Training Project Laboratory documentation

Movie website

By Sooronbaev Adilet & Doskozhoeva Eliza

1. **Content:**

2. INTRODUCTION

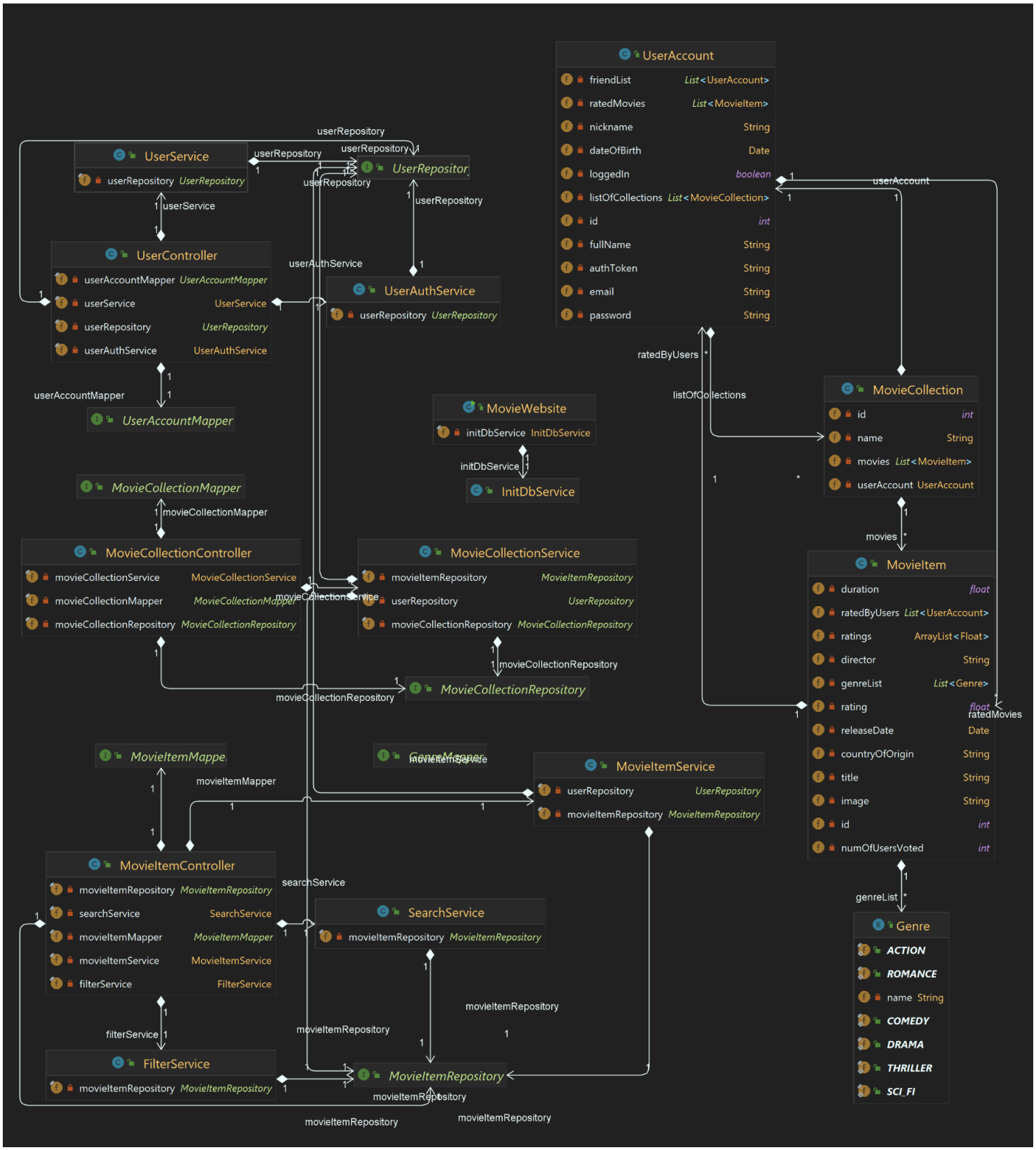
Our movie website project aims to create a comprehensive platform for movie enthusiasts to explore, organize, and engage with their favorite films. The primary goals are to provide users with a seamless and enjoyable experience in discovering movies, creating and managing collections, and connecting with a community of fellow cinephiles.

3. PROJECT OVERVIEW

Our movie website project is a seamless blend of intuitive design and robust functionality. Developed using React for the frontend and Java with Spring Boot for the backend, the platform allows users to register accounts, create personalized collections, browse through a vast catalog of movies, rate movies, and view their details. PostgreSQL serves as the backbone for storing user information and movie details, while RESTful APIs ensure smooth communication between the frontend and backend. With a focus on a responsive user interface and modern technologies, our movie website aims to provide an engaging and personalized cinematic experience.

4. IMPLEMENTATION DETAILS

Class diagram

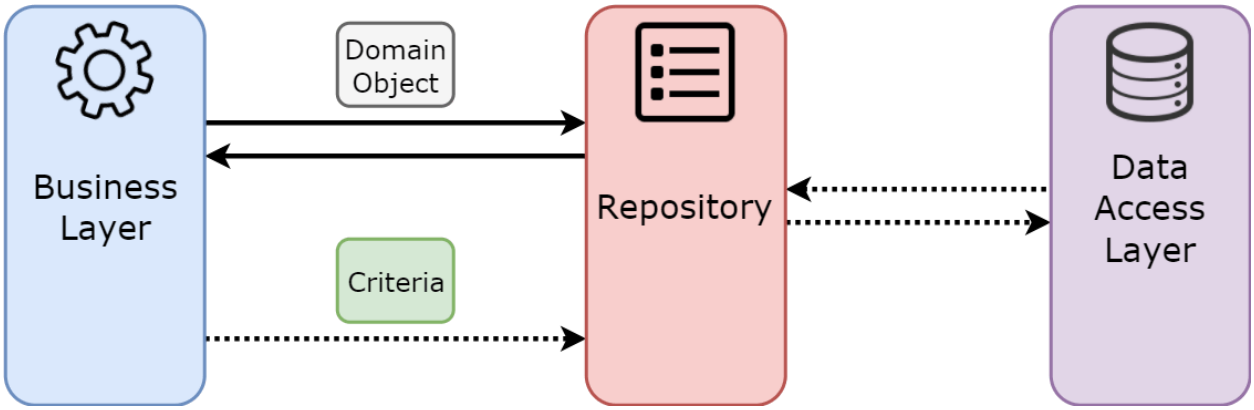


Repository pattern

The project has adopted the Repository Pattern to abstract the data access layer from the higher-level business logic.

This pattern facilitates a separation of concerns by providing interfaces and classes responsible for handling CRUD (Create, Read, Update, Delete) operations on specific entities.

The repositories leverage Spring Data JPA to interact with the underlying database, offering a structured approach to manage database operations efficiently.



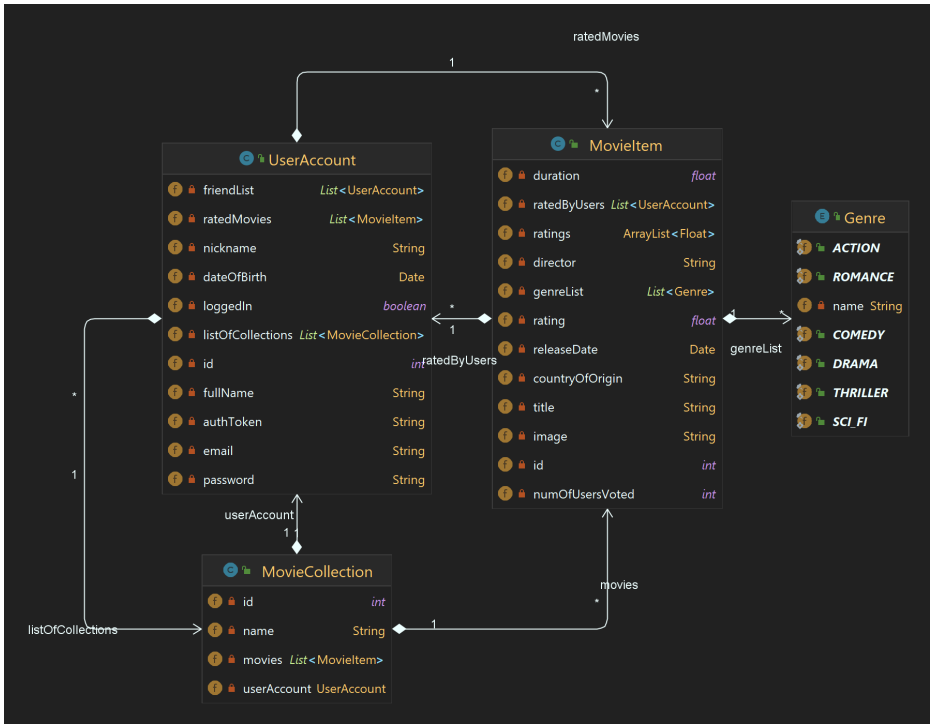
Model-View-Controller (MVC)

