Software Requirements Specification

for

<MOVIE RECOMMENDATION SYSTEM>

Prepared by Tinagaran Subramanian

10 February 2023

Content

1. Introduction

1.1 Document Purpose

This System Requirements Specification (SRS) outlines the requirements for a Movie Recommendation System that utilizes Cosine Similarity and Pearson Correlation as its recommendation algorithm. It also provides details on the project's scope, features, and design to ensure an efficient and well-managed completion and serve as a reference for future use.

1.2 Project Scope

The purpose of this movie recommendation system is to offer customized movie suggestions to users based on their previous movie ratings. The system will analyze the movie ratings given by different users and determine the similarity of their preferences through the use of cosine similarity and Pearson's similarity. By doing this, the system can suggest new movies that are likely to be appealing to each user based on their prior ratings.

The movie recommendation system will gather movie ratings data from users to form a matrix and use Cosine similarity and Pearson Correlation to compare the ratings given to different movies by various users. This will enable the system to identify movies that have received high ratings from users with similar preferences to the target user and make recommendations for movies that are likely to be of interest. By utilizing both cosine similarity and Pearson's similarity, the system will be able to provide a thorough and precise evaluation of movie preferences, taking into account both individual user ratings and general trends among the larger user group.

1.3 Intended Audience and Reading Suggestions

This System Requirements Specification (SRS) document is meant for various groups of people, including project managers, software engineers, quality control teams, business executives, and even movie enthusiasts who are concerned with the creation and execution of the Movie Recommender System.

2. Overall Description

2.1 Product Perspective

This product is a web-based application system that recommends movies to users based on the similarity score between other movies. Users can create and login to their account. Users can view a list of movies with detailed information. They can add a movie to their favorites and remove this if they want. Users can also view recommended films based on Cosine Similarity or Pearson's Correlation. Users can also rate the movies that they watched and update the ratings.

The application has an admin view for managing the movies and users. Admins can add and view movies, create and view user accounts, and create and view admin accounts.

2.2 Product Features

The Movie Recommender System features the following features.

For users:

Ability to create an account
Ability to log in to their account
Ability to view a list of movies
Ability to search the list of movies
Ability to add movies to their favorites
Ability to view their favorite movies
Ability to remove movies from their favorites
Ability to rate and update movie ratings
Ability to view their rated movies
Ability to view recommended movies based on the movie ratings, using either Cosine Similarity or Pearson correlation.

For admins:

Ability to log in to their account
Ability to view and add movies to the system
Ability to view and add users
Ability to view and add admins accounts

2.3 User Classes and Characteristics

Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the favored user classes from those who are less important to satisfy.

3. Similarity Matrix Algorithm

There are 2 similarity matrix algorithms used to recommend movies in this system which are Cosine Similarity and Pearson Correlation.

3.1 Cosine Similarity

Cosine similarity is a common metric used to compare the similarity between two items, such as movies. It is based on the cosine of the angle between two vectors in a multidimensional space, where each dimension represents a particular aspect of the item being compared.

In the context of movie recommendations, it is used to find the similarity between two movies based on the ratings they receive.

To calculate the cosine similarity between two movies, the ratings given to each movie by different users are represented as vectors. The similarity score is then calculated as the cosine of the angle between these vectors. The cosine similarity score ranges from 0 to 1, with a score of 1 indicating that the two movies are highly similar to one another. On the other hand, a score of 0 indicates that the two movies have no similarities.

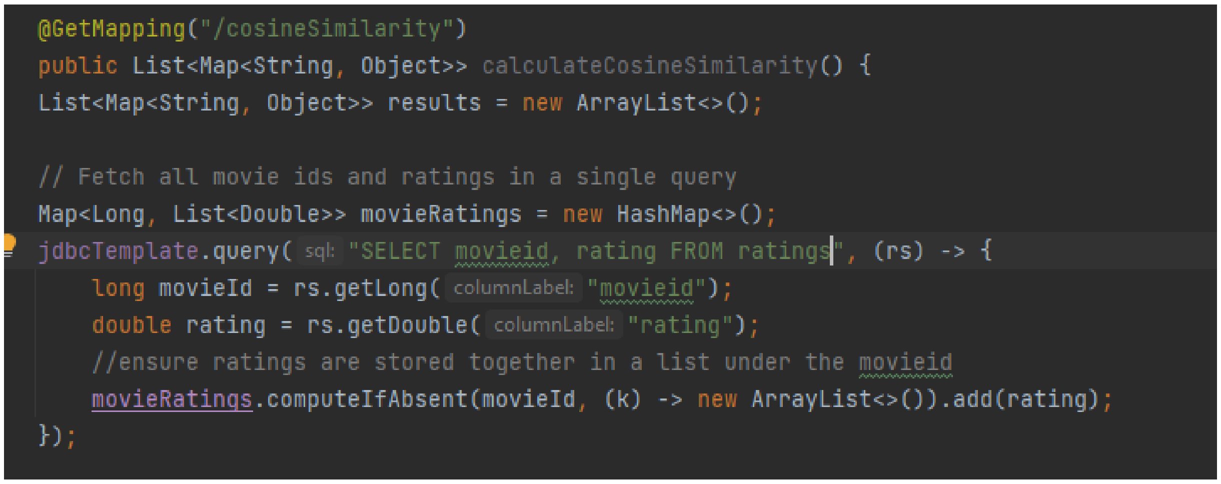
This method is useful in making movie recommendations as it helps to identify movies that are rated similarly by the same group of users. For example, if two movies receive high ratings from the same group of users, they will likely have a high cosine similarity score and may be recommended to a user who has rated the first movie highly.

In summary, cosine similarity is a way of comparing the ratings of two movies to determine their similarity score, which can range from 0 to 1. The higher the score, the more similar the ratings of the two movies are, which may suggest that a user who likes one movie will also like the other.

The formula of cosine similarity is below:

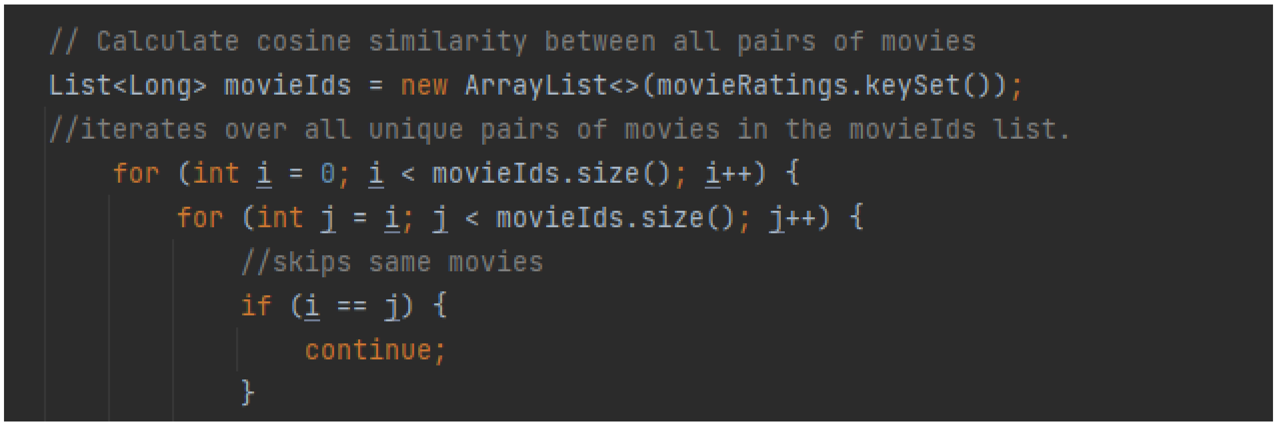
$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

This is how the Cosine Similarity Matrix is created for this Movie Recommender System.



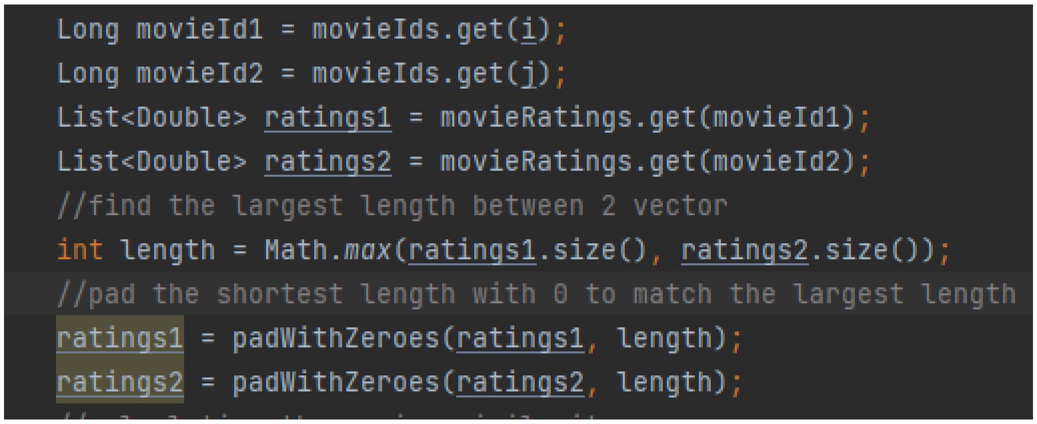
```java
@GetMapping("/cosineSimilarity")
public List<Map<String, Object>> calculateCosineSimilarity() {
List<Map<String, Object>> results = new ArrayList<>();

// Fetch all movie ids and ratings in a single query
Map<Long, List<Double>> movieRatings = new HashMap<>();
jdbcTemplate.query( sql: "SELECT movieid, rating FROM ratings", (rs) -> {
    long movieId = rs.getLong( columnLabel: "movieid");
    double rating = rs.getDouble( columnLabel: "rating");
    //ensure ratings are stored together in a list under the movieid
    movieRatings.computeIfAbsent(movieId, (k) -> new ArrayList<>()).add(rating);
});
```

Dataset of ratings for the movies are gathered which is rated by many users. For this system, Grouplens dataset is used. It is then fetched into a RestController API. The movie ids are grouped together and all the ratings are stored according to the movie id. It is to ensure that there are no duplicate values of movie id.

