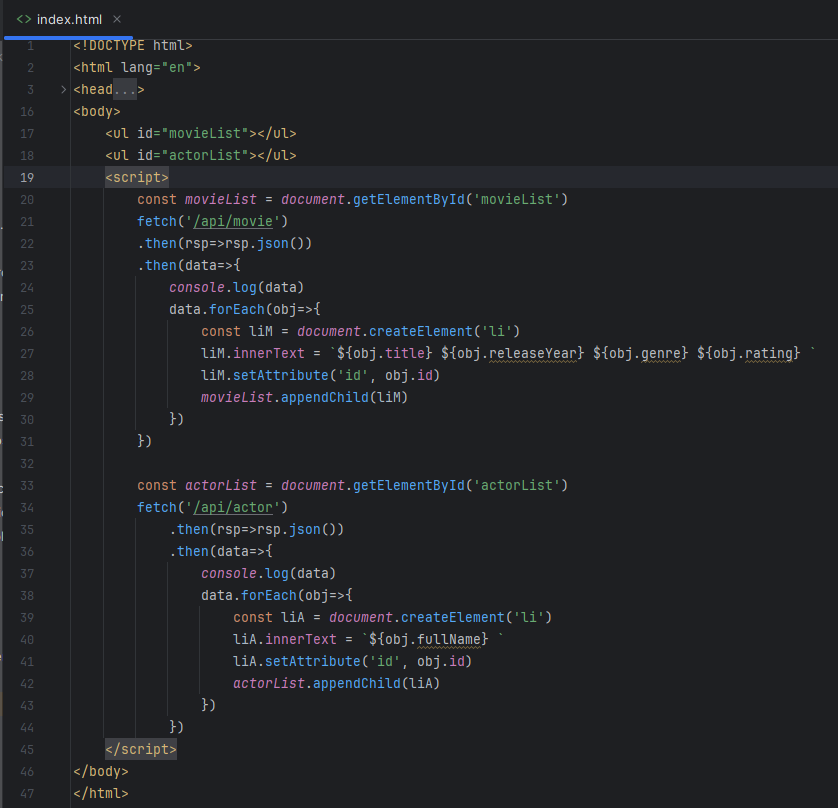
These controller classes define the RESTful API endpoints for managing actors and movies in the application, facilitating CRUD operations.

**Conclusion**

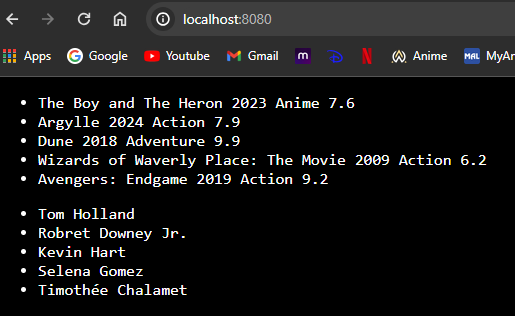
When I turn on the application with a json formatter, and I go first to <http://localhost:8080/api/actor>:  
  


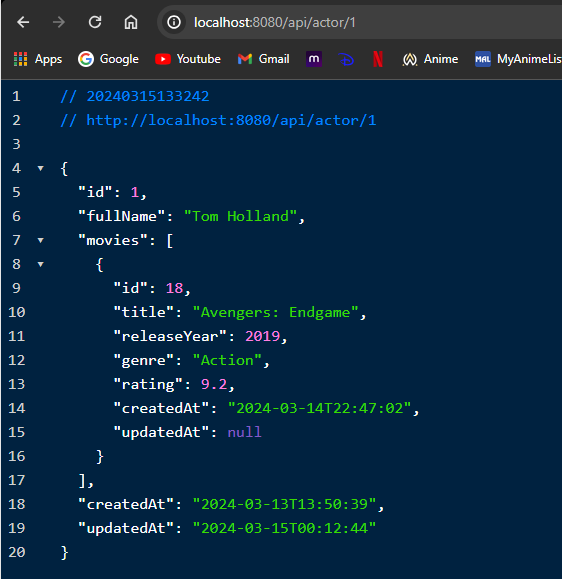
When I go to <http://localhost:8080/api/movie>:  


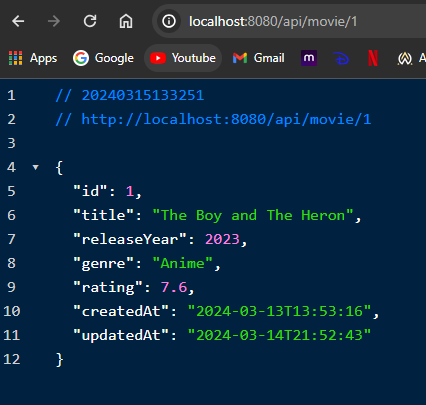
This HTML document is creating a simple webpage that fetches movie and actor data from backend and displays them in unordered lists.



Now I did the .html file so it would show something on the page <http://localhost:8080>:



Also it would work for both when I go to some id like <http://localhost:8080/api/actor/1> or <http://localhost:8080/api/movie/1> :  
  


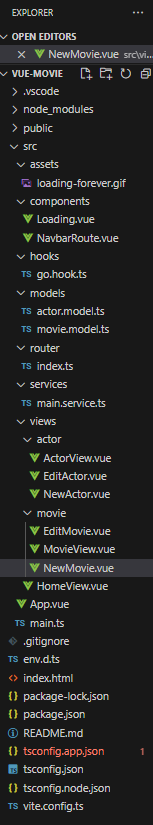


**Frontend**

Overview

The frontend of this project is developed using Vue.js, a progressive JavaScript framework for building user interfaces. It consists of several Vue components that interact with the backend API to display and manage movie and actor data.

Technologies Used: Vue.js, Vue Router, Axios (for HTTP requests), TypeScript (for type-safe development)



**Project Structure**

The frontend code is organized into the following components and services:

App.vue: The root component of the Vue.js application, responsible for rendering other components based on the current route.

Home.vue: The home page component.

ActorView.vue: A component to view details of all actors.

EditActor.vue: A component to edit details of an existing actor.

NewActor.vue: A component to create a new actor.

MovieView.vue: A component to view details of all movies.

EditMovie.vue: A component to edit details of an existing movie.

NewMovie.vue: A component to create a new movie.

main.service.ts: A TypeScript service file containing methods to interact with the backend API endpoints for CRUD operations on movies and actors.

actor.model.ts: A TypeScript file defining the model interface for actor objects.

movie.model.ts: A TypeScript file defining the model interface for movie objects.

**Functionality**

Vue Components

Home.vue: Displays a homepage

ActorView.vue: Displays details of all actors, including full name and related movies.

EditActor.vue: Allows editing details of an existing actor.

NewActor.vue: Provides a form to create a new actor.

MovieView.vue: Displays details of all movies, including title, release year, genre, and rating.

EditMovie.vue: Allows editing details of an existing movie.

NewMovie.vue: Provides a form to create a new.

**Services**

main.service.ts: Contains methods to make HTTP requests to the backend API endpoints for CRUD operations on movies and actors. It utilizes Axios for making HTTP requests and TypeScript for type-safe development.