The application is structured based on the MVC architectural pattern, segregating responsibilities into three main components:

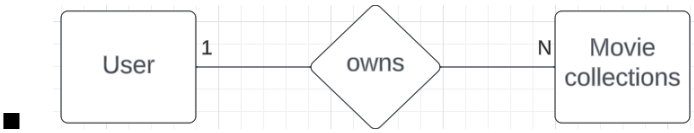
- **Model** (Entity and Repository): Defines the data structure and handles data access to the database.
- **View** (not explicitly mentioned in Spring MVC): Typically refers to the presentation layer in traditional MVC, though in Spring Boot, it largely concerns the HTTP response structure.
- **Controller:** Handles incoming HTTP requests, orchestrates interactions with the service layer, and forms the endpoint interface for the application.

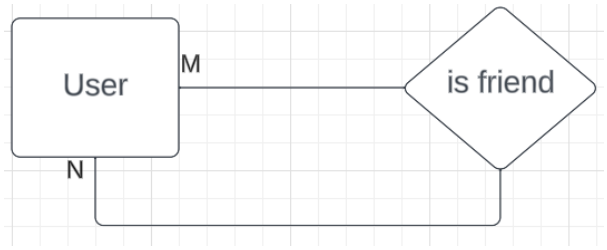
Components

- **Model classes:** Represent the structure and relationships of the application's data entities, mapping to corresponding tables in the database.



  - **UserAccount:** encapsulates essential user information within the application. It serves as a representation of a user profile and contains various fields. The most important field:
    - ***AuthToken:*** Serves as a unique token generated upon user login, granting authorized access to various functionalities and operations within the application.
    - ***ListOfCollections:*** represents a list of movie collections the user has. ***@OneToMany*** annotation means that one user can have multiple collections and a collection can only belong to one user.

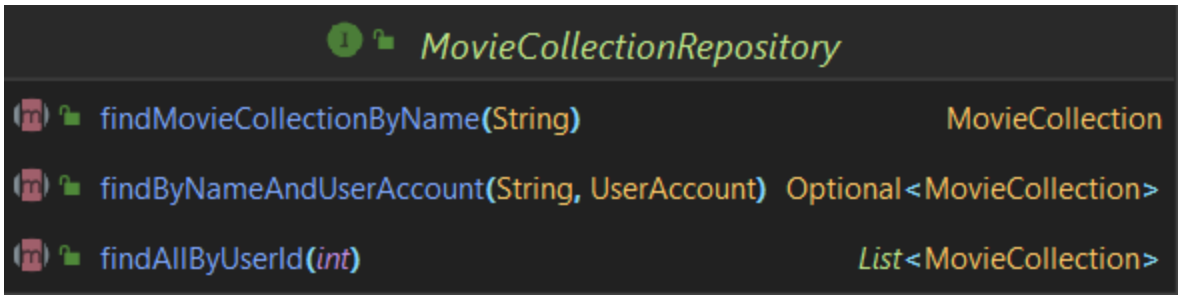


- **FriendList**: represent a list of friends the user has.
  - ***@ManyToMany*** annotation means that one user can have many friends and one user can be a friend to many users.
  - ***@JoinTable*** creates an intermediate table representing a many-to-many relationship
  - 

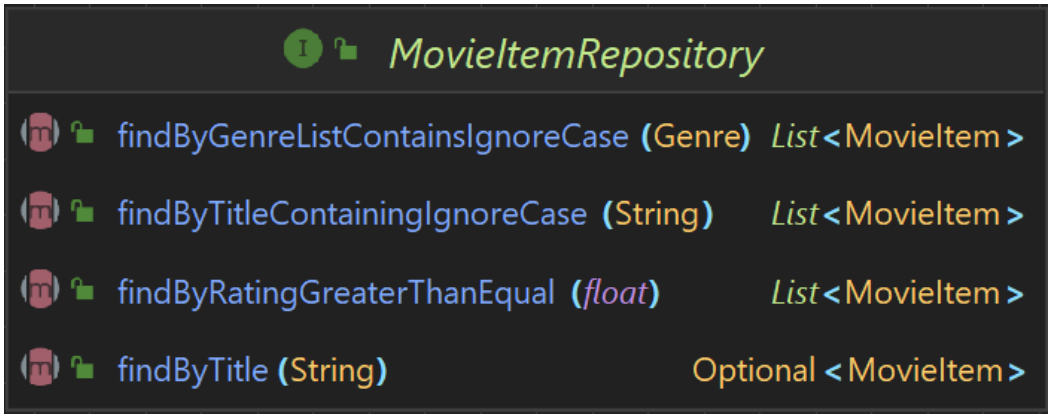
- **RatedMovies:** represents the movies that the user has rated.
  - ***@ManyToMany*** annotation means that many users can rate many movies and one movie can be rated by multiple users.
  - ***@JoinTable*** creates an intermediate table representing a many-to-many relationship.
  - 

- **MovieItem:** encapsulates details and information pertaining to a movie within the application. It serves as a structural blueprint to represent and manage specific movie-related data
  - **GenreList:** Represents a list of genres a movie can have.
    - ***@ElementCollection*** stores a collection of simple elements (in this case, Genre instances) rather than entities. Each element in this collection will correspond to a Genre value.
    - **@Enumerated(EnumType.STRING)** specifies that the Genre enum values should be stored as strings in the database, using the string representation of each enum value.

        - **@CollectionTable(name = "genres", joinColumns = @JoinColumn(name = "movie_item_id"))** configures the database table (genres) where the genres associated with each MovieItem will be stored, linking them through a column named movie_item_id to identify which MovieItem each genre belongs to.
    - **Genre:** enum that encapsulates distinct categories or genres attributed to movies.
- **Repository classes:** Implement the Repository Pattern and provide a set of methods to interact with the database using Spring Data JPA. They encapsulate the logic for data manipulation and querying.
    - **MovieCollectionRepository:** interface that serves as a repository for managing instances of the MovieCollection entity within the application. This repository interfaces with the underlying database using Spring Data JPA mechanisms.

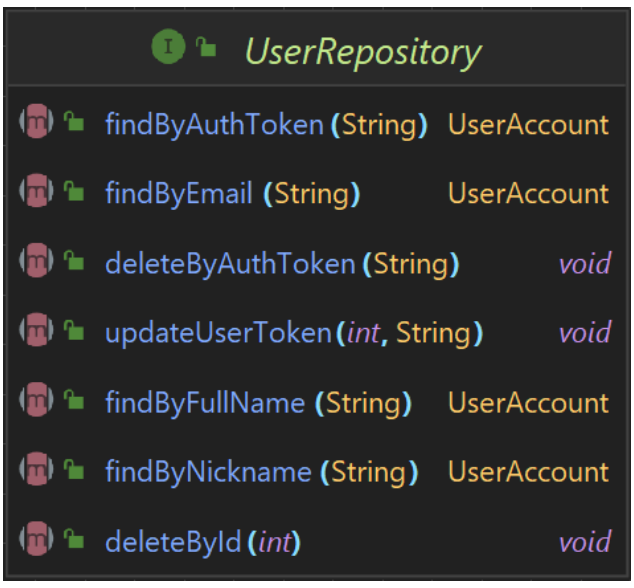


        - ***findMovieCollectionByName(String):*** method that retrieves a Movie collection from the database based on its name.
        - ***findByNameAndUserAccount(String, UserAccount):*** method that retrieves a collection from the database with a specific name which belongs to a specific user.
        - ***findAllByUserId(int):*** method that retrieves all the collections from the database that belong to a specific user
    - **MovieItemRepository:** interface that serves as a repository for managing instances of the MovieItem entity within the application. This repository interfaces with the underlying database using Spring Data JPA mechanisms