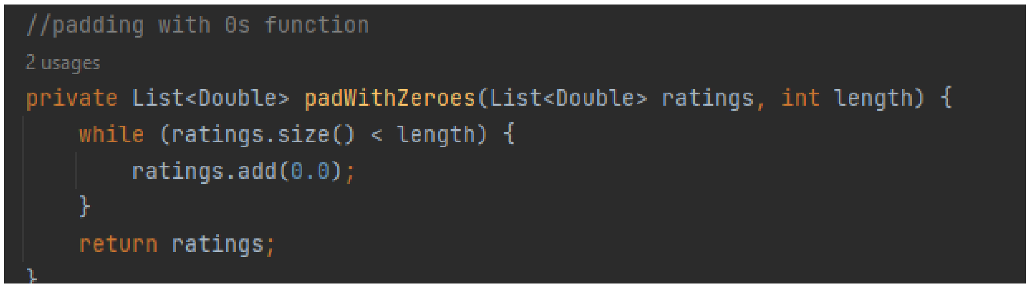


```java
// Calculate cosine similarity between all pairs of movies
List<Long> movieIds = new ArrayList<>(movieRatings.keySet());
//iterates over all unique pairs of movies in the movieIds list.
    for (int i = 0; i < movieIds.size(); i++) {
        for (int j = i; j < movieIds.size(); j++) {
            //skips same movies
            if (i == j) {
                continue;
            }
```

The cosine similarty calculation is started here. First, a nested loop iterates over all unique pairs of movies in the movieIds list. The outer loop runs from 0 to movieIds.size(), and the inner loop runs from i to movieIds.size(). If i and j are equal, the iteration skips to the next iteration using the continue statement, since cosine similarity between a movie and itself is not meaningful.



```java
Long movieId1 = movieIds.get(i);
Long movieId2 = movieIds.get(j);
List<Double> ratings1 = movieRatings.get(movieId1);
List<Double> ratings2 = movieRatings.get(movieId2);
//find the largest length between 2 vector
int length = Math.max(ratings1.size(), ratings2.size());
//pad the shortest length with 0 to match the largest length
ratings1 = padWithZeroes(ratings1, length);
ratings2 = padWithZeroes(ratings2, length);
```



```java
//padding with 0s function
2 usages
private List<Double> padWithZeroes(List<Double> ratings, int length) {
    while (ratings.size() < length) {
        ratings.add(0.0);
    }
    return ratings;
}
```

In this section, we are inside the loop where it gets the ratings for first movie (movieId1) and second movie (movieId2) to calculate the cosine similarity between each other.

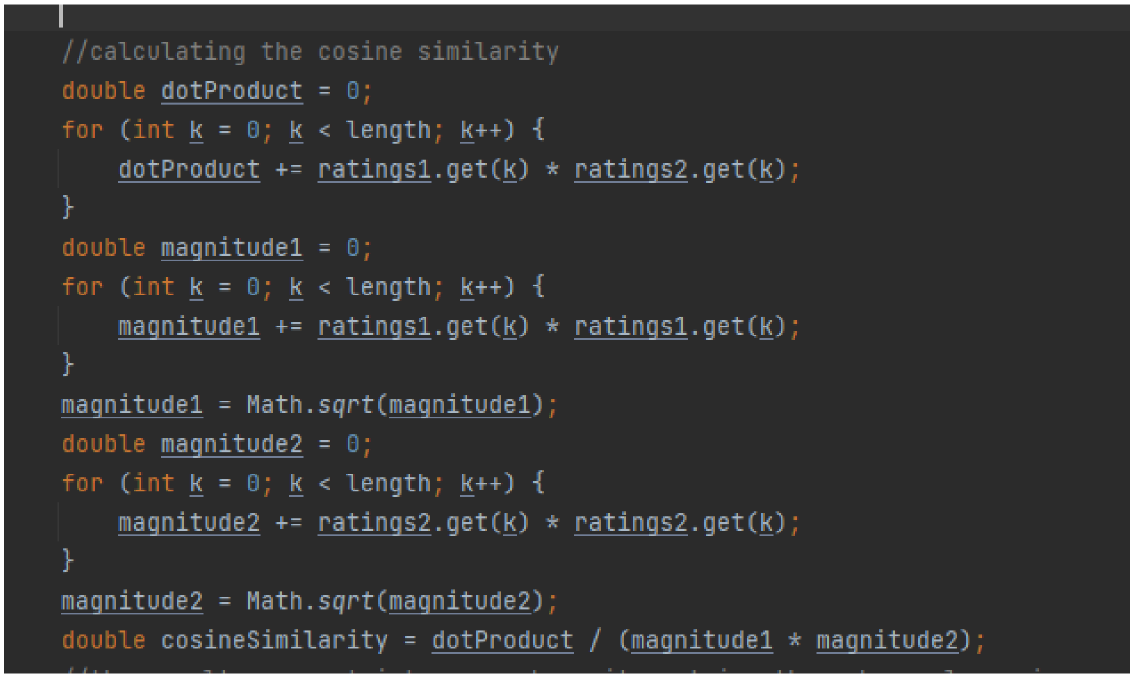
The length of the two lists of ratings for the two movies are calculated and sets length to the maximum of these two lengths.

If needed, the shorter list of ratings is padded with zeroes, so that both lists have the same length as specified by length. If this is not implemented, the similarity score will not be accurate because it will only calculate up till the shortest list length.

For example:

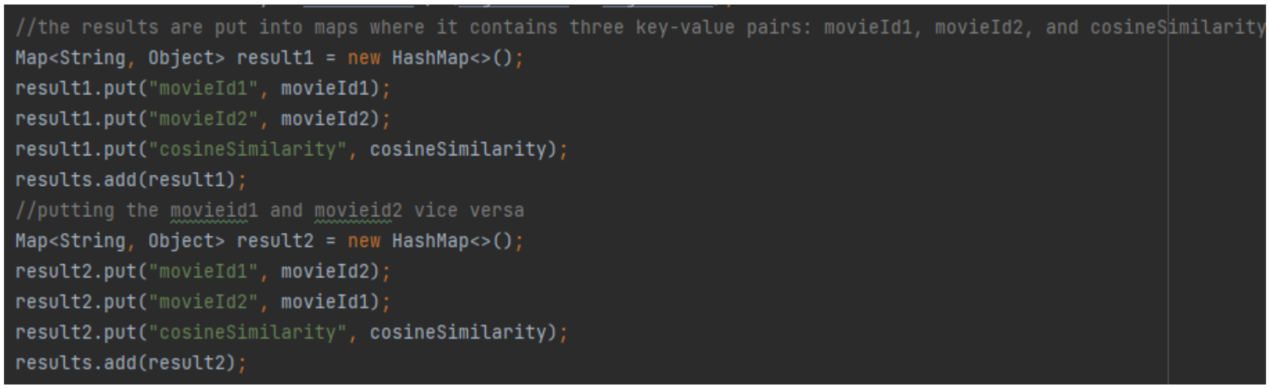
If movied1 ratings are (1.4.3,5,3) and movieId2 ratings are (3.4,5), the ratings that will be taken for the cosine similarity calculation are movieId1(1,4,3) and movieId2 (3.4,5).

If padded with zeroes, it will be movieId1(1,4,3, 5,3) and movieId2 (3.4,5,0,0).



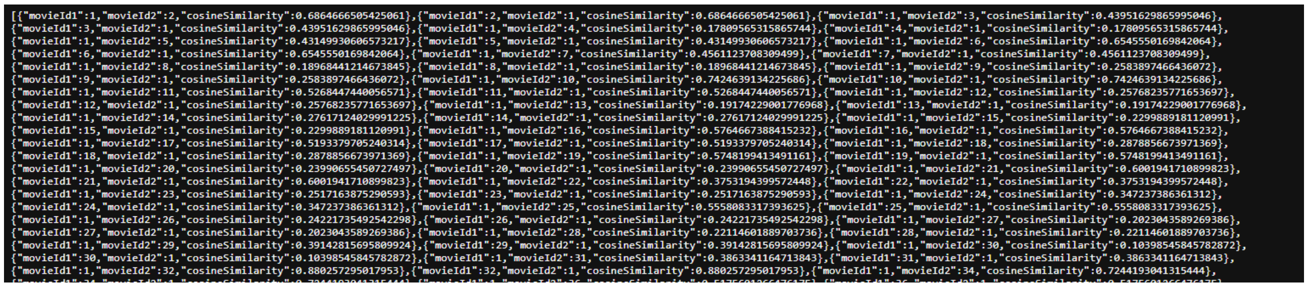
```
//calculating the cosine similarity
double dotProduct = 0;
for (int k = 0; k < length; k++) {
    dotProduct += ratings1.get(k) * ratings2.get(k);
}
double magnitude1 = 0;
for (int k = 0; k < length; k++) {
    magnitude1 += ratings1.get(k) * ratings1.get(k);
}
magnitude1 = Math.sqrt(magnitude1);
double magnitude2 = 0;
for (int k = 0; k < length; k++) {
    magnitude2 += ratings2.get(k) * ratings2.get(k);
}
magnitude2 = Math.sqrt(magnitude2);
double cosineSimilarity = dotProduct / (magnitude1 * magnitude2);
```

The next few lines of code calculate the dot product of the two lists of ratings, which is a measure of the angle between the two vectors in a high-dimensional space where each dimension corresponds to a movie rating. The dot product is the sum of the products of each pair of ratings.
Then, it calculate the magnitude of each vector, which is a measure of the length of each vector. The magnitude is the square root of the sum of the squares of each rating. Then it calculates the cosine similarity between the two vectors as the ratio of their dot product to the product of their magnitudes.



