Practice Problem: Movie Library Management System

Goal

The goal of this project is to build a RESTful API for managing a Movie Library. The system will allow users to perform CRUD operations on movies, directors, and genres. This practice exercise aims to solidify your understanding of building backend applications with Spring Boot, including entity relationships, data persistence, and API development.

Entities

1. **Movie**
    - Attributes: `id`, `title`, `releaseYear`, `plot`, `director`, `genre`
    - Relationships:
        - Many-to-One with `Director`
        - Many-to-One with `Genre`
2. **Director**
    - Attributes: `id`, `name`, `birthDate`
    - Relationships:
        - One-to-Many with `Movie`
3. **Genre**
    - Attributes: `id`, `name`, `description`
    - Relationships:
        - One-to-Many with `Movie`

Functionalities

- **Movie Operations:**
    - Create, Read, Update, Delete (CRUD) operations

- **Director Operations:**
  - CRUD operations
- **Genre Operations:**
  - CRUD operations

Endpoints

- `POST /movies` - Create a new movie.
- `GET /movies/{id}` - Retrieve details of a specific movie.
- `PUT /movies/{id}` - Update details of a specific movie.
- `DELETE /movies/{id}` - Delete a specific movie.
- `POST /directors` - Create a new director.
- `GET /directors/{id}` - Retrieve details of a specific director.
- `PUT /directors/{id}` - Update details of a specific director.
- `DELETE /directors/{id}` - Delete a specific director.
- `POST /genres` - Create a new genre.
- `GET /genres/{id}` - Retrieve details of a specific genre.
- `PUT /genres/{id}` - Update details of a specific genre.
- `DELETE /genres/{id}` - Delete a specific genre.

Validation and Error Handling

- Implement validation for input data (e.g., required fields, valid formats).
- Handle exceptions and provide meaningful error responses (e.g., 404 for not found, 400 for bad requests).

Additional Features (Optional)

- Pagination and sorting for retrieving lists of movies, directors, and genres.
- Search functionality to find movies by title, directors by name, or genres by name.

Implementation Steps

1. Set up your Spring Boot project with dependencies like Spring Web, Spring Data JPA, and an embedded database (e.g., H2 Database).
2. Define entity classes (`Movie`, `Director`, `Genre`) with appropriate annotations (`@Entity`, `@OneToMany`, `@ManyToOne`) for relationships.
3. Implement repositories (`MovieRepository`, `DirectorRepository`, `GenreRepository`) extending `JpaRepository` to handle database operations.
4. Create service classes (`MovieService`, `DirectorService`, `GenreService`) to encapsulate business logic and interact with repositories.
5. Develop REST controllers (`MovieController`, `DirectorController`, `GenreController`) to define endpoints and handle HTTP requests.
6. Write unit and integration tests to verify the functionality of your API using tools like JUnit and Mockito.
7. Test your API using tools like Postman or IntelliJ IDEA's built-in HTTP client to ensure endpoints work as expected.

Conclusion

By completing this practice project, you will gain practical experience in building a comprehensive backend application with Spring Boot, handling complex entity relationships, and developing robust RESTful APIs. This exercise will help reinforce your understanding of software architecture and best practices in API development.