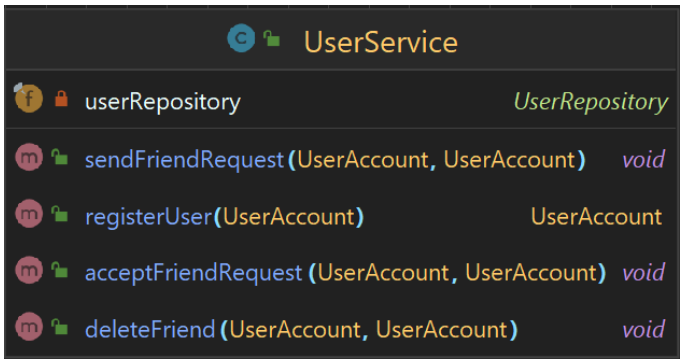


- ***findByGenreListContainsIgnoreCase(Genre):*** method that retrieves movies from the database based on the list of genres
- ***findByTitleContainingIgnoreCase(String)***: method that retrieves movies from the database that contain a specific sequence of characters
- ***findByRatingGreaterThanEqual(float):*** method that retrieves movies that has a greater or equal rating value indicated in the method signature

- **UserRepository:** interface that serves as a repository for managing instances of the UserAccount entity within the application. This repository interfaces with the underlying database using Spring Data JPA mechanisms

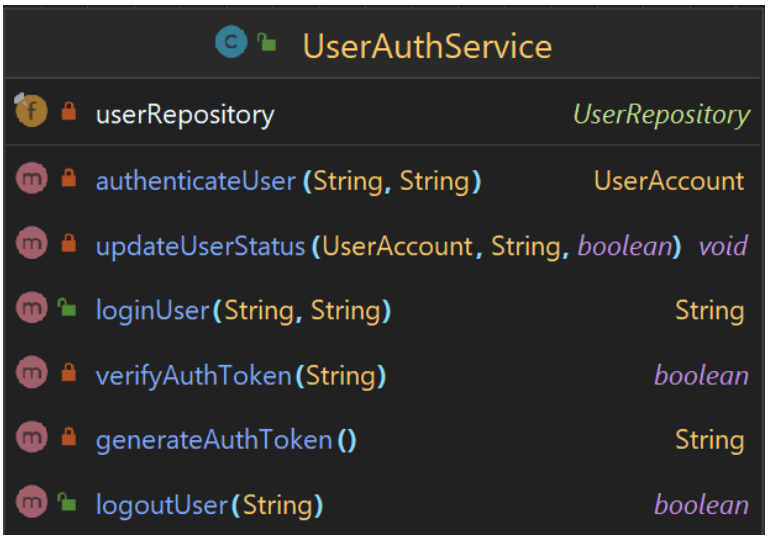


- ***Service classes:*** contain the business logic, coordinating operations between ***controllers*** and ***repositories***.
    - **UserService:** class that represents a business logic associated with the user



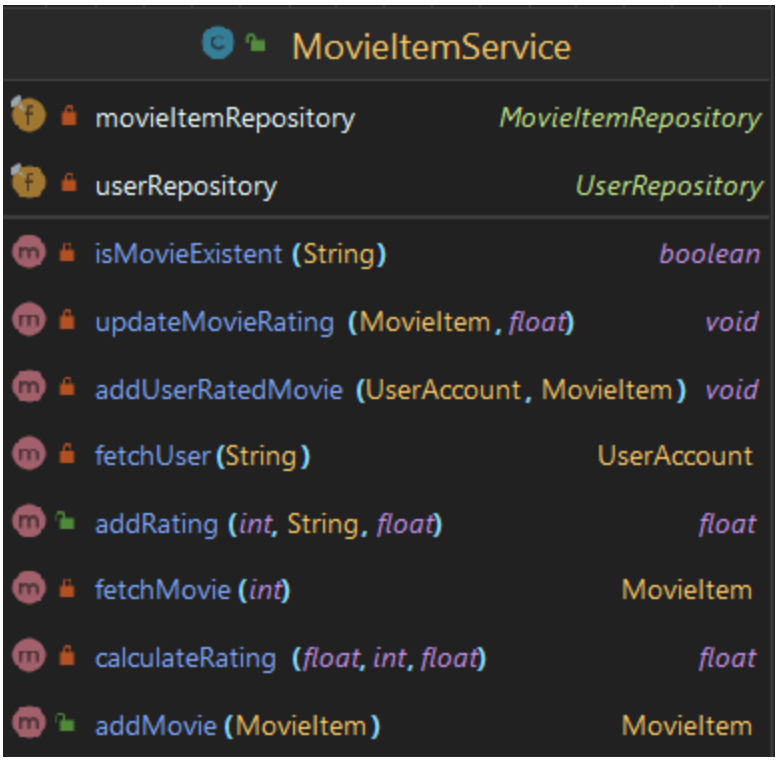

```
public UserAccount registerUser(UserAccount
userAccount)
```

        - Method that is responsible for registering a new user in the database.
        - Before registering a user it first checks if the user with such email or nickname exists.
    - ***sendFriendRequest()***, ***acceptFriendRequest()***, ***deleteFriend()*** methods are not yet implemented
    - **UserAuthService:** class that represents a business logic that is associated with the user authentication



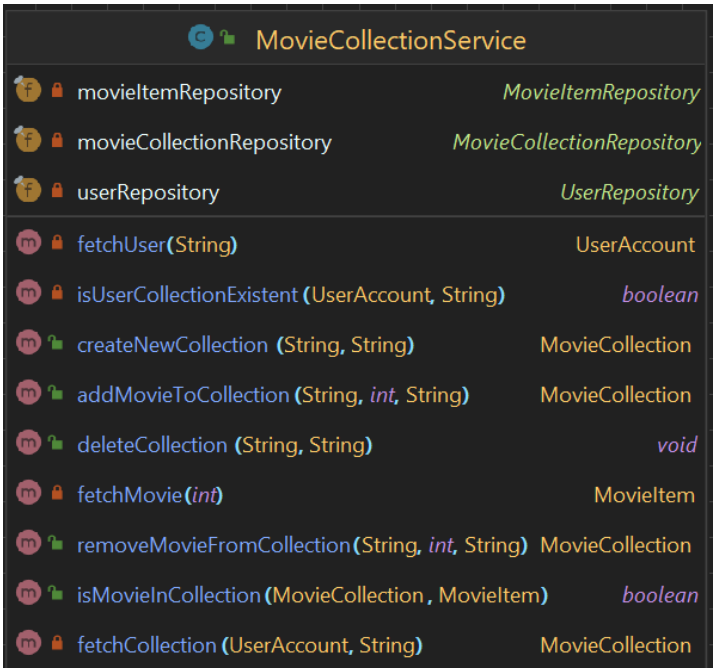
- `public String loginUser(String nickname, String password)`
    - Method that is responsible for logging in the user.
    - It first checks is the user with such nickname and password exists by using ***authenticateUser()*** method
    - If such a user exists then the authentication token is generated for this user by using ***generateAuthToken()*** method.
    - Method ***updateUserStatus()*** sets an authentication token for the user in the database and sets that the user is logged.
- `public boolean logoutUser(String authToken)`
    - Method that is responsible for logging out the user.
    - It invalidates the user's authentication token in the database and sets his logged in state to false by using ***updateUserStatus()*** method.

- **MovieItemService:** class that represents a business logic associated with the movie item



- `public MovieItem addMovie(MovieItem movieItem)`
  - Method that is responsible for adding a new movie to the database.
  - Before adding a movie, it checks if a movie with such a name already exists.
- `public float addRating(int movieId, String authToken, float rating)`
  - Method that is responsible for adding a rating on a movie.
  - It first checks if the user has already rated the movie before by checking the list of rated movies of the user.
  - If user hasn't rated the movie, it calculates a new rating with ***calculateRating()*** method
  - Then it updates the rating of a movie by inserting a new rating into the database with ***updateRating()*** method
  - It also adds this rated movie to the user's list of rated movies in the database by using ***addUserRatedMovie()*** method
- `private float calculateRating(float previousRating, int numVotes, float newRating)`
  - This method is used for calculating a new rating of a movie.
  - It multiplies the previous movie's rating by the number of users that rated the movie and adds a new rating to this value.
  - The new sum of ratings is then divided by the number of users that rated the movie increased by one to get a new average rating.