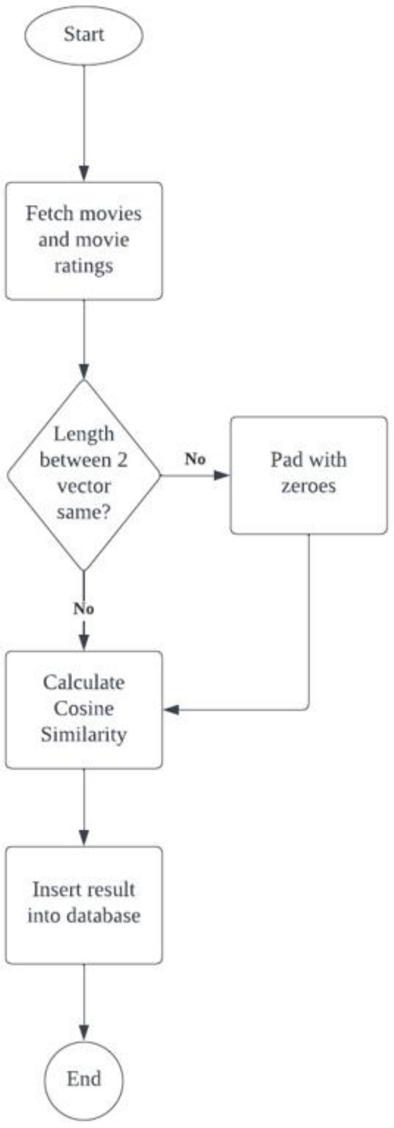
```
//the results are put into maps where it contains three key-value pairs: movieId1, movieId2, and cosineSimilarity
Map<String, Object> result1 = new HashMap<>();
result1.put("movieId1", movieId1);
result1.put("movieId2", movieId2);
result1.put("cosineSimilarity", cosineSimilarity);
results.add(result1);
//putting the movieid1 and movieid2 vice versa
Map<String, Object> result2 = new HashMap<>();
result2.put("movieId1", movieId2);
result2.put("movieId2", movieId1);
result2.put("cosineSimilarity", cosineSimilarity);
results.add(result2);
```

Finally, the code adds two maps to the results list, one with movieId1 and movieId2 as the keys and their respective cosine similarity as the value, and another map with movieId2 and movieId1 as the keys and the same cosine similarity value. The higher the cosine similarity score, the more similar the movies are to each other.The results can be saved into a Oracle database for retrieval or can be fetched from the RestController API.



```
[{"movieId1":1,"movieId2":2,"cosineSimilarity":0.6864666505425061},{"movieId1":2,"movieId2":1,"cosineSimilarity":0.6864666505425061},{"movieId1":1,"movieId2":3,"cosineSimilarity":0.4395162986599504 6},
{"movieId1":3,"movieId2":1,"cosineSimilarity":0.43951629865995046},{"movieId1":1,"movieId2":4,"cosineSimilarity":0.17809565315865744},{"movieId1":4,"movieId2":1,"cosineSimilarity":0.17809565315865744},
{"movieId1":1,"movieId2":5,"cosineSimilarity":0.4314993060657321 7},{"movieId1":5,"movieId2":1,"cosineSimilarity":0.43149930606573217},{"movieId1":1,"movieId2":6,"cosineSimilarity":0.654555016984206 4},
{"movieId1":6,"movieId2":1,"cosineSimilarity":0.6545550169842064},{"movieId1":1,"movieId2":7,"cosineSimilarity":0.4561123708309499},{"movieId1":7,"movieId2":1,"cosineSimilarity":0.4561123708309499},
{"movieId1":1,"movieId2":8,"cosineSimilarity":0.18968441214673845},{"movieId1":8,"movieId2":1,"cosineSimilarity":0.18968441214673845},{"movieId1":1,"movieId2":9,"cosineSimilarity":0.2583897466436072},
{"movieId1":9,"movieId2":1,"cosineSimilarity":0.2583897466436072},{"movieId1":1,"movieId2":10,"cosineSimilarity":0.742463913422568 6},{"movieId1":10,"movieId2":1,"cosineSimilarity":0.7424639134225686},
{"movieId1":1,"movieId2":11,"cosineSimilarity":0.5268447440056571},{"movieId1":11,"movieId2":1,"cosineSimilarity":0.5268447440056571},{"movieId1":1,"movieId2":12,"cosineSimilarity":0.25768235771653697},
{"movieId1":12,"movieId2":1,"cosineSimilarity":0.25768235771653697},{"movieId1":1,"movieId2":13,"cosineSimilarity":0.19174229001776968},{"movieId1":13,"movieId2":1,"cosineSimilarity":0.19174229001776968},
{"movieId1":1,"movieId2":14,"cosineSimilarity":0.27617124029991225},{"movieId1":14,"movieId2":1,"cosineSimilarity":0.27617124029991225},{"movieId1":1,"movieId2":15,"cosineSimilarity":0.2299889181120991},
{"movieId1":15,"movieId2":1,"cosineSimilarity":0.2299889181120991},{"movieId1":1,"movieId2":16,"cosineSimilarity":0.5764667388415232},{"movieId1":16,"movieId2":1,"cosineSimilarity":0.5764667388415232},
{"movieId1":1,"movieId2":17,"cosineSimilarity":0.5193379705240314},{"movieId1":17,"movieId2":1,"cosineSimilarity":0.5193379705240314},{"movieId1":1,"movieId2":18,"cosineSimilarity":0.2878856673971369},
{"movieId1":18,"movieId2":1,"cosineSimilarity":0.2878856673971369},{"movieId1":1,"movieId2":19,"cosineSimilarity":0.5748199413491161},{"movieId1":19,"movieId2":1,"cosineSimilarity":0.5748199413491161},
{"movieId1":1,"movieId2":20,"cosineSimilarity":0.23990655450727497},{"movieId1":20,"movieId2":1,"cosineSimilarity":0.23990655450727497},{"movieId1":1,"movieId2":21,"cosineSimilarity":0.6001941710899823},
{"movieId1":21,"movieId2":1,"cosineSimilarity":0.6001941710899823},{"movieId1":1,"movieId2":22,"cosineSimilarity":0.3753194399572448},{"movieId1":22,"movieId2":1,"cosineSimilarity":0.3753194399572448},
{"movieId1":1,"movieId2":23,"cosineSimilarity":0.2517163875290593},{"movieId1":23,"movieId2":1,"cosineSimilarity":0.2517163875290593},{"movieId1":1,"movieId2":24,"cosineSimilarity":0.347237386361312},
{"movieId1":24,"movieId2":1,"cosineSimilarity":0.347237386361312},{"movieId1":1,"movieId2":25,"cosineSimilarity":0.5558083317393625},{"movieId1":25,"movieId2":1,"cosineSimilarity":0.5558083317393625},
{"movieId1":1,"movieId2":26,"cosineSimilarity":0.24221735492542298},{"movieId1":26,"movieId2":1,"cosineSimilarity":0.24221735492542298},{"movieId1":1,"movieId2":27,"cosineSimilarity":0.202304358926938 6},
{"movieId1":27,"movieId2":1,"cosineSimilarity":0.2023043589269386},{"movieId1":1,"movieId2":28,"cosineSimilarity":0.22114601889703736},{"movieId1":28,"movieId2":1,"cosineSimilarity":0.22114601889703736},
{"movieId1":1,"movieId2":29,"cosineSimilarity":0.39142815695809924},{"movieId1":29,"movieId2":1,"cosineSimilarity":0.39142815695809924},{"movieId1":1,"movieId2":30,"cosineSimilarity":0.10398545845782872},
{"movieId1":30,"movieId2":1,"cosineSimilarity":0.10398545845782872},{"movieId1":1,"movieId2":31,"cosineSimilarity":0.3863341164713843},{"movieId1":31,"movieId2":1,"cosineSimilarity":0.3863341164713843},
{"movieId1":1,"movieId2":32,"cosineSimilarity":0.880257295017953},{"movieId1":32,"movieId2":1,"cosineSimilarity":0.880257295017953},{"movieId1":1,"movieId2":34,"cosineSimilarity":0.7244193041315444},
```

3.2 Cosine Similarity Flowchart


Start
Fetch movies and movie ratings
Length between 2 vector same?
No
Pad with zeroes
No
Calculate Cosine Similarity
Insert result into database
End

3.3 Pearson Correlation

The Pearson correlation is a measurement that determines the linear relationship between two variables on a scale ranging from -1 to 1. A correlation of -1 indicates a perfect negative relationship, a correlation of 1 represents a perfect positive relationship, and a correlation of 0 represents no relationship. The correlation coefficient is calculated by dividing the covariance of the two variables by the product of their standard deviations.

This method can be applied in a movie recommendation system to evaluate the similarity of users' movie preferences. A high Pearson correlation value implies that the preferences of two users are alike.

To construct a movie similarity matrix based on user ratings, user ratings data for a collection of movies are collected. Then, the data is transformed into a ratings matrix where each row represents a user and each column represents a movie,

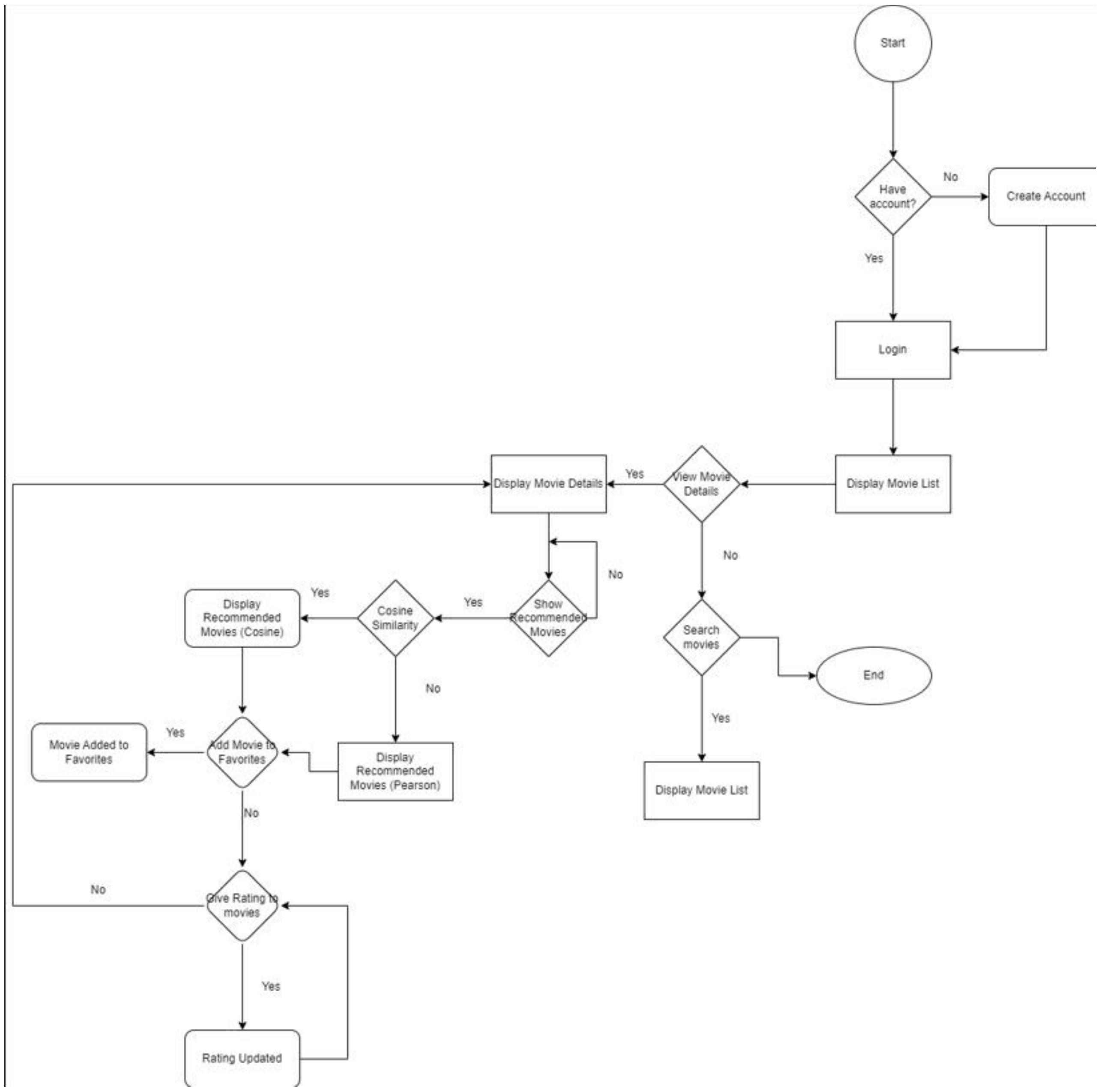
Pearson correlation is created between two movies by comparing the ratings of the same users, and finally, a similarity matrix that holds the similarity scores between two movies in each cell is constricted. This similarity matrix is used to recommend movies to a user by identifying films that are similar to those highly rated by the user.