- **MovieCollectionService:** class that is responsible for business logic related to movie collections.




  - ```
    public MovieCollection createNewCollection(String authToken, String collectionName)
    ```
    - Method that creates a new collection for the user.
    - It accept authentication token of the user and the name of the collection and creates corresponding collection for the user
    - First the method checks if a collection with a similar name exists for the user in the database.
    - If such a collection doesn't exist, then we create a new collection for the user.
  - ```
    public MovieCollection addMovieToCollection(String authToken, int movieId, String collectionName)
    ```
    - Method that adds a movie to a collection with the specified name.
    - Method accepts token of the user, movie id and name of the collection

        - It first checks if the specified movie is already in a collection, and if it is not then we add this movie to the collection

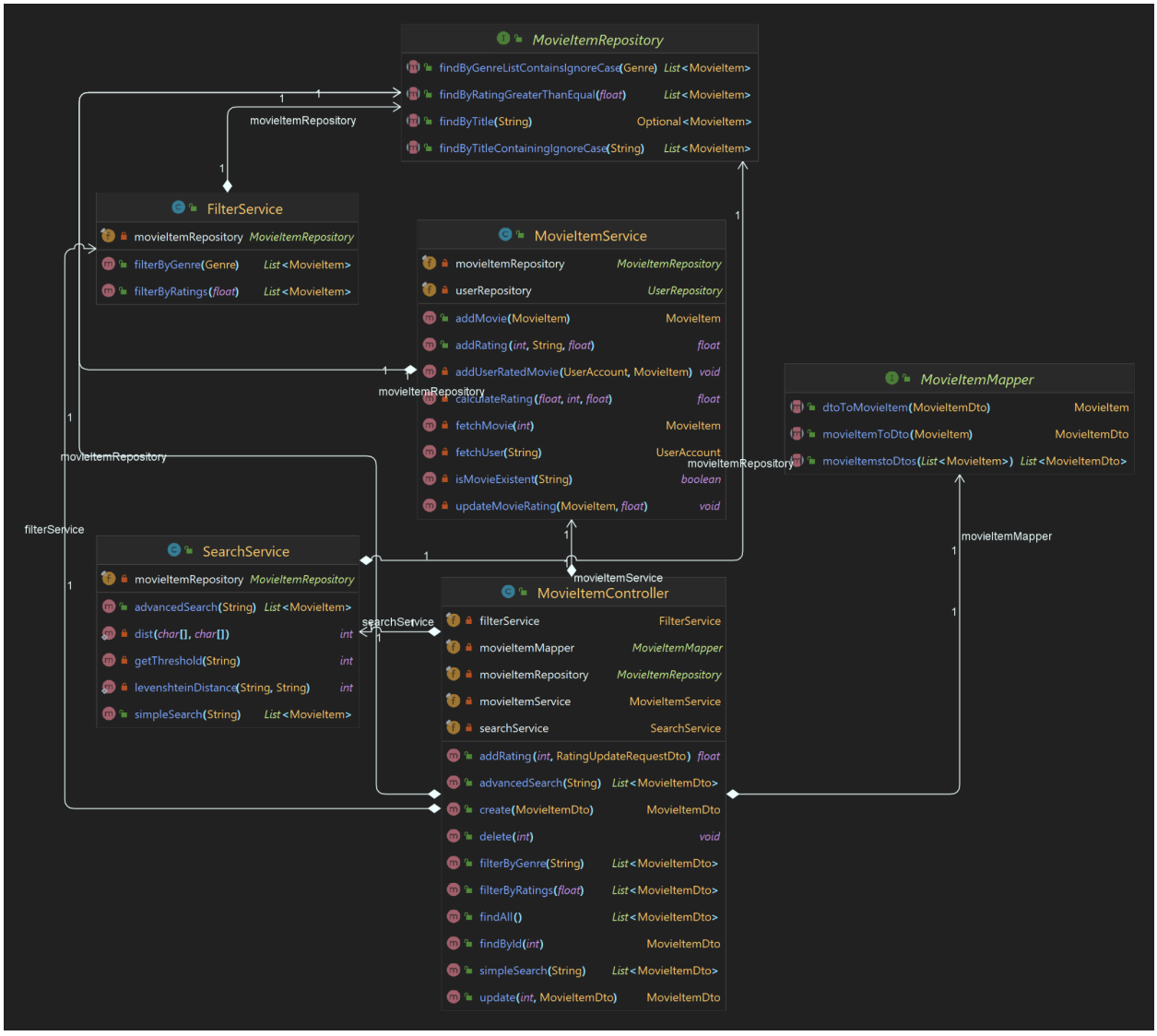
    - `public void deleteCollection(String authToken, String collectionName)`
        - Method that is used to delete a collection of the user

    - `public MovieCollection removeMovieFromCollection(String authToken, int movieId, String collectionName)`
        - This method is similar to ***addMovieToCollection()***, but it deletes a movie from a collection

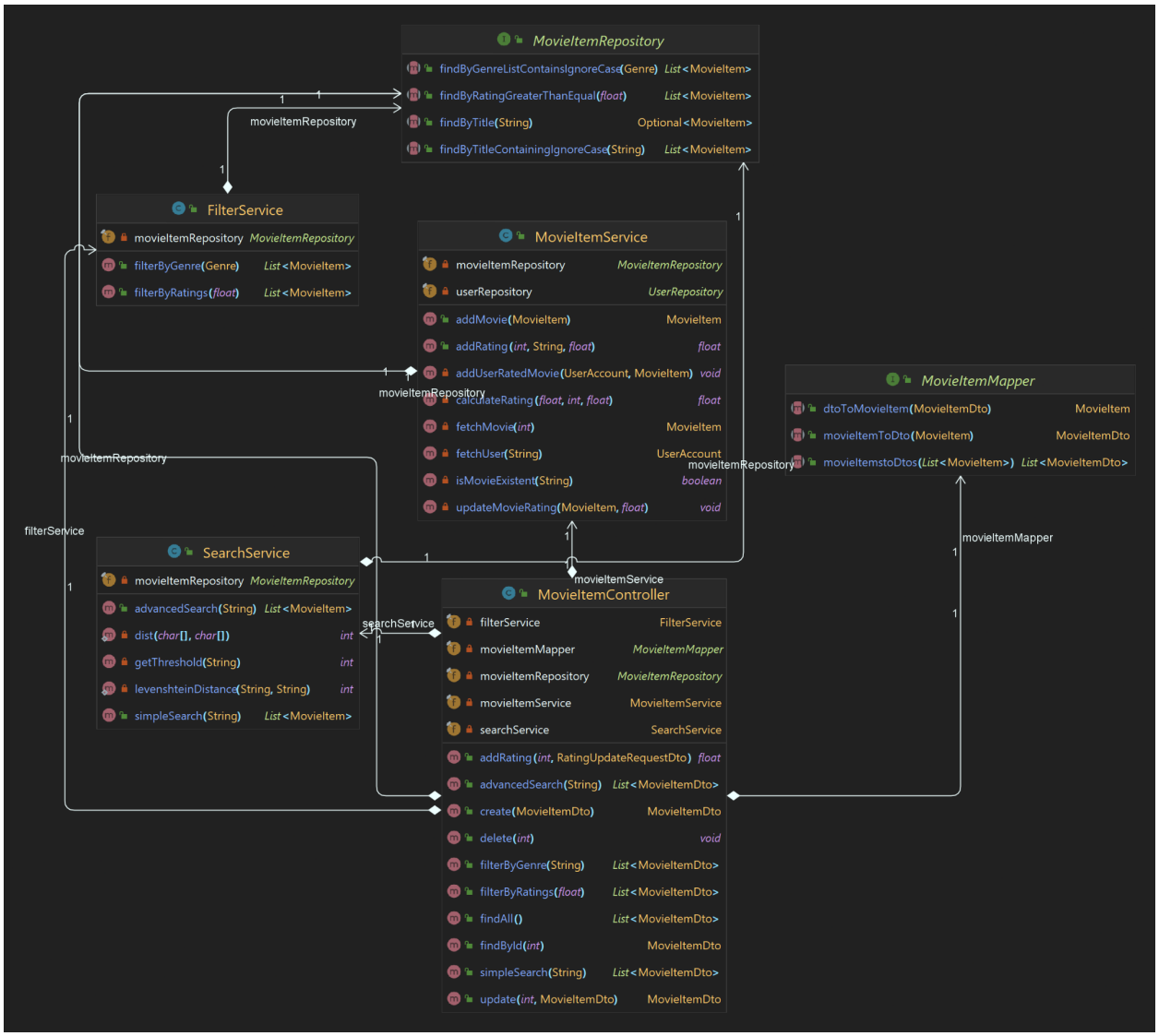
- **SearchService:** encapsulates essential business logic associated with searching for movies.
    - `public List<MovieItem> simpleSearch(String searchTerm)`
        - Conducts a case-insensitive search for movie titles containing the provided search term .
        - Returns a list of movie items having such substring.
    - `public List<MovieItem> advancedSearch(String searchTerm)`
        - Implements an advanced search algorithm based on Levenshtein distance. It calculates the distance between the provided search term and each movie title, considering a dynamic threshold based on the length of the movie name.
        - Returns a refined list of movie items that closely match the search term, ensuring that the returned movies are the most relevant.
- **FilterSearch:** manages the logic related to filtering movies based on specific criteria.
    - `public List<MovieItem> filterByGenre(Genre genre)`
        - Retrieves a list of movie items containing a specified genre. The search is case-insensitive.
        - Returns a filtered list of movie items in that genre.
    - `public List<MovieItem> filterByRatings(float minRating)`

- Retrieves a list of movie items with ratings equal to or higher than the provided minimum rating.

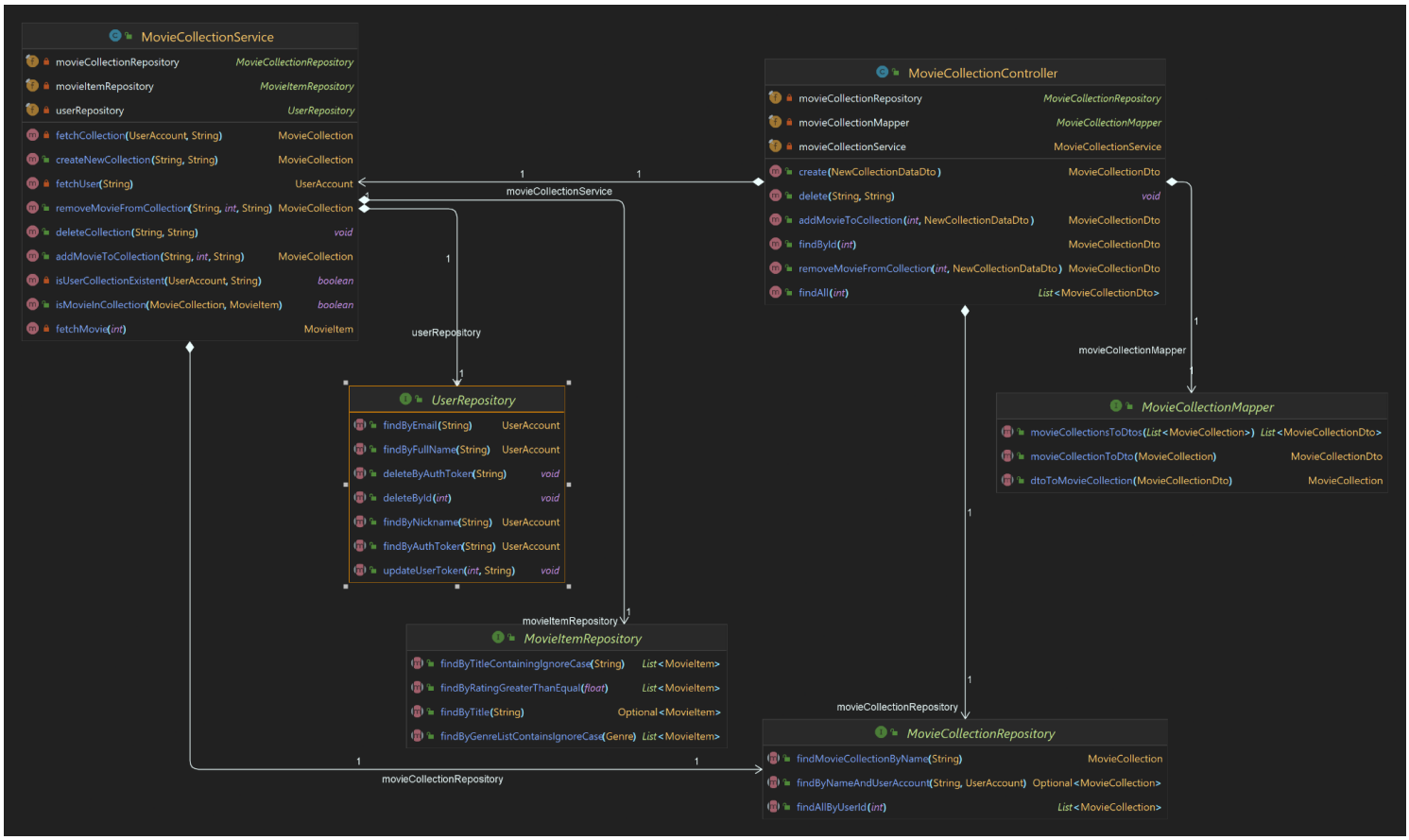
- **Controller classes**: serve as essential components responsible for handling incoming HTTP requests and coordinating interactions between the client and the backend services
    - **UserController:** orchestrates user-related functionalities within the application through a set of RESTful endpoints, facilitating user registration, authentication, modification, and retrieval operations.



- **MovieItemController:** orchestrates various HTTP endpoints to manage movie-related operations within the application. It interacts with services, repositories, and mappers to facilitate movie creation, retrieval, modification, and filtering.



- **MovieCollectionController:** manages endpoints for handling movie collection-related operations within the application. It interacts with services, repositories, and mappers to facilitate collection creation, modification, and retrieval.



- **CorsConfig:** Configuration class for CORS (Cross-Origin Resource Sharing) in a Spring Boot application.
  - This class ensures that the backend accepts requests from the specified frontend URL.
    - `config.addAllowedOrigin("http://localhost:3000");`
    - Allows requests from "http://localhost:3000"

- **Use of Data Transfer Objects (DTOs):**
    - **DTOs** alongside **Controller classes** alleviate complexities in data management within REST APIs by addressing issues related to circular references, data mapping, selective presentation, data aggregation, and fostering a flexible and decoupled architecture.

- **Mapper classes:** serves to facilitate the conversion and mapping of data between entities (database models) and data transfer objects (DTOs). It ensures a clean separation between the database model structure and the data presented to the frontend.
    - **UserAccountMapper:** This class maps between UserAccount entities and UserAccountDto data transfer objects.
        - It provides methods for mapping single entities, lists of entities, and DTOs to entities and vice versa.

```
@Mapping(target = "userAccountDto", ignore = true)
MovieCollectionDto movieCollectionToDto(MovieCollection collection);
```

        - The notation above indicates that the userAccountDto field in the MovieCollectionDto should be ignored during the mapping to get rid of the mapping loop.

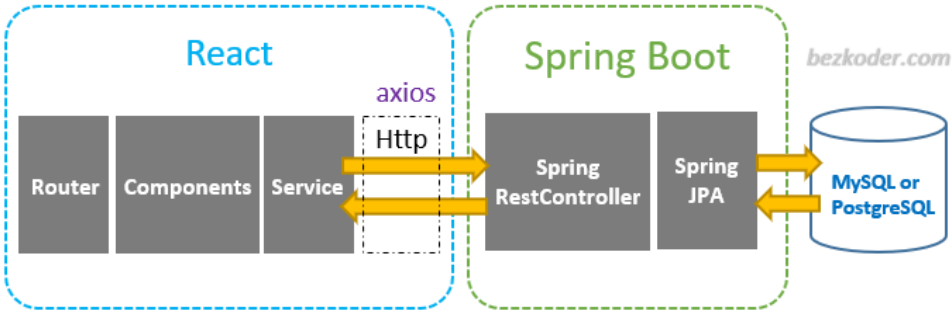
```
@Mapping(target = "ratedByUsers", ignore = true)
MovieItemDto movieItemToDto(MovieItem movieItem);
```

        - The notation above indicates that the ratedByUsers field in the MovieItemDto should be ignored during the mapping to get rid of the mapping loop.
    - **MovieItemMapper:** This class maps between MovieItem entities and MovieItemDto data transfer objects.
        - Similar to UserAccountMapper, it provides methods for mapping single entities, lists of entities, and DTOs to entities and vice versa.

- **MovieCollectionMapper**: This class maps between MovieCollection entities and MovieCollectionDto data transfer objects.
    - It provides methods for mapping single entities, lists of entities, and DTOs to entities and vice versa.