The formula for Pearson Correlation is attached below.

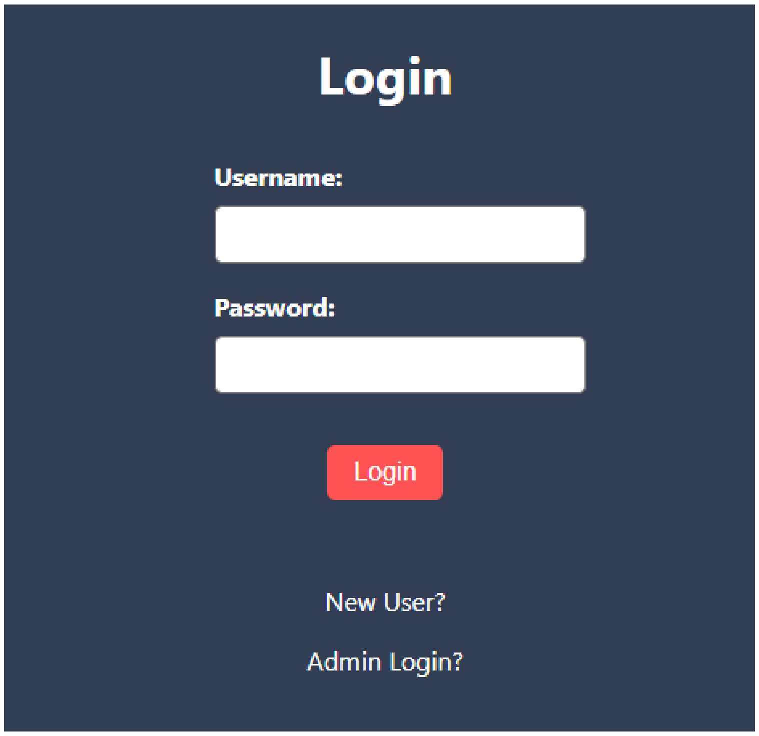
$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}}$$

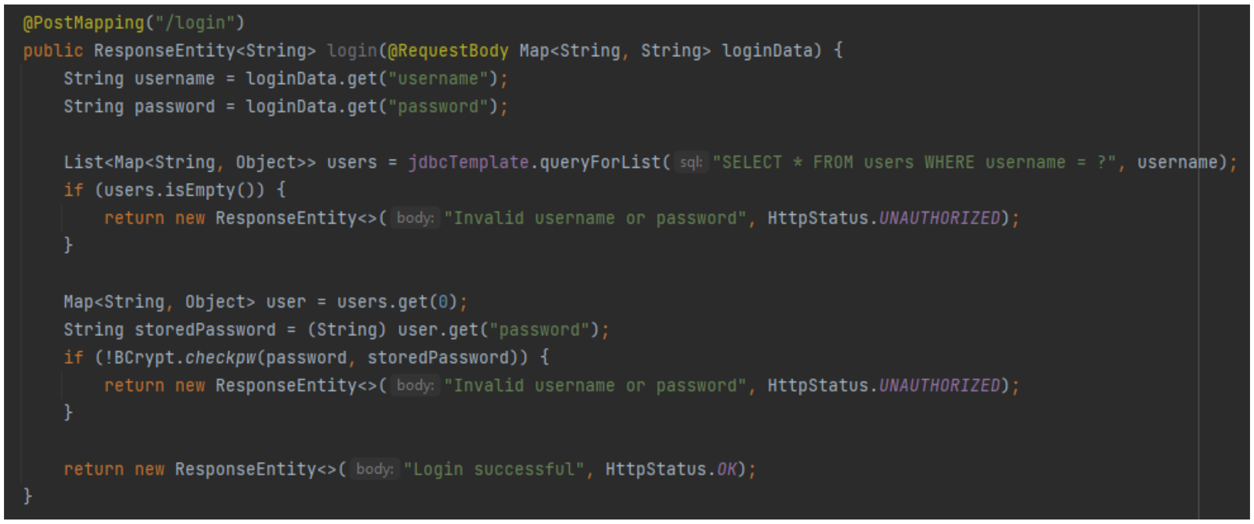
4. System Features

4.1 User Flow Chart



4.2 User Login

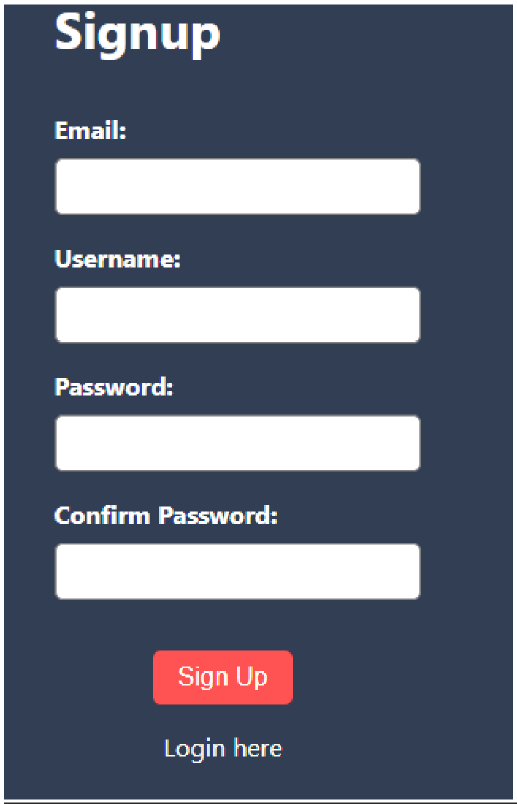


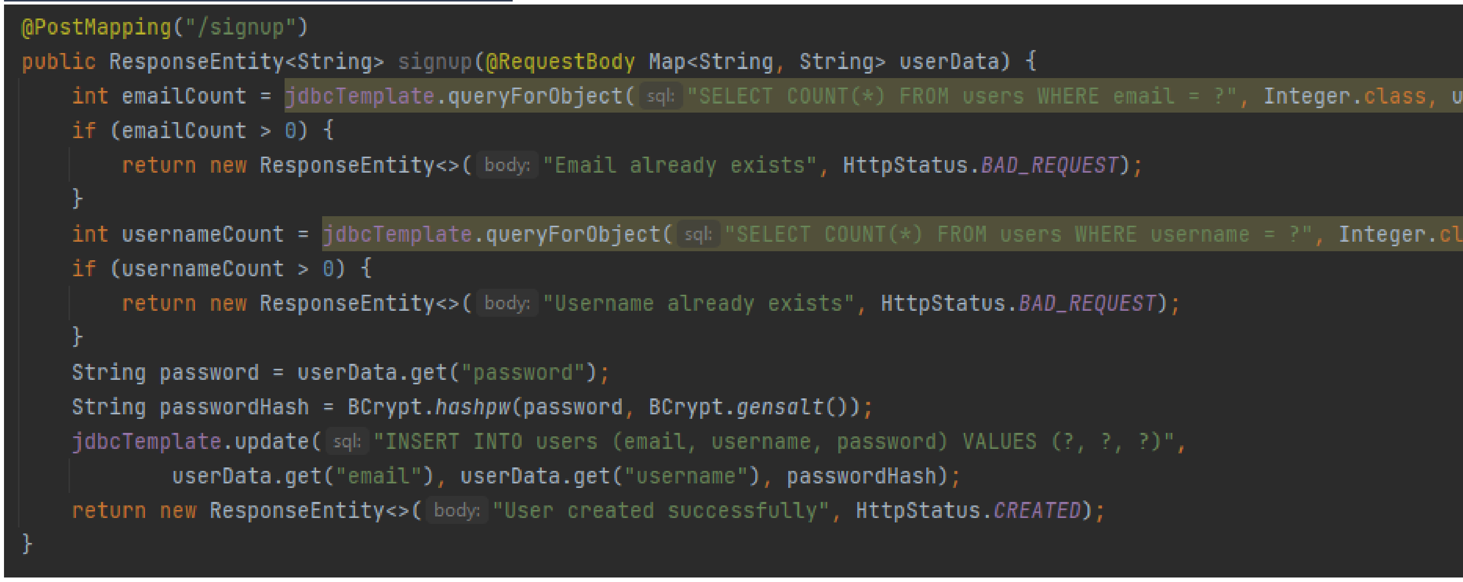


```java
@PostMapping("/login")
public ResponseEntity<String> login(@RequestBody Map<String, String> loginData) {
    String username = loginData.get("username");
    String password = loginData.get("password");

    List<Map<String, Object>> users = jdbcTemplate.queryForList( sql: "SELECT * FROM users WHERE username = ?", username);
    if (users.isEmpty()) {
        return new ResponseEntity<>( body: "Invalid username or password", HttpStatus.UNAUTHORIZED);
    }

    Map<String, Object> user = users.get(0);
    String storedPassword = (String) user.get("password");
    if (!BCrypt.checkpw(password, storedPassword)) {
        return new ResponseEntity<>( body: "Invalid username or password", HttpStatus.UNAUTHORIZED);
    }

    return new ResponseEntity<>( body: "Login successful", HttpStatus.OK);
}
```

This is the user login where user login their account. User can create new account by clicking new user.