5. ARCHITECTURE

- **Overview**
    - The frontend, developed using React, serves as the user interface, offering an interactive and visually appealing experience. On the backend, Java with Spring Boot powers the server-side logic and data processing.
- **Frontend Backend interactions**



    - Axios, a promise-based HTTP client, is employed to enhance the interaction between the frontend and backend through the utilization of RESTful APIs. This ensures efficient communication, enabling the frontend to make requests for various actions such as user registration, movie rating, or collection creation. These requests are then transmitted to the backend via RESTful API endpoints.

- **Example of frontend sending post request using Axios to register new user:**

```
const onSubmit = async (e) => {
    e.preventDefault();
    await axios.post("http://localhost:8080/api/users/register", user);

    navigate("/");  //Home
};
```

- The backend processes these requests, leveraging a PostgreSQL database to store and retrieve user account details, movie information, and user-generated collections. The architecture follows a client-server model, where the React frontend acts as the client making requests, and the Java Spring Boot backend serves as the server processing these requests.
- **Continuation of the example above:**

```
@RestController
@RequestMapping("/api/users")
@RequiredArgsConstructor
public class UserController {
```

```
@PostMapping("/register")
public UserAccountDto register(@RequestBody UserAccountDto user) {
    user.setId(0);
    return userAccountMapper.userToDto(
            userService.registerUser(userAccountMapper.dtoToUser(user))
    );
}
```

6. TECHNOLOGIES USED

- **Backend:**
    - **Java Programming Language**
        - Used for backend implementation
    - **IntelliJ IDEA Community Edition 2022.1**
        - The development environment was IntelliJ IDEA, providing a rich set of tools for Java development.
    - **Maven**
        - Served as our project management and comprehension tool. It facilitated the build process, managing dependencies, and providing a standardized project structure.
    - **Spring Boot Framework**
        - Adopted to streamline the development of our backend services. It promotes convention over configuration, simplifying the setup and development of production-ready Spring applications.
    - **Lombok Java library**
        - Was employed to reduce boilerplate code in our project. It provided annotations to generate common methods like getters, setters, and constructors, enhancing code readability and maintainability.
    - **MapStruct Object mapping tool**
        - Simplifies the mapping between Java beans. It generated mapping code based on defined interfaces, enhancing the efficiency of our data transformation processes.
- **Database:**
    - **REST API**
        - Used for providing a scalable and stateless communication interface. This design choice facilitated interoperability between different components of our application.
    - **PostgreSQL**

        - It is a powerful open-source relational database, used for data storage. It offered reliability, extensibility, and compliance with SQL standards, ensuring robust and efficient data management.
    - **Postman**
        - Served as a valuable tool for testing and validating our RESTful APIs. It helps to interact with backend controllers, sending requests and examining responses to ensure correct functionality.
- **Frontend:**
    - **ReactJS**
        - A JavaScript library for building user interfaces.
    - **Visual Studio Code**
        - Provided a lightweight and powerful integrated development environment (IDE) for React development.
    - **JavaScript, CSS, HTML**
        - These core web technologies were utilized to build the frontend user interface. JavaScript provided dynamic interactivity, CSS styled the visual elements, and HTML structured the content, collectively delivering a modern and user-friendly web experience.

7. TESTING

Unit tests:

- In the unit testing phase, the application's modules were scrutinized individually to ensure their functionality and correctness. Mockito, a mocking framework, played a crucial role in simulating repository interactions and isolating unit tests from external dependencies.

Test class	Test signature	Description
MovieItemServiceTest	public void testAddRating()	Validates the functionality of adding a new movie rating. Verifies the correct calculation of the updated rating.
	public void testAddMovie()	Validates the functionality of adding a new movie into a collection. Ensures the correct saving of movie properties.
UserServiceTest	public void testRegisterUser_Success()	Validates the successful registration process of the user. Ensures the correct saving of new users to the database.
	public void testRegisterUser_Fail()	Validates the failing registration process of the user. Ensures the user is not created and added to the database

Integrated Tests

- Integration tests were conducted to assess the collaborative behavior of multiple units, ensuring they seamlessly work together to accomplish broader functionalities.

Test class	Test signature	Description
MovieCollectionServiceIT	public void testAddRemoveMovieFromCollection()	Validates that movie successfully added and removed from collection. Ensures that movie indeed in such collection and vice versa
	public void testCreateDeleteNewCollection()	Validates the creation and deletion collection. Ensures that the user has a new collection in the database and can remove it.
SearchFilterIT	public void testSearch()	Validates the correctness of simple search and advances search of movies. Ensures simple search returns the movies with such substring, while the advanced can search movie with typo.
	public void testFilter()	Ensures the correctness of filtering methods. Such as by genre and by minimum rating.
UserAuthServiceIT	void testLoginSuccess()	Validates the user can successfully log in.

		Verifies status of user changed and authentication token created.
	void testLoginFailPassword()	Validates the user can fail to log in. Verifies that ResponseStatusExceptio n was thrown, the status of the user has not changed and authentication token was not created.
	void testLogoutSuccess()	Validates the user can successfully log out. Verifies status of user changed and authentication token destroyed.
	void testLogoutNonexistentUser()	Validates the user can fail in log out. Verifies that the NullPointerException throws since there is no such user.

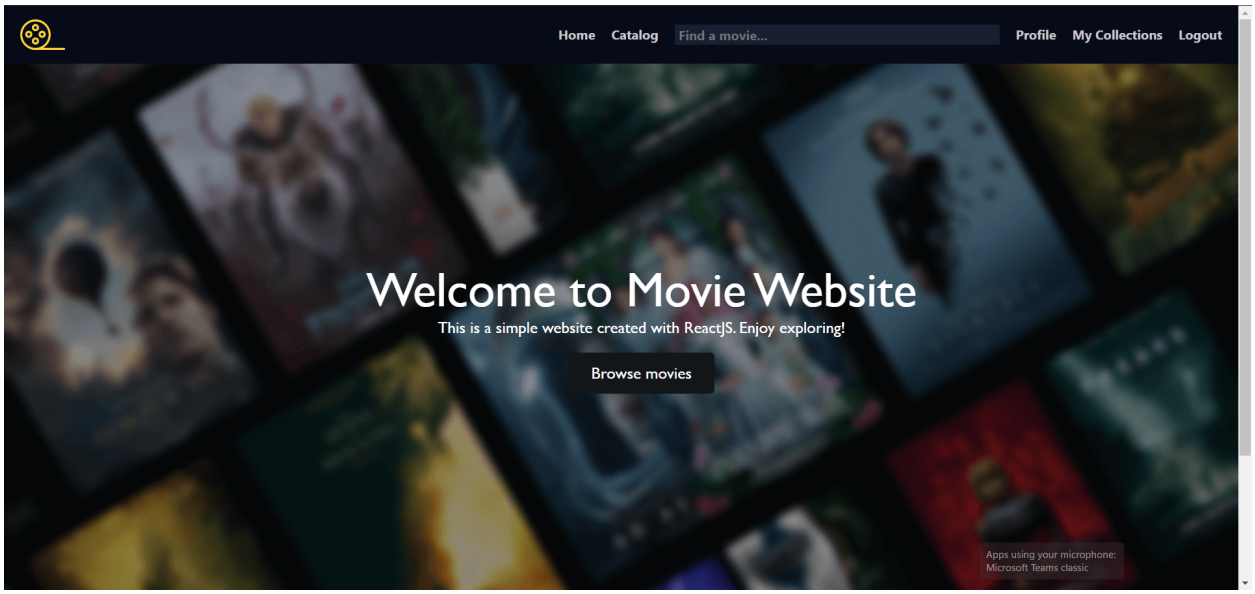
8. PROJECT DEPLOYMENT GUIDE

- **Prerequisites**:
    - Install JDK 20.
    - Setup Intellij IDEA.
- **Configuration**:
    - Ensure your *application.properties* file in the backend contains the correct database connection details:
    - ***spring.datasource.url=jdbc:postgresql://localhost:5432/DBName***
    - ***spring.datasource.username=postgres***
    - ***spring.datasource.password=YourPassword***
- **Start Backend:**
    - Open your project in Intellij IDEA.
    - Run your SpringBoot application to start the backend server.
- **Frontend (ReactJS):**
    - **Prerequisites:**
        - Install Node.js.
        - Set up Visual Studio Code.
    - **Dependencies Installation:**
        - Open a terminal in the frontend folder.
        - Run **npm install** to install all necessary dependencies (Bootstrap, npm packages, etc.).
    - **Start Frontend:**
        - In the terminal, run ***npm start*** to initiate the React application.
        - The frontend server will start, and the React application will be accessible.
- **Database (PostgreSQL with pgAdmin):**
    - Database Setup:
        - Ensure PostgreSQL is installed and running.
        - Use pgAdmin to manage your PostgreSQL database.
        - Create a database.
        - **Connect Backend to Database:**