4.3 User Register

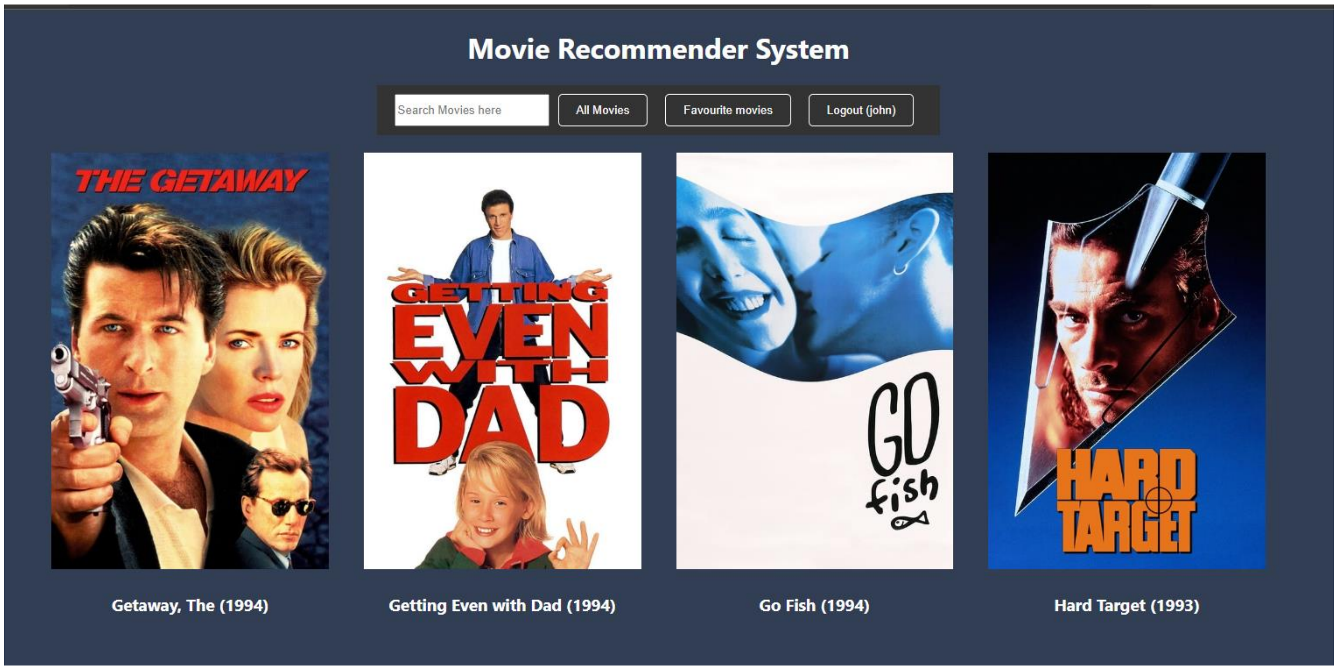


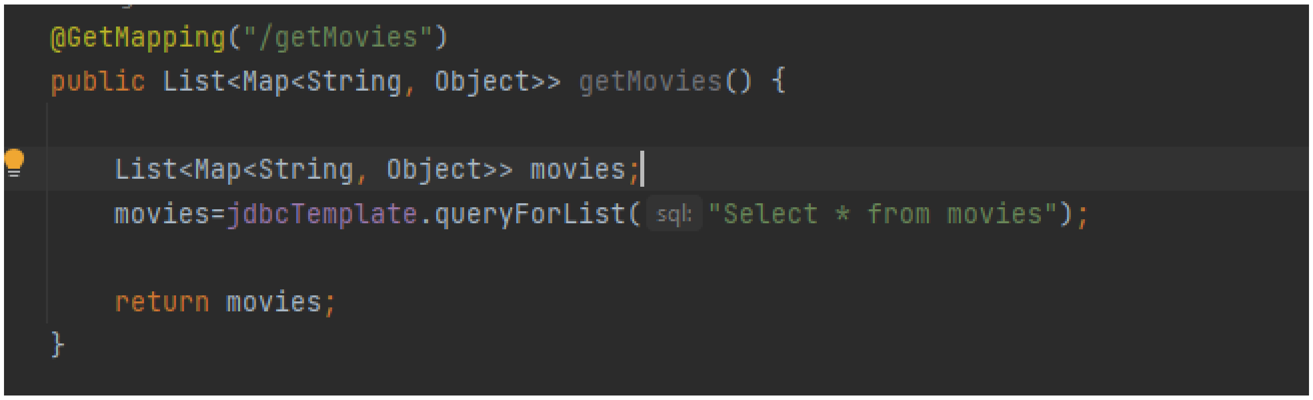


```java
@PostMapping("/signup")
public ResponseEntity<String> signup(@RequestBody Map<String, String> userData) {
    int emailCount = jdbcTemplate.queryForObject( sql: "SELECT COUNT(*) FROM users WHERE email = ?", Integer.class, u
    if (emailCount > 0) {
        return new ResponseEntity<>( body: "Email already exists", HttpStatus.BAD_REQUEST);
    }
    int usernameCount = jdbcTemplate.queryForObject( sql: "SELECT COUNT(*) FROM users WHERE username = ?", Integer.cl
    if (usernameCount > 0) {
        return new ResponseEntity<>( body: "Username already exists", HttpStatus.BAD_REQUEST);
    }
    String password = userData.get("password");
    String passwordHash = BCrypt.hashpw(password, BCrypt.gensalt());
    jdbcTemplate.update( sql: "INSERT INTO users (email, username, password) VALUES (?, ?, ?)",
            userData.get("email"), userData.get("username"), passwordHash);
    return new ResponseEntity<>( body: "User created successfully", HttpStatus.CREATED);
}
```

User can register new account here. It checks for password confirmation and existing username or email before successful registration.

4.4 Movie List

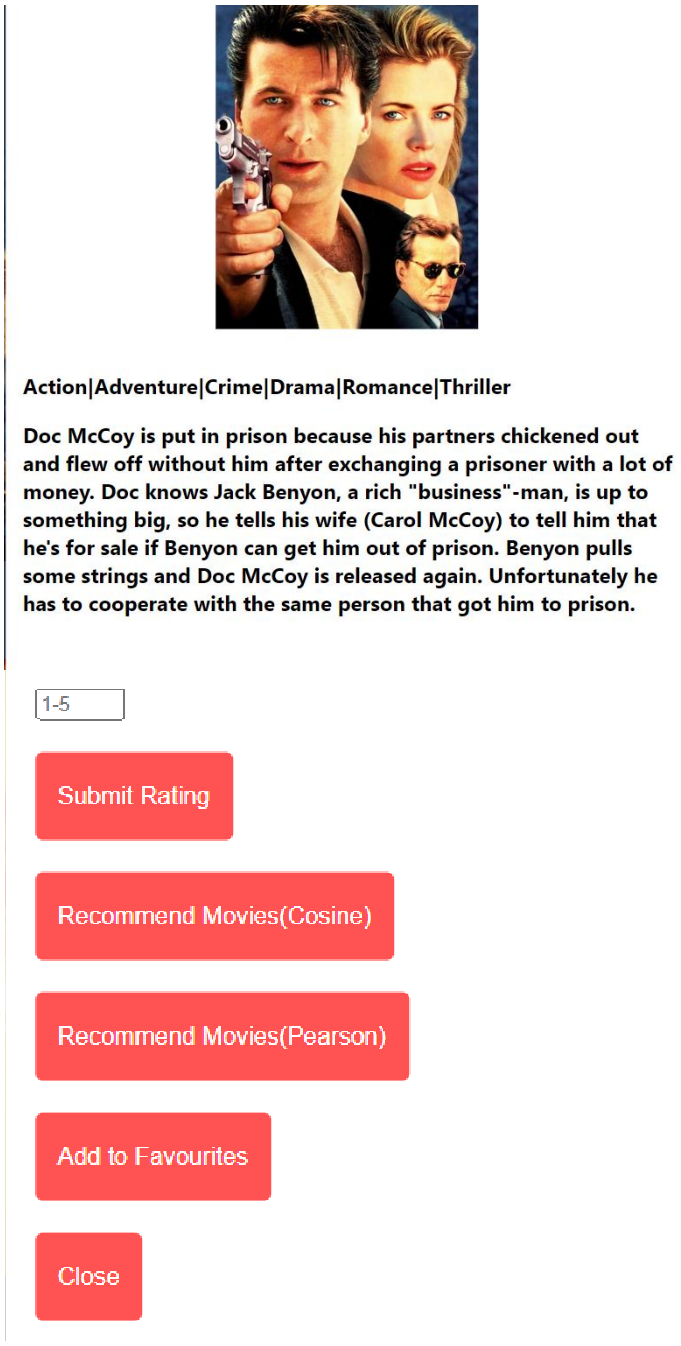




```
@GetMapping("/getMovies")
public List<Map<String, Object>> getMovies() {

    List<Map<String, Object>> movies;
    movies=jdbcTemplate.queryForList( sql: "Select * from movies");

    return movies;
}
```

After login, it shows the user the movie list page. It has navigation bar that show all movies, favourite movies and logout.

4.5 Movie Details

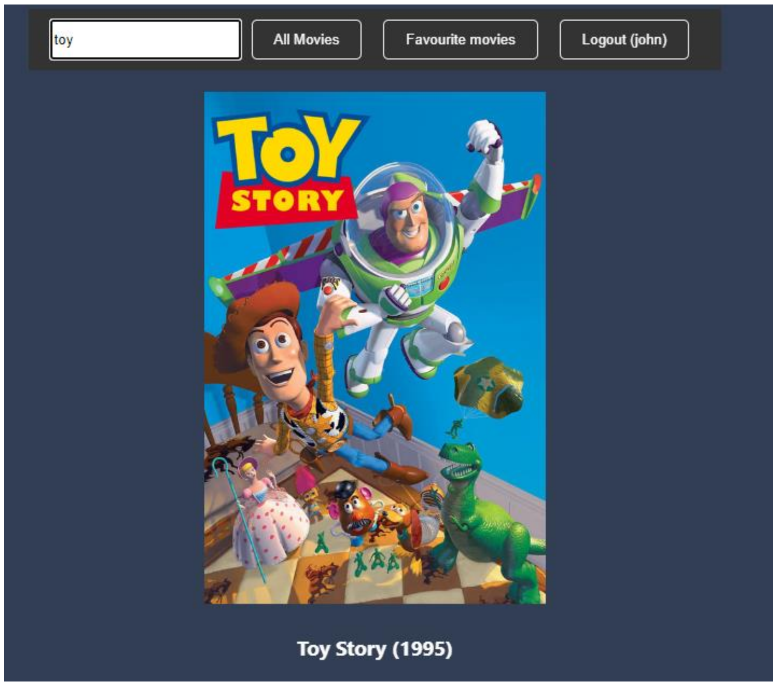


Action|Adventure|Crime|Drama|Romance|Thriller

Doc McCoy is put in prison because his partners chickened out and flew off without him after exchanging a prisoner with a lot of money. Doc knows Jack Benyon, a rich "business"-man, is up to something big, so he tells his wife (Carol McCoy) to tell him that he's for sale if Benyon can get him out of prison. Benyon pulls some strings and Doc McCoy is released again. Unfortunately he has to cooperate with the same person that got him to prison.

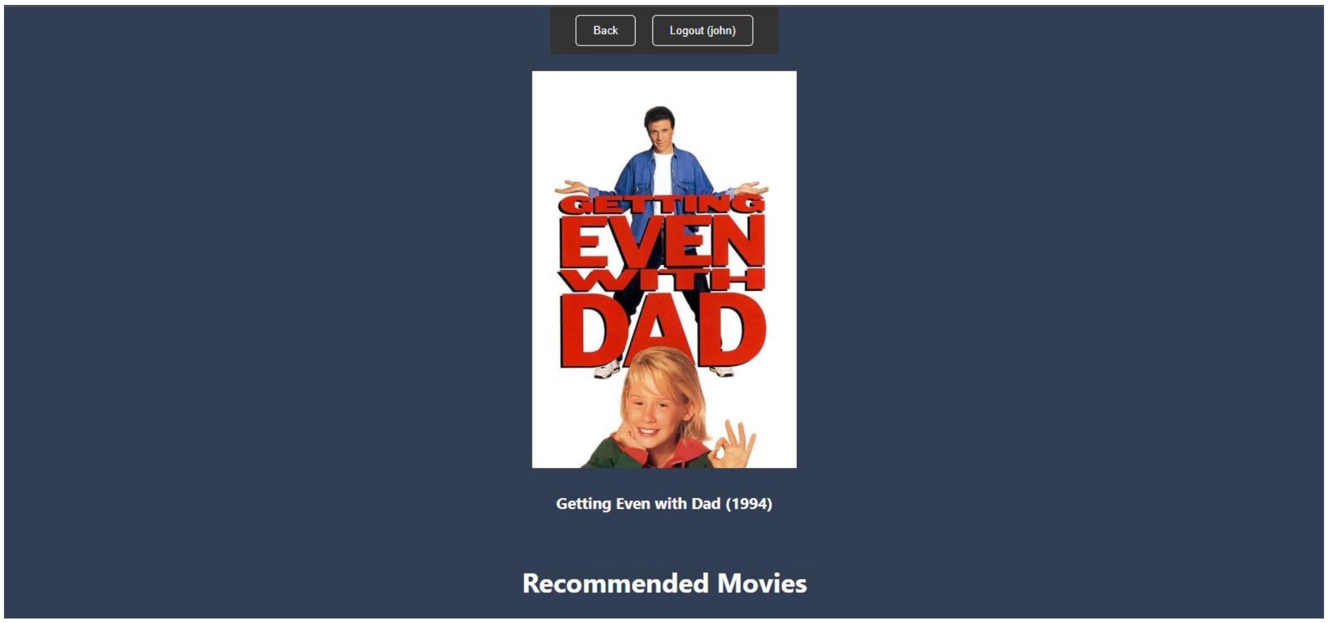
Here it shows the movie details when clicked on a specific movie. We have options to submit rating, recommend movies via cosine similarity, recommend movies via Pearson correlation, add to favourites and close.

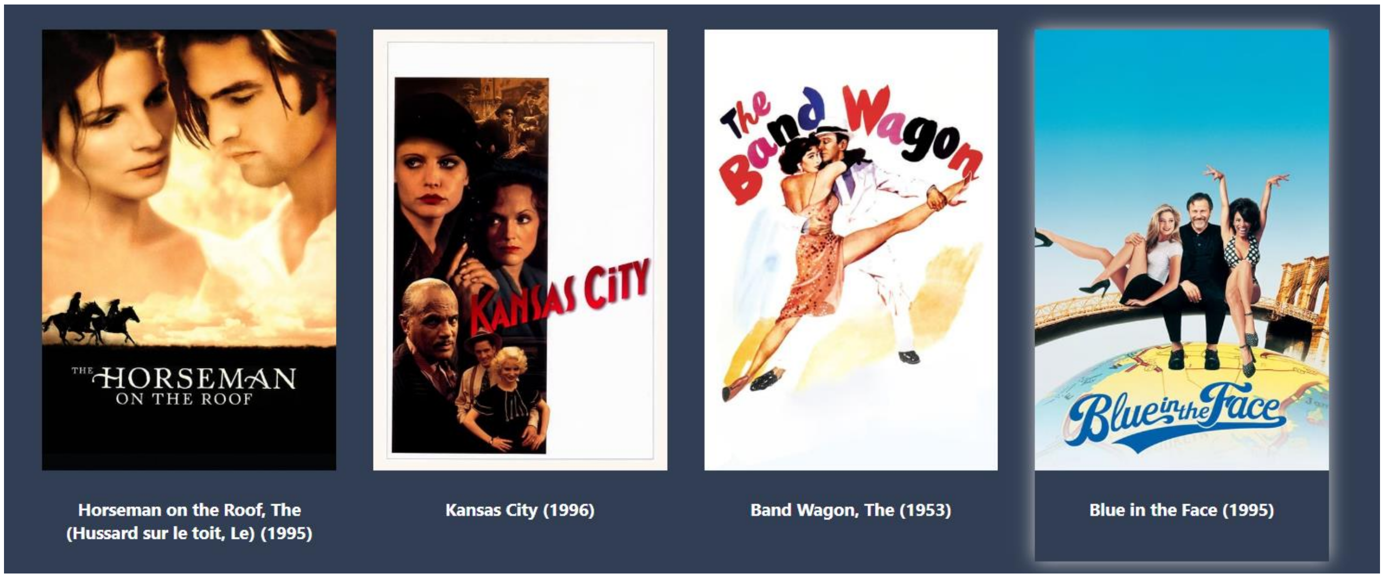
4.6 Search Movie



User can search movie and the result will be shown based on the query.

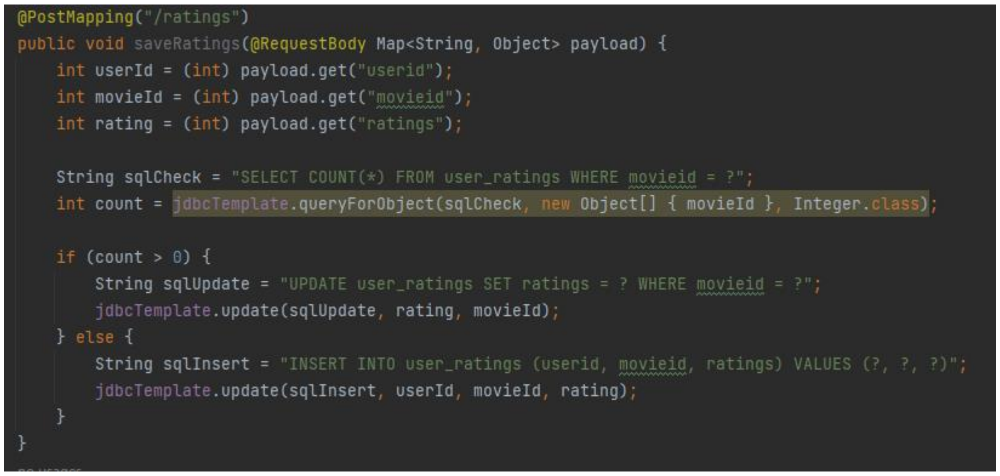
4.7 Recommended Movies





As explained in the previous section about similarity matrix, it will show the recommended movies based on the algorithm chosen.

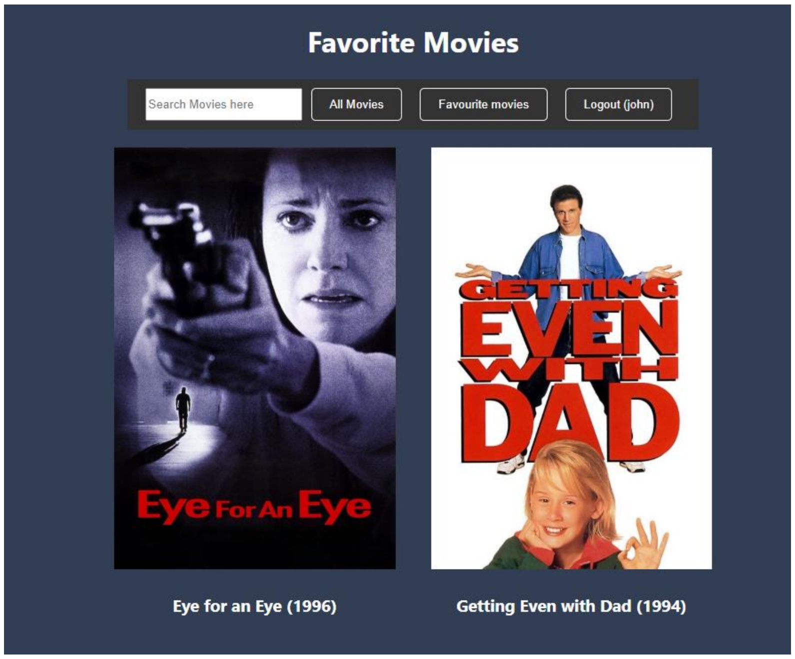
4.8 Submit Ratings

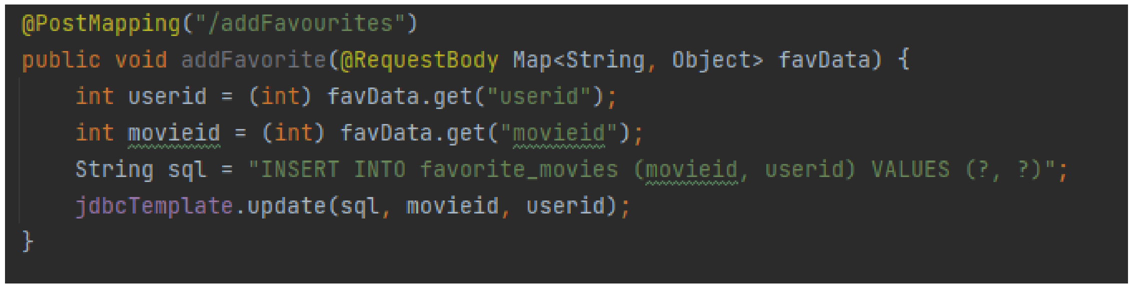


```java
@PostMapping("/ratings")
public void saveRatings(@RequestBody Map<String, Object> payload) {
    int userId = (int) payload.get("userid");
    int movieId = (int) payload.get("movieid");
    int rating = (int) payload.get("ratings");

    String sqlCheck = "SELECT COUNT(*) FROM user_ratings WHERE movieid = ?";
    int count = jdbcTemplate.queryForObject(sqlCheck, new Object[] { movieId }, Integer.class);

    if (count > 0) {
        String sqlUpdate = "UPDATE user_ratings SET ratings = ? WHERE movieid = ?";
        jdbcTemplate.update(sqlUpdate, rating, movieId);
    } else {
        String sqlInsert = "INSERT INTO user_ratings (userid, movieid, ratings) VALUES (?, ?, ?)";
        jdbcTemplate.update(sqlInsert, userId, movieId, rating);
    }
}
```

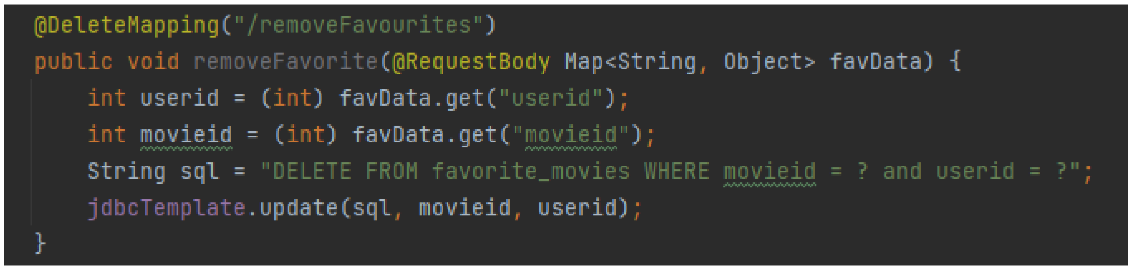
This will submit the ratings of the movie based on the userid, This will check whether the user has already given a rating, and if the user did, it will update the rating.

4.9 Add/Remove Favourite





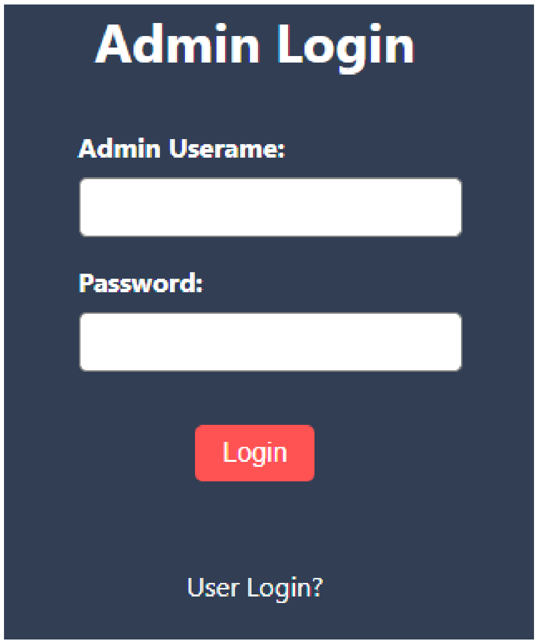
```
@PostMapping("/addFavourites")
public void addFavorite(@RequestBody Map<String, Object> favData) {
    int userid = (int) favData.get("userid");
    int movieid = (int) favData.get("movieid");
    String sql = "INSERT INTO favorite_movies (movieid, userid) VALUES (?, ?)";
    jdbcTemplate.update(sql, movieid, userid);
}
```

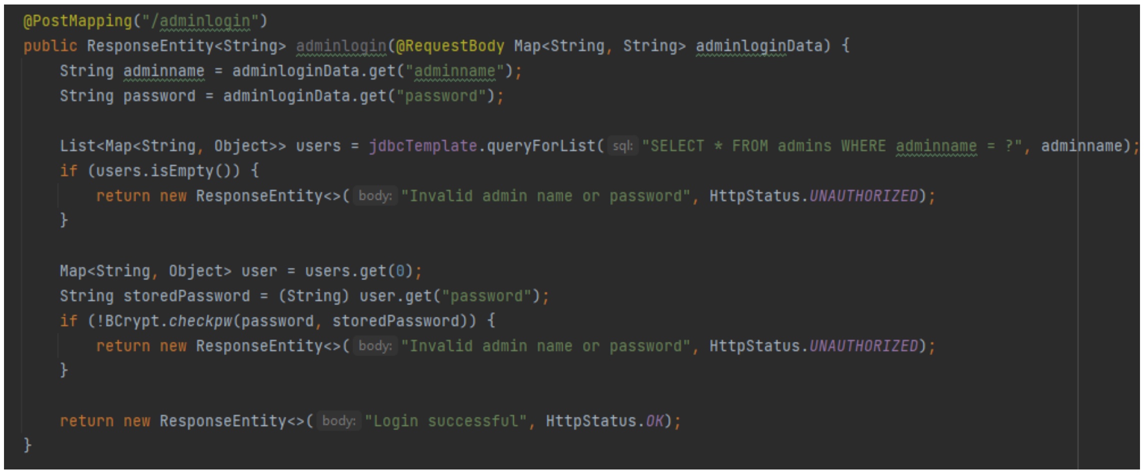


```
@DeleteMapping("/removeFavourites")
public void removeFavorite(@RequestBody Map<String, Object> favData) {
    int userid = (int) favData.get("userid");
    int movieid = (int) favData.get("movieid");
    String sql = "DELETE FROM favorite_movies WHERE movieid = ? and userid = ?";
    jdbcTemplate.update(sql, movieid, userid);
}
```

This will add or remove favourite movies. User can see favourite movies in the favourite movie menu.

4.10 Admin Login

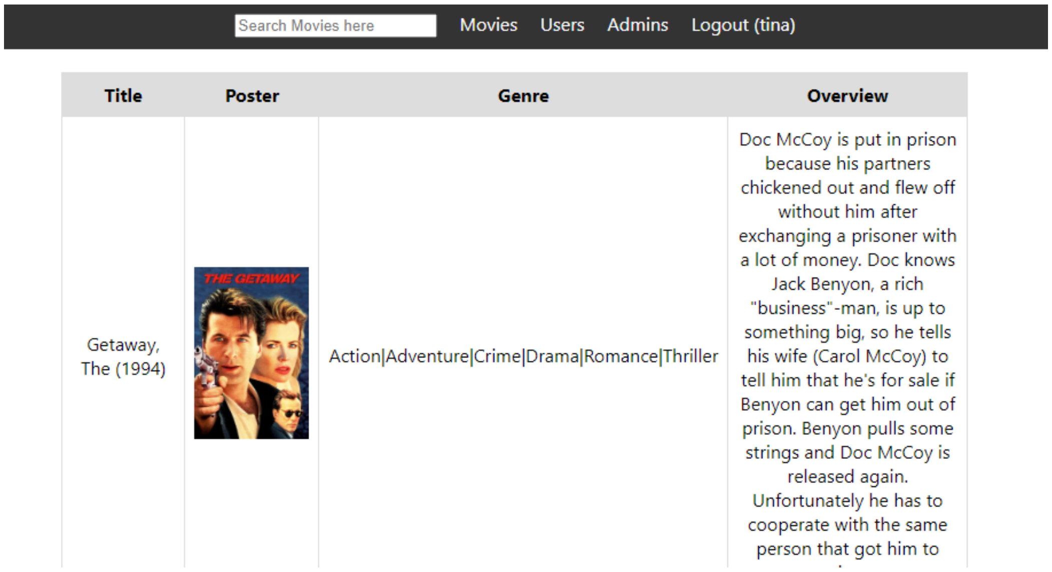




```
@PostMapping("/adminlogin")
public ResponseEntity<String> adminlogin(@RequestBody Map<String, String> adminloginData) {
    String adminname = adminloginData.get("adminname");
    String password = adminloginData.get("password");

    List<Map<String, Object>> users = jdbcTemplate.queryForList( sql: "SELECT * FROM admins WHERE adminname = ?", adminname);
    if (users.isEmpty()) {
        return new ResponseEntity<>( body: "Invalid admin name or password", HttpStatus.UNAUTHORIZED);
    }

    Map<String, Object> user = users.get(0);
    String storedPassword = (String) user.get("password");
    if (!BCrypt.checkpw(password, storedPassword)) {
        return new ResponseEntity<>( body: "Invalid admin name or password", HttpStatus.UNAUTHORIZED);
    }

    return new ResponseEntity<>( body: "Login successful", HttpStatus.OK);
}
```

Admin can login here.

4.11 Admin Dashboard

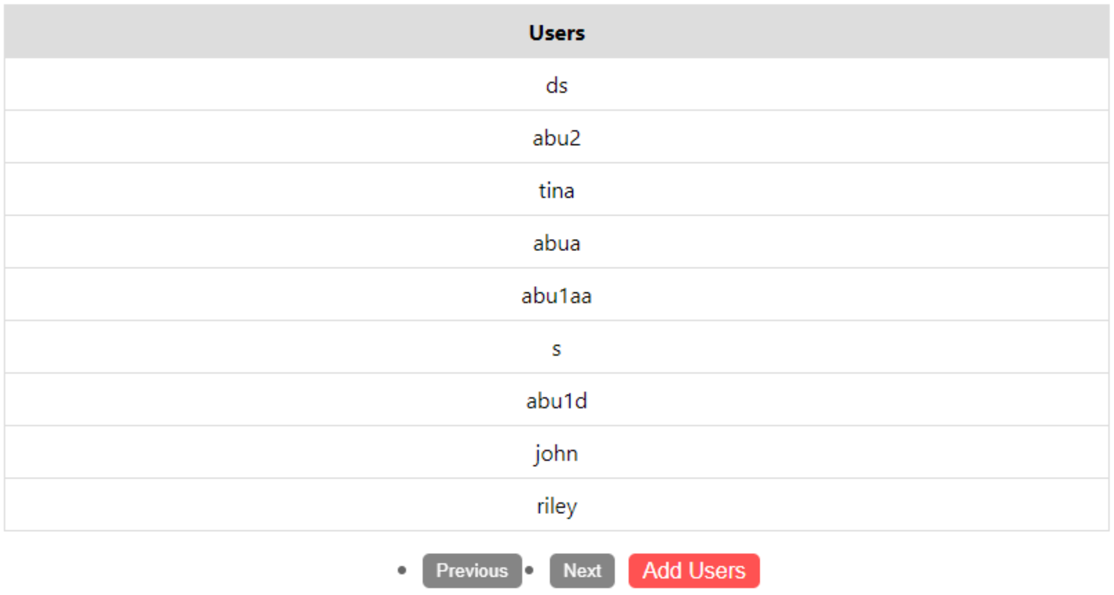


Search Movies here | Movies | Users | Admins | Logout (tina)

Title	Poster	Genre	Overview
Getaway, The (1994)		Action|Adventure|Crime|Drama|Romance|Thriller	Doc McCoy is put in prison because his partners chickened out and flew off without him after exchanging a prisoner with a lot of money. Doc knows Jack Benyon, a rich "business"-man, is up to something big, so he tells his wife (Carol McCoy) to tell him that he's for sale if Benyon can get him out of prison. Benyon pulls some strings and Doc McCoy is released again. Unfortunately he has to cooperate with the same person that got him to

This is the admin dashboard where the admin can see the list of movies and search it.

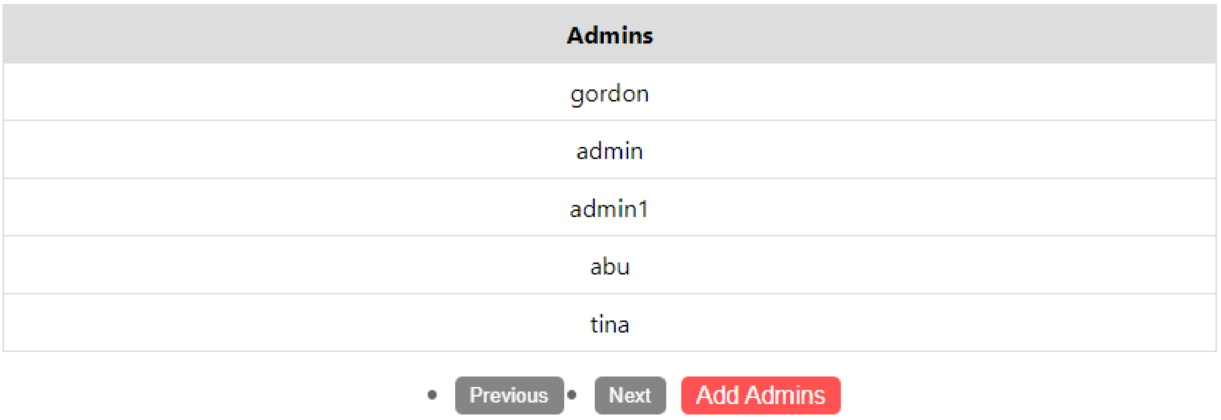
4.12 View/Add User



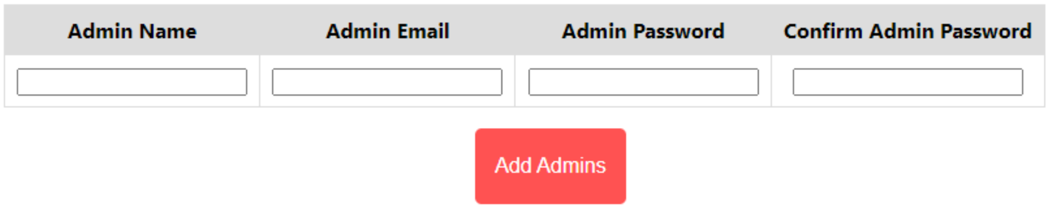
Users
ds
abu2
tina
abua
abu1aa
s
abu1d
john
riley

Admin can view and add users. Adding users has the same code as creating user account.

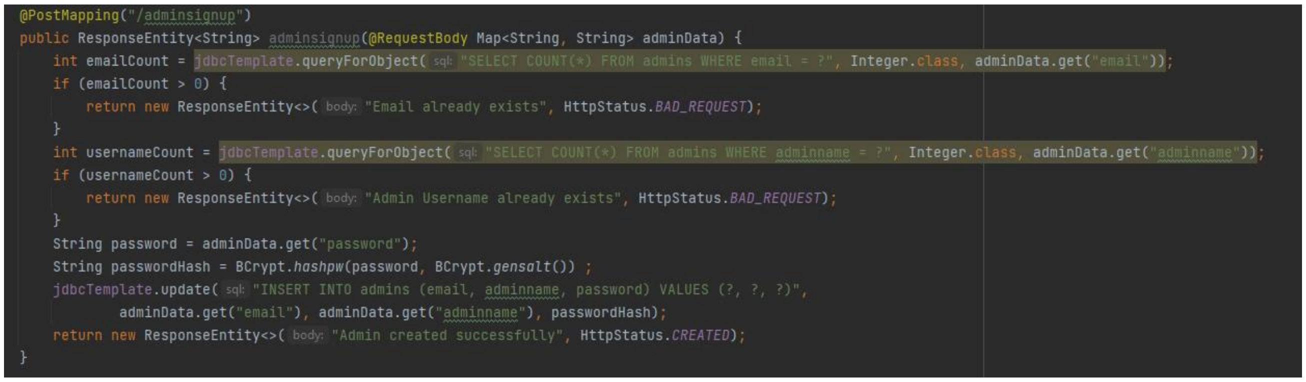
4.13 View/Add Admin



Admins
gordon
admin
admin1
abu
tina



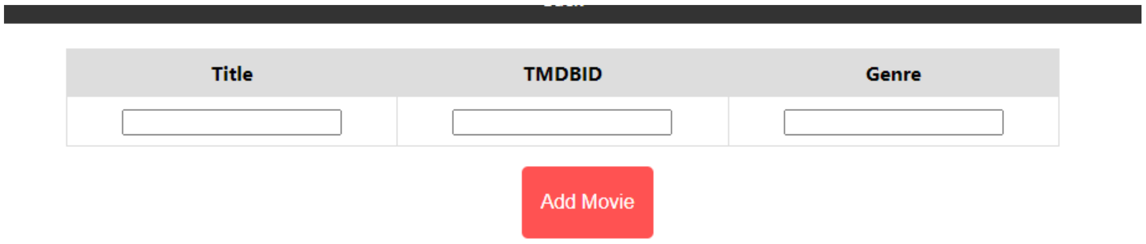
Admin Name	Admin Email	Admin Password	Confirm Admin Password



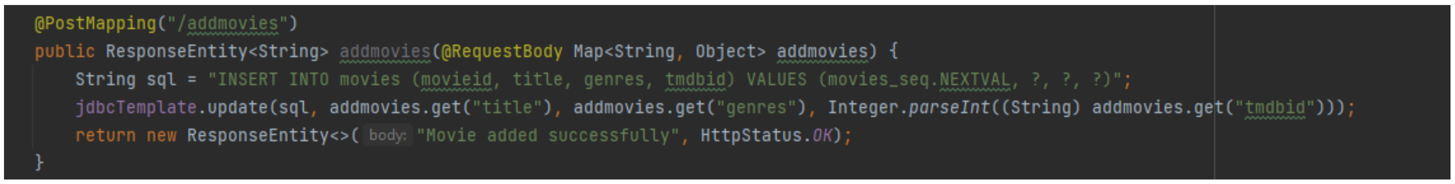
```
@PostMapping("/adminsignup")
public ResponseEntity<String> adminsignup(@RequestBody Map<String, String> adminData) {
    int emailCount = jdbcTemplate.queryForObject( sql: "SELECT COUNT(*) FROM admins WHERE email = ?", Integer.class, adminData.get("email"));
    if (emailCount > 0) {
        return new ResponseEntity<>( body: "Email already exists", HttpStatus.BAD_REQUEST);
    }
    int usernameCount = jdbcTemplate.queryForObject( sql: "SELECT COUNT(*) FROM admins WHERE adminname = ?", Integer.class, adminData.get("adminname"));
    if (usernameCount > 0) {
        return new ResponseEntity<>( body: "Admin Username already exists", HttpStatus.BAD_REQUEST);
    }
    String password = adminData.get("password");
    String passwordHash = BCrypt.hashpw(password, BCrypt.gensalt()) ;
    jdbcTemplate.update( sql: "INSERT INTO admins (email, adminname, password) VALUES (?, ?, ?)",
            adminData.get("email"), adminData.get("adminname"), passwordHash);
    return new ResponseEntity<>( body: "Admin created successfully", HttpStatus.CREATED);
}
```

Admin can view and add new admins.

4.14 Add Movies



Title	TMDBID	Genre



```
@PostMapping("/addmovies")
public ResponseEntity<String> addmovies(@RequestBody Map<String, Object> addmovies) {
    String sql = "INSERT INTO movies (movieid, title, genres, tmdbid) VALUES (movies_seq.NEXTVAL, ?, ?, ?)";
    jdbcTemplate.update(sql, addmovies.get("title"), addmovies.get("genres"), Integer.parseInt((String) addmovies.get("tmdbid")));
    return new ResponseEntity<>( body: "Movie added successfully", HttpStatus.OK);
}
```

Admin can add movies as well. TMDBID is used to get poster and description the TMDB.