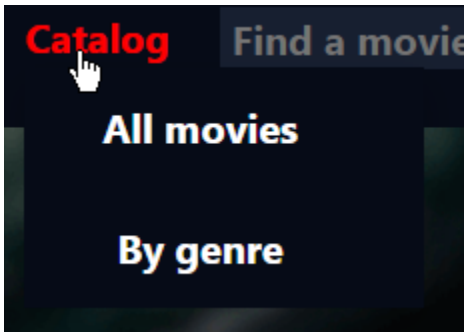
        - Verify that the connection details in application.properties match your PostgreSQL database configurations.
- **Deployment Workflow:**
    - Backend:
        - Start your SpringBoot application in Intelij IDEA to activate the backend server.
- **Frontend:**
    - Launch the React application by running ***npm start*** in the terminal within the frontend directory.
- **Database:**
    - Ensure that your PostgreSQL database is running and properly configured through pgAdmin.
- **Access the Application:**
    - Access your web application via a web browser using the appropriate URL or port number where the frontend server is running.

9. WEBSITE SCREENSHOTS

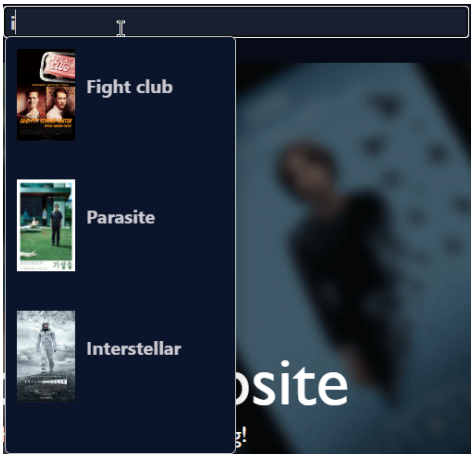
Main page:



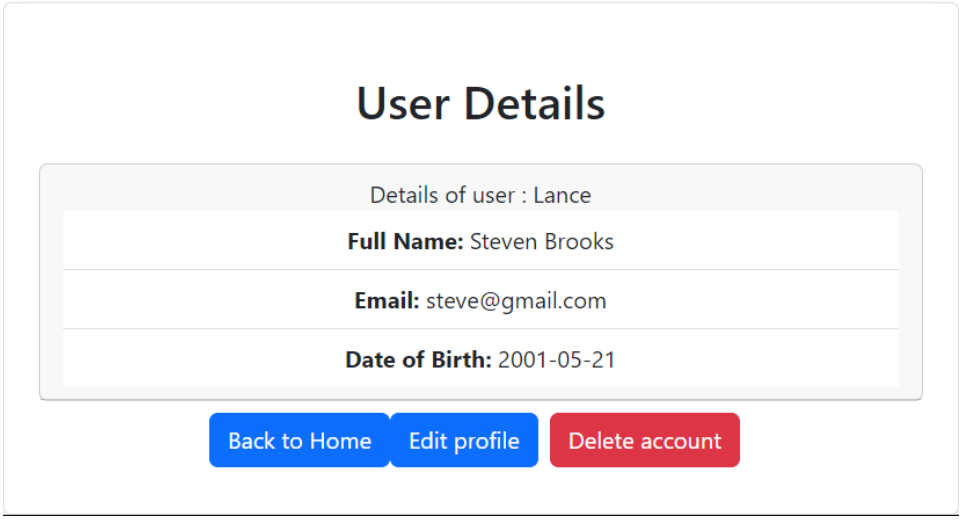
Catalog dropdown:



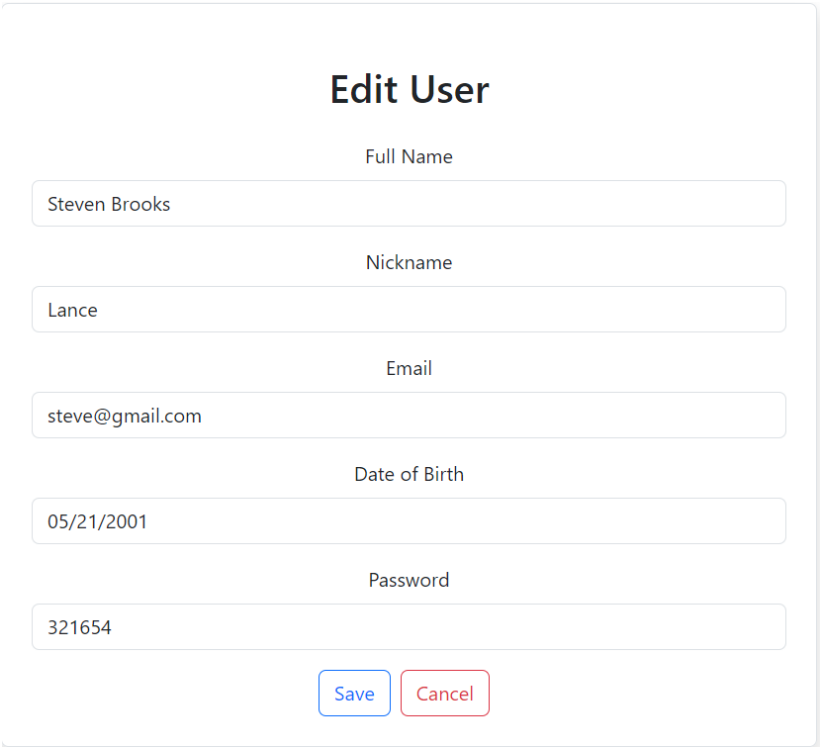
Search:



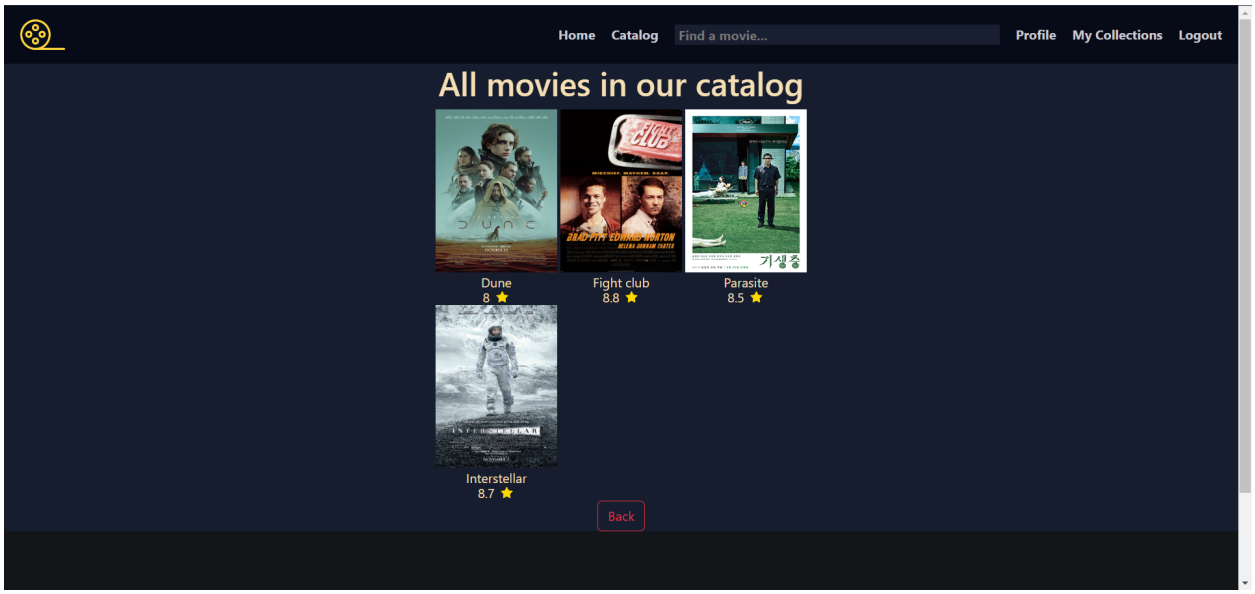
User's profile information page:



Edit user's profile information page:

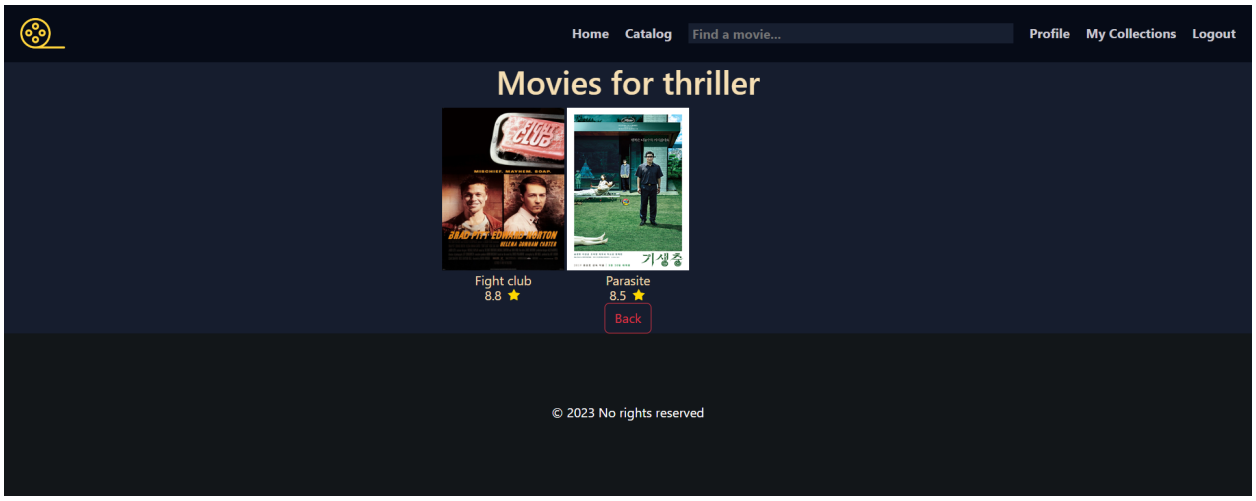


All movies page:

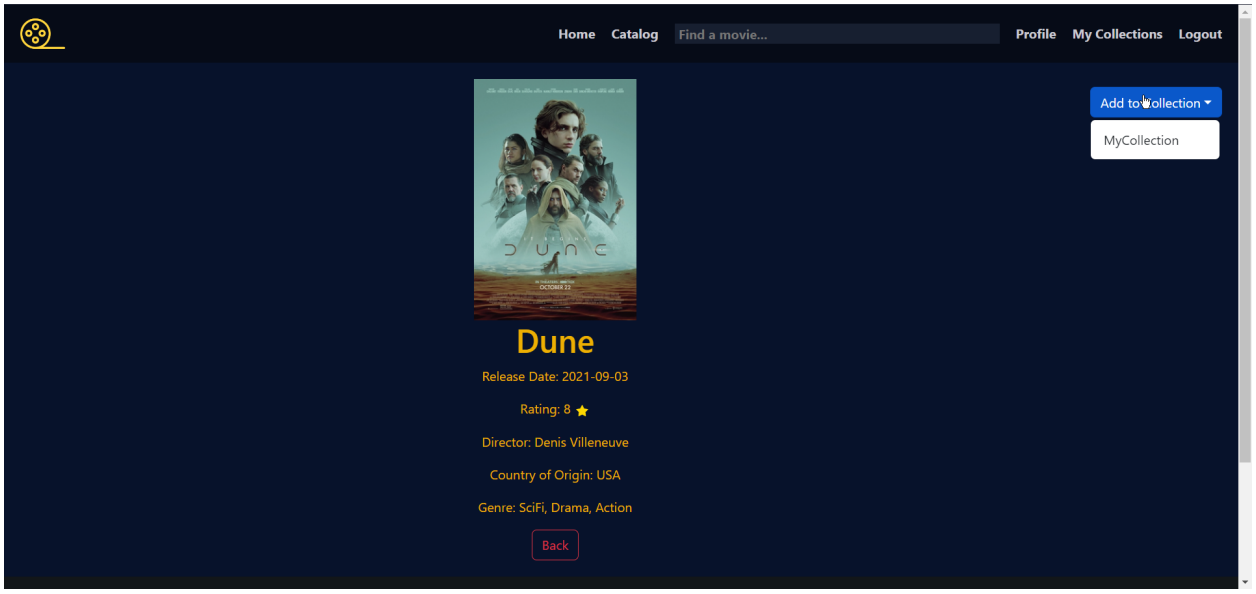


Categories page:

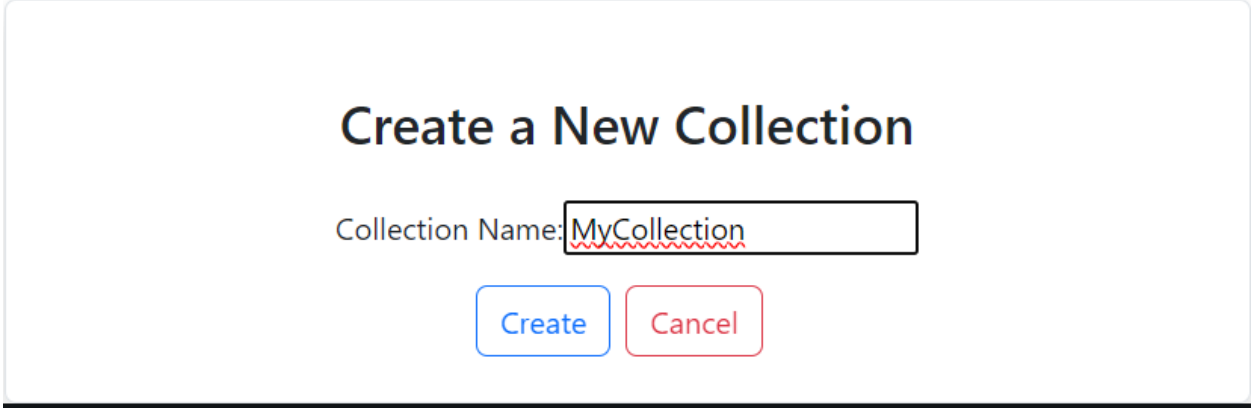
Movies in “thriller” category:



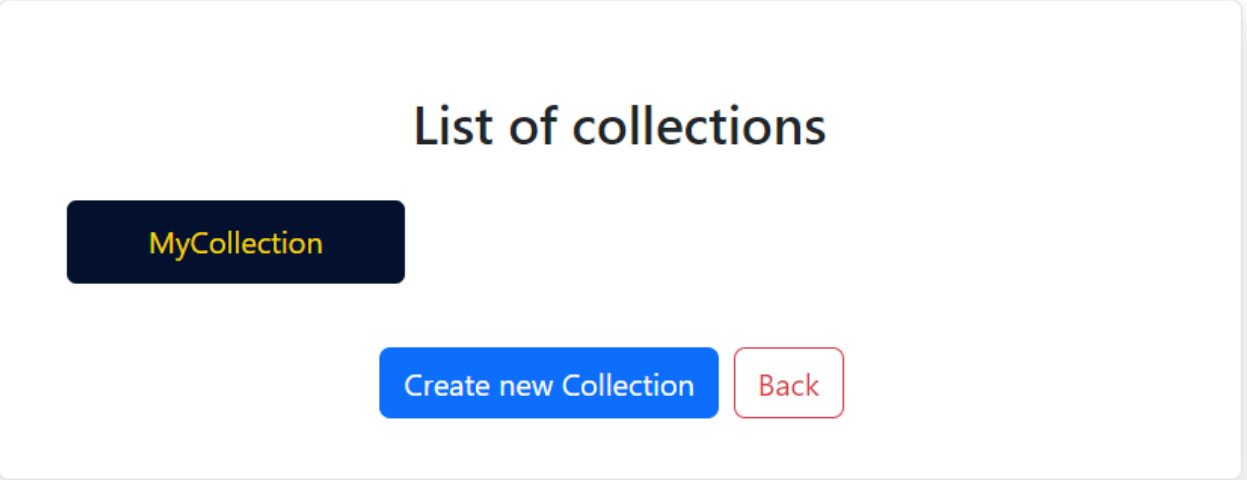
Movie description page:



Create collection:



List of collections page:



Movies in collection: