

Software Requirements Specification

For

Movie Recommender System

Capstone Project

MUHAMMAD AFIQ AFWAN BIN MOHD ZUKI

8 FEBRUARY 2023

TABLE OF CONTENTS

1.0	Introduction	3
	1.1 Document Purpose.....	3
	1.2 Product Scope.....	3
	1.3 Intended Audience and Document Overview.....	3
2.0	Overall Description	4
	2.1 Product Perspective	4
	2.2 Features List.....	4
3.0	System Features.....	6
	3.1 Use Case Diagram	6
	3.2 System Architecture	7
	3.3 Flowchart.....	8
	3.4 Entity Relationship Diagram (ERD)	9
	3.5 Login	10
	3.6 Register	12
	3.7 User Homepage.....	15
	3.8 Movie Details & Recommended Movies	18
	3.9 Add Rating	21
	3.10 Add Movie to Favorite.....	23
	3.11 Remove Movie from Favorite	25
	3.12 View Favorite Movies List	26
	3.13 Search Movie	28
	3.14 Admin Dashboard.....	30
	3.15 Add User	31
	3.16 View List of Users.....	33
	3.17 Add Movie.....	34
	3.18 View List of Movies.....	36
4.0	Pearson Correlation	38

1.0 Introduction

1.1 Document Purpose

The purpose of this System Requirements Specification (SRS) is to outline the requirements for Movie Recommender System that uses Pearson Correlation as the algorithm for recommendation. In addition to said requirements, the document also provides the specifications on the scope, features, and design. Hence, this document should act as a guideline for efficient and well-managed project completion and serve as a reference in the future.

1.2 Product Scope

The objective of this project is to create and implement a movie recommender system. The system will use Pearson Correlation to compare the movies rated by users and recommend movies that are most similar to the movie that are chosen by the user. The website will be used primarily by the user and admin. Users are required to create an individual account to use the system. Besides, users should be able to view list of movies, view movie details, give rating, add movies to favorite, view list of favorite movies, and view recommended movies. In addition, the admin of the system is able to login to the system to add new users, add new movies, view list of users, and view list of movies. The ultimate goal of the system is to provide a platform for users to access databases of movies and get movie recommendations.

1.3 Intended Audience and Document Overview

The intended audience for this program is for individuals who are interested in movies, including movie fans, and anyone who is interested in discovering new movies. The intended use of this program is to provide users with a comprehensive and seamless experience when viewing and searching for movies. This program will also recommend movies according to the movie selected by users. The target audiences for this document are developers and clients. The purpose of this document is to describe the SRS for this program. Several modellings diagram have been described in the documentation including use case diagram, entity relationship diagram, and product screenshot.

2.0 Overall Description

2.1 Product Perspective

This product is a web-based application system that provides users with the ability to register their own accounts. They will have access to movie listings, movie details, add rating, the ability to add movies to their favorites, view their favorite movie list, and see recommended films. Admins can log in to add new users, add new movies, view user lists, and view movie lists. The system's purpose is to offer a convenient platform for users to browse movie databases and get movie recommendations.

2.2 Features List

Log in - The log in feature shall provide user and admin with the ability to log in to the system.

Create account - The create account feature shall allow user the ability to register their account to use the system.

View movies - The view movies feature shall provide user with the ability to view movies that is displayed on the dashboard.

View recommended movies - The recommended movies feature shall provide user with the ability to view recommended movies based on the selected movie.

Search movies - The search movies feature shall provide user with the ability to search for movies based on title.

Add rating - The add rating feature shall provide user with the ability to give rating for movie.

Add favorite - The add favorite feature shall provide user with the ability to add movies to their favorite list.

Delete favorite - The delete favorite feature shall provide user with the ability to remove or delete movie from favorite list.

Add user - The add user feature shall provide admin the ability to add new user for both normal user and admin role.

Add movie - The add movie feature shall provide admin with the ability to add movie with title, poster, genre, and description.

View list of users - The view list of users feature shall provide admin with the ability to view users' information details

View list of movies - The view list of movies feature shall provide admin with the ability to view movies information details.

3.0 System Features

3.1 Use Case Diagram

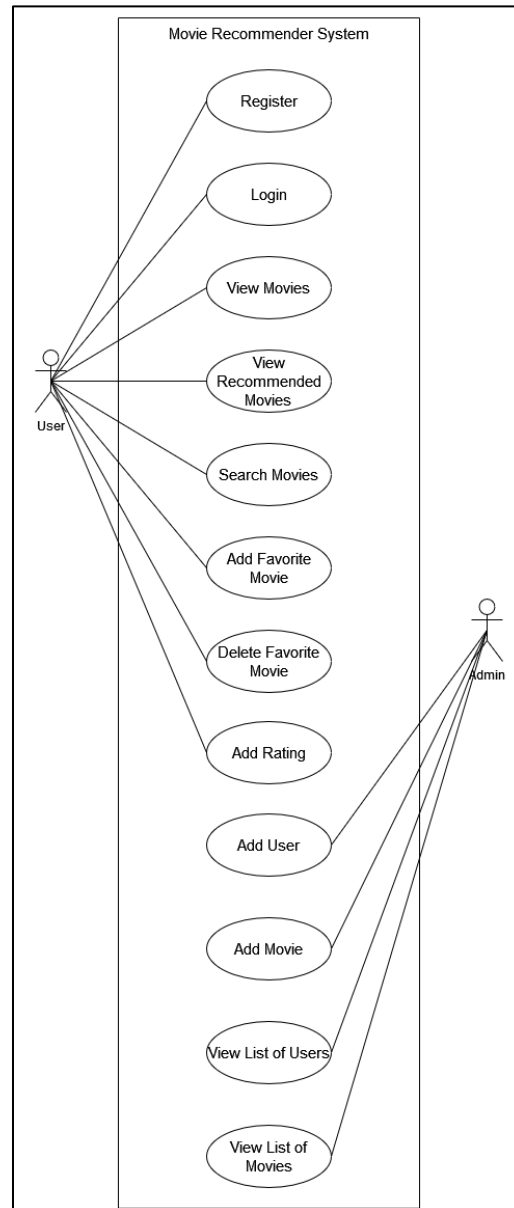


Figure 1: Use Case Diagram

The use case diagram in Figure 1 above consists of 2 main actors of the system which are user and admin. Besides, there are also 12 functionalities for this system which are register account, login, view movies, view recommended movies, search movies, add rating, add favorite movie, delete favorite movie, add user, add movie, view list of movies, and view list of users.

3.2 System Architecture

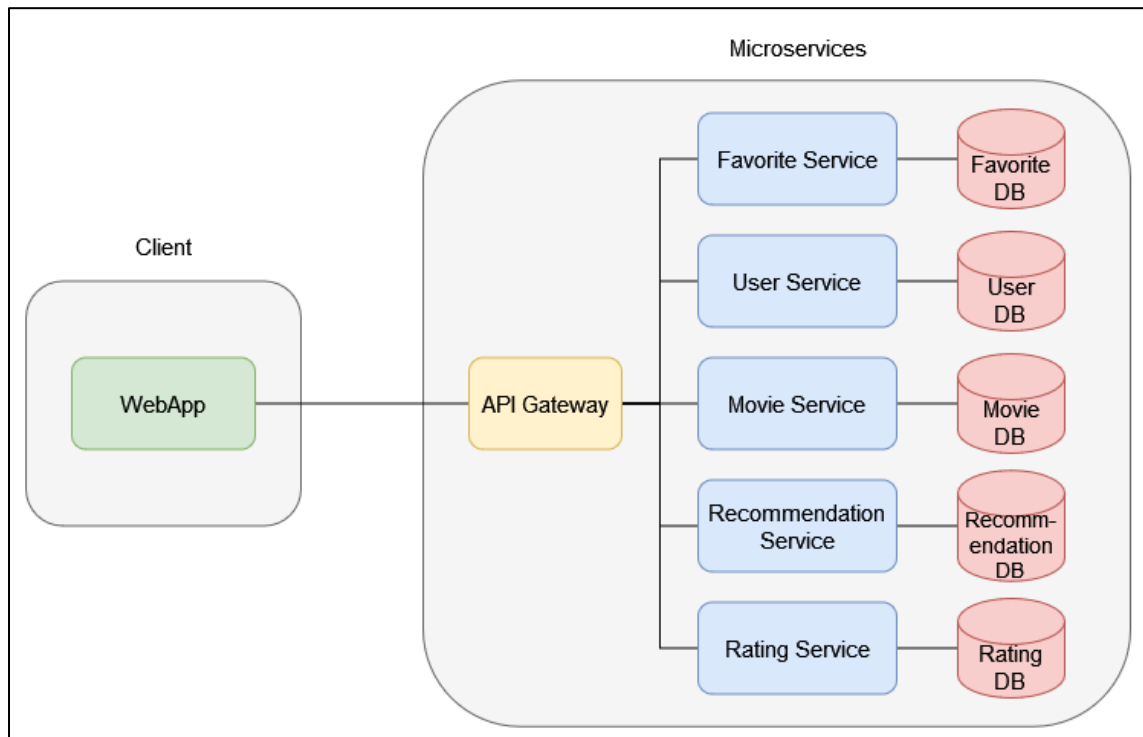


Figure 2: System Architecture

System architecture outlines the techniques and methods that serve as the system's fundamental structure. Microservice is chosen as the web development architecture where every service is running independently. Every service has its own program and is responsible for specific functions. The architecture consists of frontend, backend, and database. The front-end layer is written in JavaScript, HTML, Thymeleaf, and Bootstrap. A user of the application interacts with the web application via a web browser. The API requests are made through the API gateway that establishes a connection between the client and the microservices. Java Spring Boot is used as the framework for developing microservices. In total five microservices are developed for the system, and each microservice has its own database. IBM DB2 is utilized as the database to handle database-related tasks such as storing, querying, and retrieving data.

3.3 Flowchart

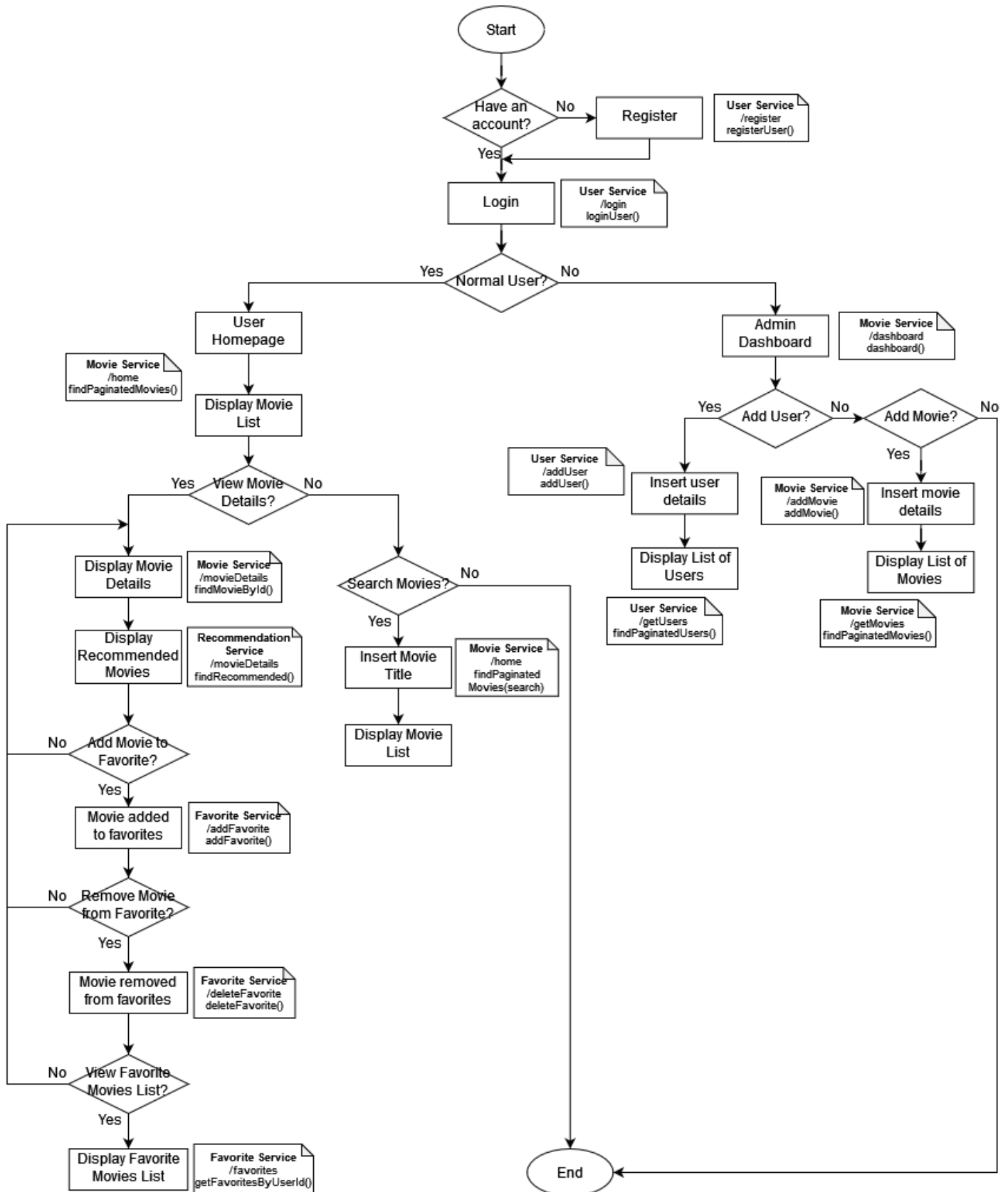


Figure 3: System Flowchart

3.4 Entity Relationship Diagram (ERD)

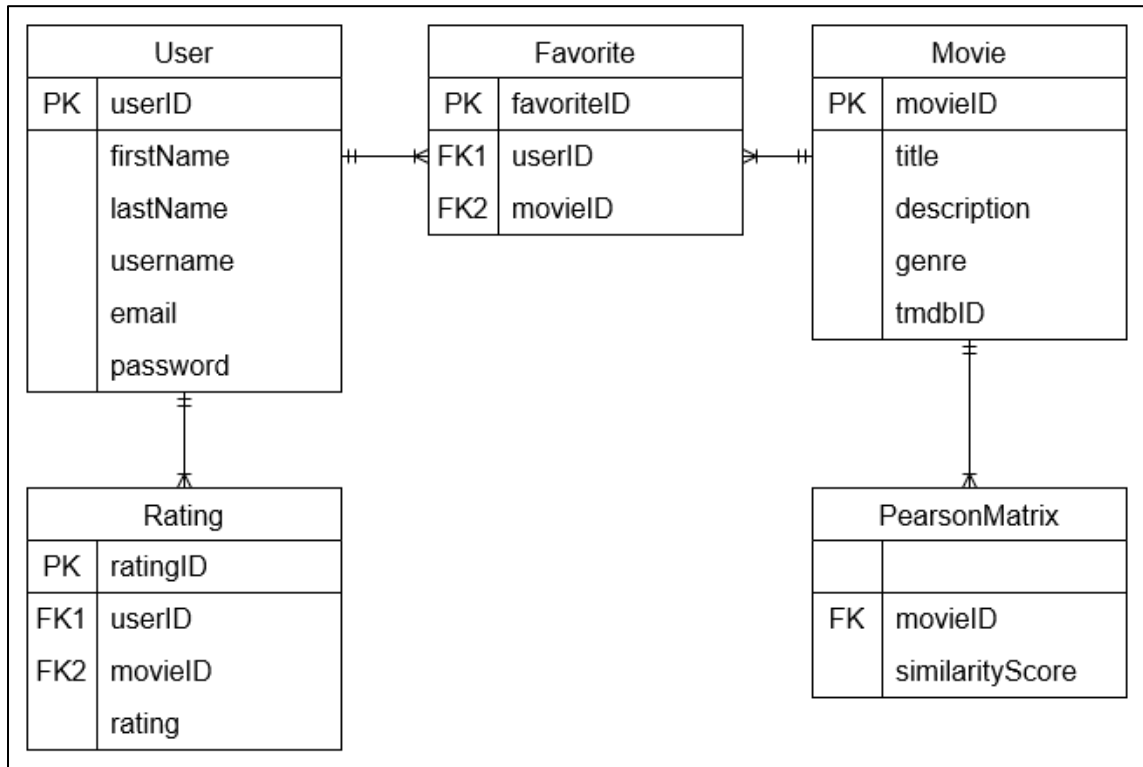


Figure 4: ERD of Movie Recommender System

Figure 4 is the ERD of Movie Recommender System which shows the relationship between all the tables in database.

3.5 Login

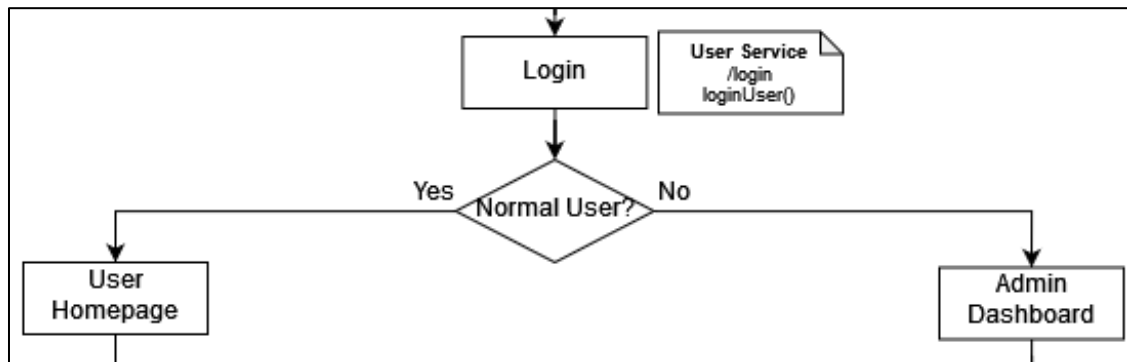


Figure 5: Login Flow

The screenshot shows the 'Index Page (Login Form)' for a 'Movie Recommender System'. The page features a login form on the left and a system description on the right, all set against a background of movie posters.

Login Form:

- Username:
- Password:
- Login button (green)
- Don't have an account? Register button (blue)

Movie Recommender System:

A movie recommendation system is a website that uses algorithm to recommend movies to users based on rated movies.

Figure 6: Index Page (Login Form)

The login page shown in Figure 6 above is where the user and admin can log in to the system. User and admin need to enter valid email and password before proceeding to their own specific dashboard page and perform actions or tasks based on their access privileges such as admin can view and add both movies and new users, meanwhile user can view movies, search movies, add to favorite, and view recommended movies.

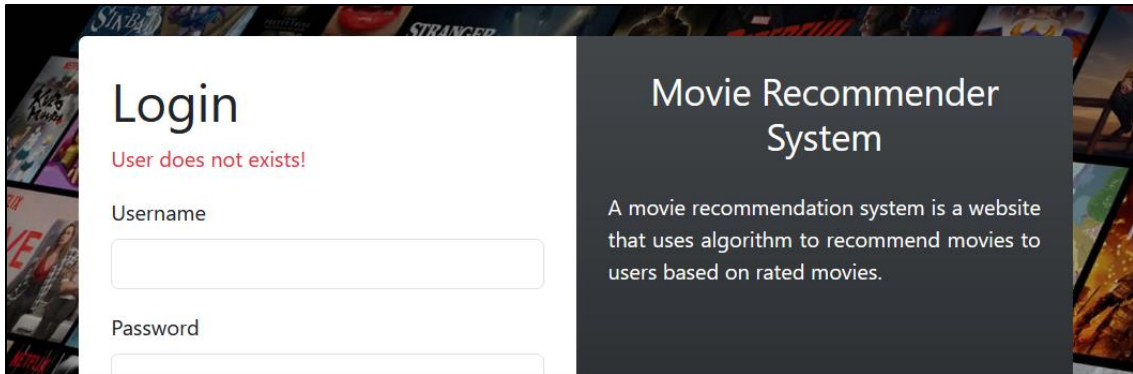


Figure 7: Login Validation

In Figure 7 above, if user enters wrong username/password, a warning message will be displayed on top of the form.

```
// Endpoint for login
no usages
@PostMapping("/login")
public String login(@ModelAttribute("user") User user, Model model, HttpSession session)
{
    try {
        HttpEntity<User> entity = new HttpEntity<User>(user);
        // Call api to post user data for login
        User getUser = restTemplate
            .exchange(url: "http://localhost:8081/user/login", HttpMethod.POST, entity, User.class).getBody();
        // Add session
        session.setAttribute(s: "username", getUser.getUsername());
        session.setAttribute(s: "userId", getUser.getUserId());
        // Check user roles to redirect to specific homepage
        if (getUser.getRole().equals("ADMIN")) {
            return "redirect:/dashboard";
        } else {
            return "redirect:/home";
        }
    } catch (HttpStatusException e) {
        // Get the exception body as string
        String errorpayload = e.getResponseBodyAsString();
        // Add model attribute to display error message on front-end
        model.addAttribute(attributeName: "msg", errorpayload);
        return "index";
    }
}
```

Figure 8: Web Endpoint for Login

```
// Endpoint for login
no usages
@PostMapping("/login")
public ResponseEntity<?> loginUser(@RequestBody User user)
{
    return userService.loginUser(user);
}
```

Figure 9: User Service API for Login

Based on Figure 8, the login endpoint calls the Java API of User Service in Figure 9 to post the user data for login. It will then set the session for successful logged in user and

redirect them based on roles. If there is an error, the API will return the error message accordingly.

3.6 Register

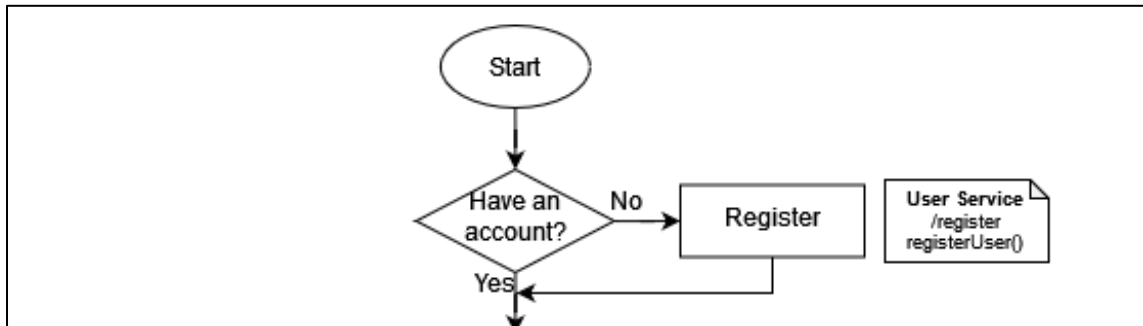


Figure 10: Register Flow

The image shows a web form titled 'Register' on a dark background. The form contains the following fields: 'First Name', 'Last Name', 'Username', 'Email', 'Password', and 'Confirm Password'. Each field is represented by a white input box. At the bottom of the form is a green button labeled 'Register'.

Figure 11: Register Page (Register Form)

As illustrated in Figure 11, users are required to enter their details using the register form to create an account. A successfully registered user will be redirected back to the login page before they can access the system.

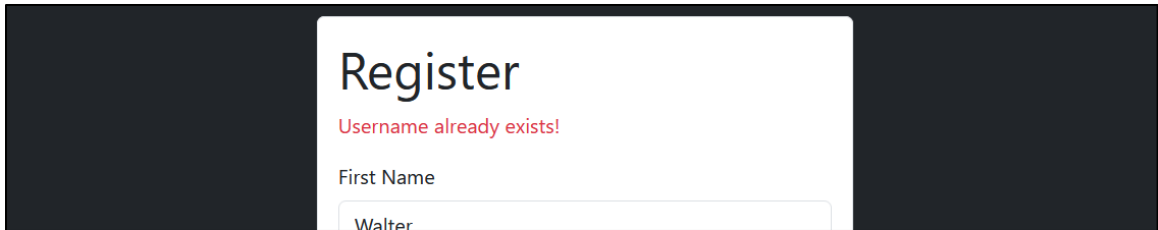


Figure 12: Register Validation

In Figure 12 above, if user enters existing username/existing email/not matching password, a warning message will be displayed on top of the form.

```
// Endpoint for registration
no usages
@PostMapping("/register")
public String register(@ModelAttribute("user") User user, Model model)
{
    try {
        HttpEntity<User> entity = new HttpEntity<User>(user);
        // Call api to post data to register
        restTemplate
            .exchange(url: "http://localhost:8081/user/register", HttpMethod.POST, entity, User.class).getBody();
        return "redirect:/";
    } catch (HttpStatusCodeException e) {
        String errorpayload = e.getResponseBodyAsString();
        // Add model attribute to display error message on front-end
        model.addAttribute(attributeName: "msg", errorpayload);
        return "register";
    }
}
```

Figure 13: Web Endpoint for Register

```
// Endpoint for registration
no usages
@PostMapping("/register")
public ResponseEntity<?> registerUser(@RequestBody User user) { return userService.registerUser(user); }
```

Figure 14: User Service API for Register

Based on Figure 13, the register endpoint calls the register API of User Service in Figure 14 to post the user data for registration. It will then redirect them back to login page once successful. If there is an error, the API will return the error message accordingly.

```

// Register method
1 usage
public ResponseEntity<?> registerUser(User user)
{
    // Check existing username if exist
    if (repo.findByUsername(user.getUsername()) != null) {
        return new ResponseEntity<> (body: "Username already exists!", HttpStatus.BAD_REQUEST);
    }
    // Check existing email if exist
    else if (repo.findByEmail(user.getEmail()) != null) {
        return new ResponseEntity<> (body: "Email already exists!", HttpStatus.BAD_REQUEST);
    } else {
        // Check if password and confirm password match
        if (user.getPassword().equals(user.getConfirmPassword())) {
            // Set user's role
            user.setRole("USER");
            // hash password using bcrypt
            user.setPassword(passEncoder.encode(user.getPassword()));
            // Save transaction
            repo.save(user);
            return new ResponseEntity<User>(user, HttpStatus.OK);
        } else {
            return new ResponseEntity<> (body: "Password does not match!", HttpStatus.BAD_REQUEST);
        }
    }
}
}

```

Figure 15: User service for Register

In Figure 15 above, the registerUser() method in User service will do the checking of username and email if it exists and return alert response. Bcrypt is used to encode the password to store it in database.

3.7 User Homepage

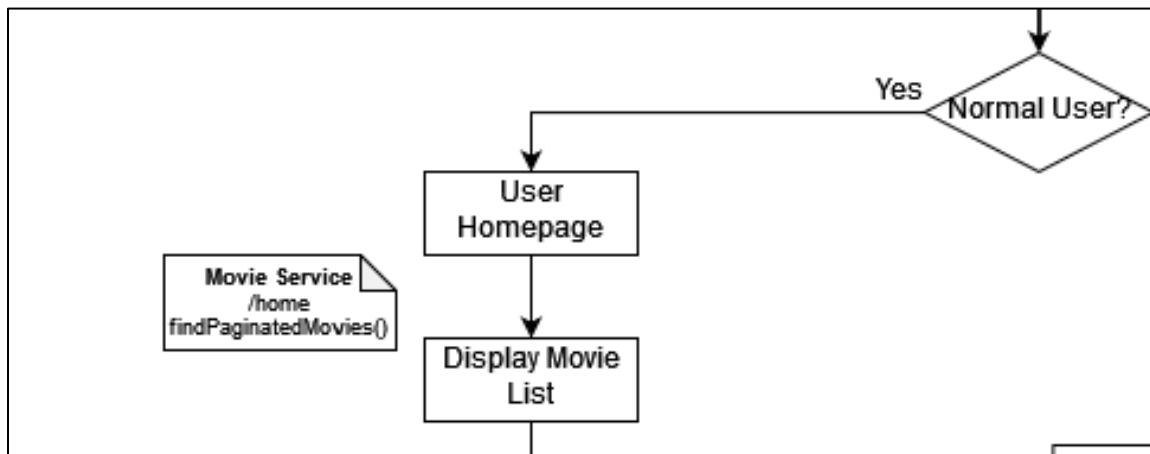


Figure 16: User Homepage and Display Movie List flow

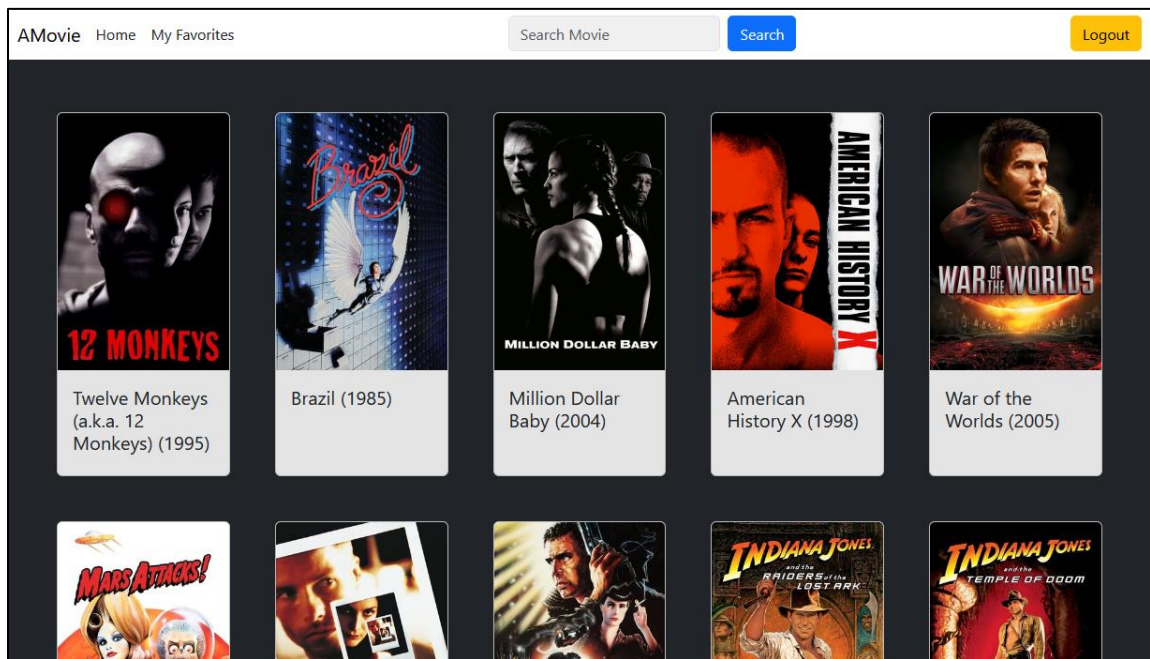


Figure 17: User Homepage with Movies

The user's homepage will display a list of movies as shown in Figure 17 above. User is able to click on each of the movie cards to view more details about the movie.

```

// Endpoint for user's homepage
@GetMapping("/home")
public String home(Model model)
{
    // Call movies pagination method to be displayed on homepage
    return findPaginated(pageNo: 1, search: "", model);
}

// Endpoint for movies pagination
1 usage
@GetMapping("/page/{pageNo}")
public String findPaginated(@PathVariable(value = "pageNo") int pageNo,
                           @RequestParam(value = "search", required=false) String search,
                           Model model) {
    HttpEntity<Integer> entity = new HttpEntity<>(pageNo);
    // Call api to fetch movies
    RestPage<Movie> page = restTemplate
        .exchange(url: "http://localhost:8083/movie/page/" + pageNo + "?search=" + search, HttpMethod.GET, entity, RestPage.class).getBody();
    List<Movie> listMovies = page.getContent();
    // Add model attribute to be use on the front-end
    model.addAttribute(attributeName: "currentPage", pageNo);
    model.addAttribute(attributeName: "totalPages", page.getTotalPages());
    model.addAttribute(attributeName: "totalItems", page.getTotalElements());
    model.addAttribute(attributeName: "search", search);
    model.addAttribute(attributeName: "listMovies", listMovies);
    return "/user/home";
}

```

Figure 18: Web Endpoint for User Homepage and Movies Pagination

```

// Endpoint for paginating list of movies
no usages
@GetMapping("/page/{pageNo}")
public Page<Movie> findPaginatedMovies(@PathVariable int pageNo, @RequestParam(value = "search", required = false) String search) {
    int pageSize = 15;
    Page<Movie> page = movieService.getPaginatedMovies(pageNo, pageSize, search);
    return page;
}

```

Figure 19: Movie Service API for Movies Pagination

Based on Figure 18, the user homepage endpoint calls `findPaginated()` method which will utilize the endpoint to paginate the movies. In the `findPaginated()` method, movies are fetch using the pagination API of Movie Service in Figure 19. It accepts page numbers, size, and search field.


```

1 usage
public Page<Movie> getPaginatedMovies(int pageNo, int pageSize, String search) {
    // Check for searching
    if (search == null || search == "")
    {
        Sort sort = Sort.by(_properties: "tmdbId").ascending();
        Pageable pageable = PageRequest.of(page: pageNo - 1, pageSize, sort);
        // Return all movies
        return this.repo.findAll(pageable);
    }
    else
    {
        Sort sort = Sort.by(_properties: "tmdbId").ascending();
        Pageable pageable = PageRequest.of(page: pageNo - 1, pageSize, sort);
        // Return movies based on search field
        return this.repo.findByTitleContainingIgnoreCase(search, pageable);
    }
}

```

Figure 20: Movie service for Paginating Movies

In Figure 20 above, the `getPaginatedMovies()` method in Movie service will do the pagination by checking if there is a search field or not. If the search field is not specified, it will return all movies, otherwise it will return only movies that contain title with the search field.

3.8 Movie Details & Recommended Movies

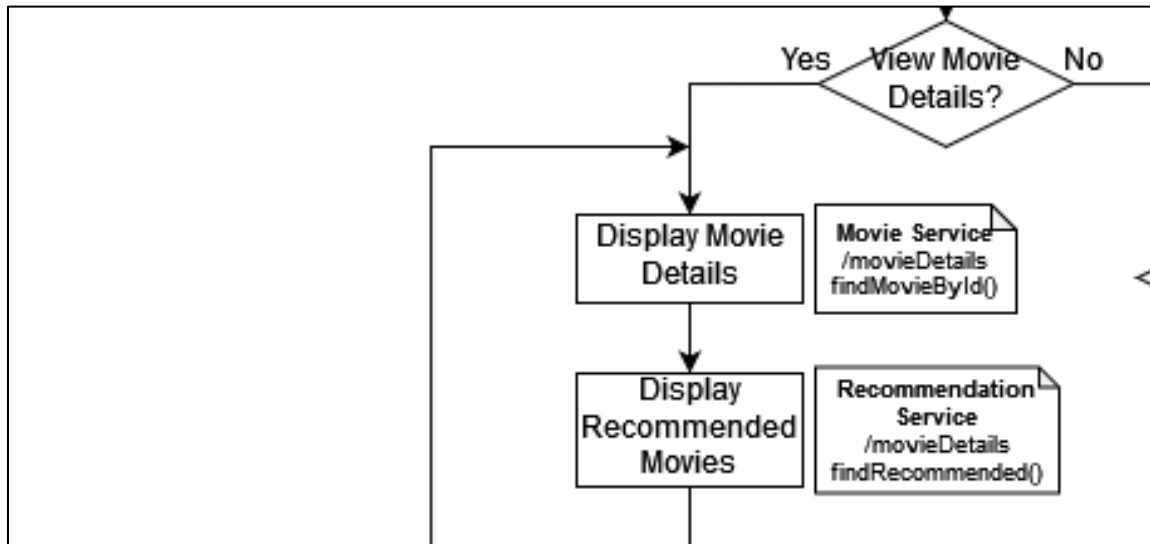


Figure 21: Movie Details & Recommended Movies Flow

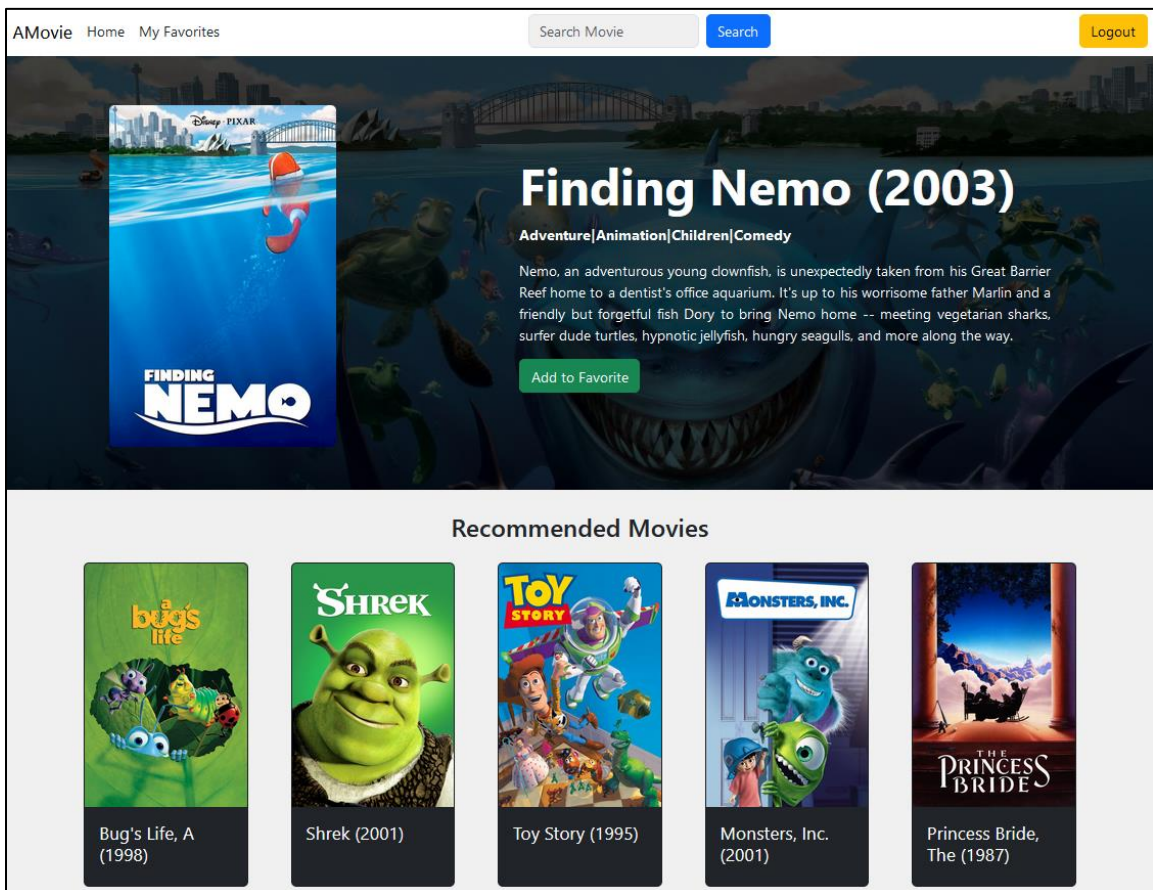


Figure 22: Movie Details & Recommended Movies

The movie details page will display additional information of the movie such as title, description, and genre. User will also be able to view recommended movies that are similar to the selected movie.

```
@GetMapping("/movieDetails")
public String getMovieDetails(@ModelAttribute("movie") Movie movie,
                              @ModelAttribute("favorite") Favorite favorite,
                              @RequestParam int id, Model model, HttpSession session) {

    try {

        HttpEntity<Integer> entity = new HttpEntity<>(id);
        // Declare object mapper to deserialize JSON object into Java object
        ObjectMapper mapper = new ObjectMapper();
        // Get recommended movie IDS
        List<Matrix> getRecommendedIDS = mapper
            .convertValue(restTemplate
                .exchange(url: "http://localhost:8086/recommended/movies/" + id, HttpMethod.GET, entity, List.class)
                .getBody(), new TypeReference<List<Matrix>>() {});

        // Find movie details based on ID to store in a List
        List<Movie> getRecommendedMovies = new ArrayList<>();
        for (Matrix data : getRecommendedIDS) {
            // Call api to get recommended movies
            Movie getRecommendedMovie = restTemplate
                .exchange(url: "http://localhost:8083/movie/movies/" + data.getMovieId(), HttpMethod.GET, entity, Movie.class)
                .getBody();
            getRecommendedMovies.add(getRecommendedMovie);
        }

        // Add model attribute to be used in the front-end
        model.addAttribute(attributeName: "recommendedMovies", getRecommendedMovies);
    }
}
```

Figure 23: Web Endpoint for Movie Details (Recommendation)

In the first part in Figure 23 above, a list of recommended movie ids will be fetched using API of Recommendation Service. Once the list is ready, it will then pass each of the ids to API of Movie Service to store in a new list of movies that contains all the movie details.

```
finally {
    // Get the rest of specific movie details
    HttpEntity<Integer> entity = new HttpEntity<>(id);
    // Call api to get specific movie
    Movie getMovie = restTemplate.exchange(url: "http://localhost:8083/movie/movies/" + id, HttpMethod.GET, entity, Movie.class)
        .getBody();
    model.addAttribute(attributeName: "movie", getMovie);
}
```

Figure 24: Web Endpoint for Movie Details (Movie Details)

In the finally block, it will then call the API of Movie Service to get movie details of the currently selected movie.

```

@Repository
public class RecommendationRepo {
    /* Since Spring JPA is quite limited for custom query, where I cannot specify custom column names,
       therefore, JDBC Template is used to fetch the recommended movies by using customize query */
    1 usage
    @Autowired
    JdbcTemplate jdbcTemplate;

    // To map the result
    1 usage
    private RowMapper<Matrix> rowMapper = (ResultSet rs, int rowNum) -> {
        Matrix matrix = new Matrix();
        matrix.setMovieId(rs.getInt(columnIndex: 1));
        return matrix;
    };

    // Custom query to find top 5 recommended movies based on the similarity
    1 usage
    public List<Matrix> findAll(int id) {
        String column = "ID_" + id;
        return jdbcTemplate.query( sql: "SELECT MOVIE_ID FROM MATRIX WHERE " + column
            + " > 0.5 AND MOVIE_ID <> " + id + " ORDER BY " + column
            + " DESC FETCH FIRST 5 ROWS ONLY", rowMapper);
    }
}

```

Figure 25: Recommendation Repository

In the Recommendation Repository as illustrated in Figure 25 above, a custom SQL query is used to fetch top 5 movie ids of recommended movies that have more than 0.5 similarity with the movie that is being selected.

3.9 Add Rating

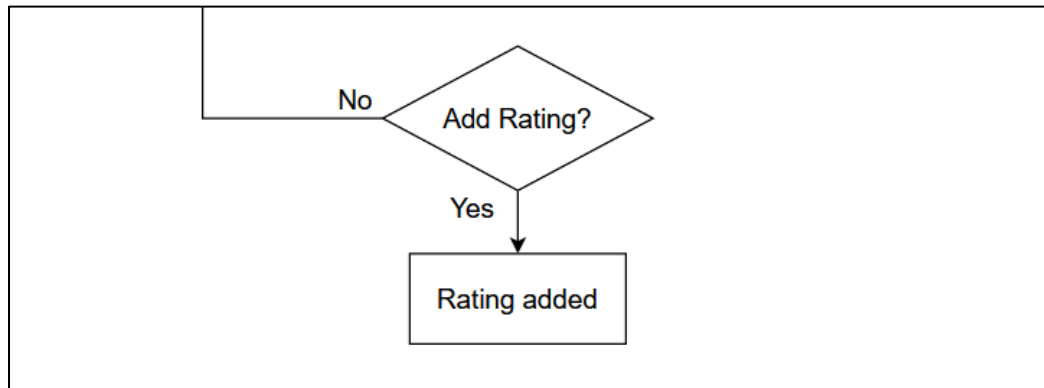


Figure 26: Add Rating Flow

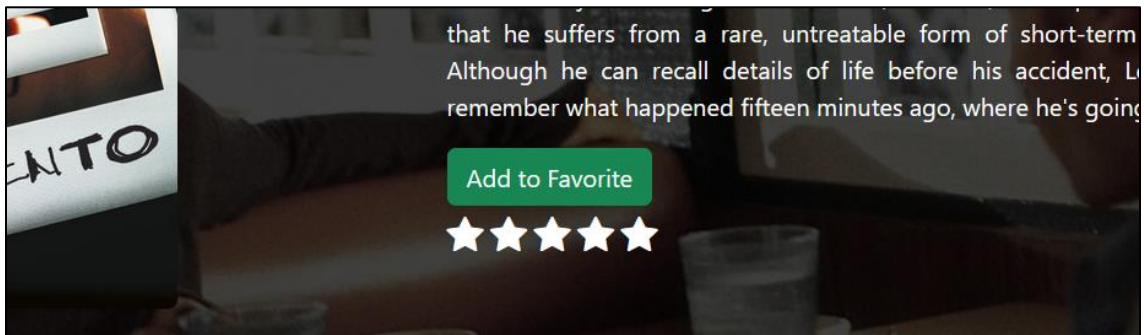


Figure 27: Add Rating

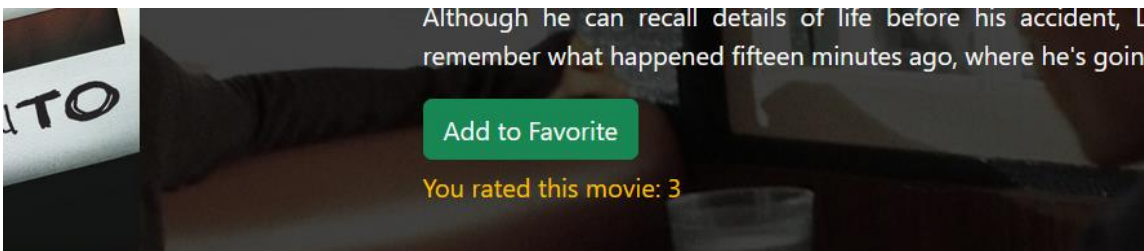


Figure 28: Add Rating Result

User is able to add rating for movie by clicking the Star icon as shown in Figure 27. Once the user has rated a movie, a text will be displayed.

```
// Endpoint for adding rating
@PostMapping("/addRating")
public String addRating(@ModelAttribute("rating") Rating rating, Model model)
{
    System.out.println(rating);
    HttpEntity<Rating> entity = new HttpEntity<Rating>(rating);
    // Call api to post data for adding new rating
    restTemplate.exchange("http://localhost:8087/rating/addRating", HttpMethod.POST, entity, Rating.class).getBody();
    return "redirect:/movieDetails?id=" + rating.getMovieId();
}
```

Figure 29: Web Endpoint for Add Rating Movie

```
public Rating addRating(Rating rating)
{
    return repo.save(rating);
}
```

Figure 30: Rating service for Add Rating Movie

Based on Figure 29, the add rating movie endpoint calls the API of Rating Service to rate a movie. It will then be redirected back to the movie details page once the movie has been rated. In Figure 30, the method `addRating()` in Rating service uses JPA to store the data in database.

3.10 Add Movie to Favorite

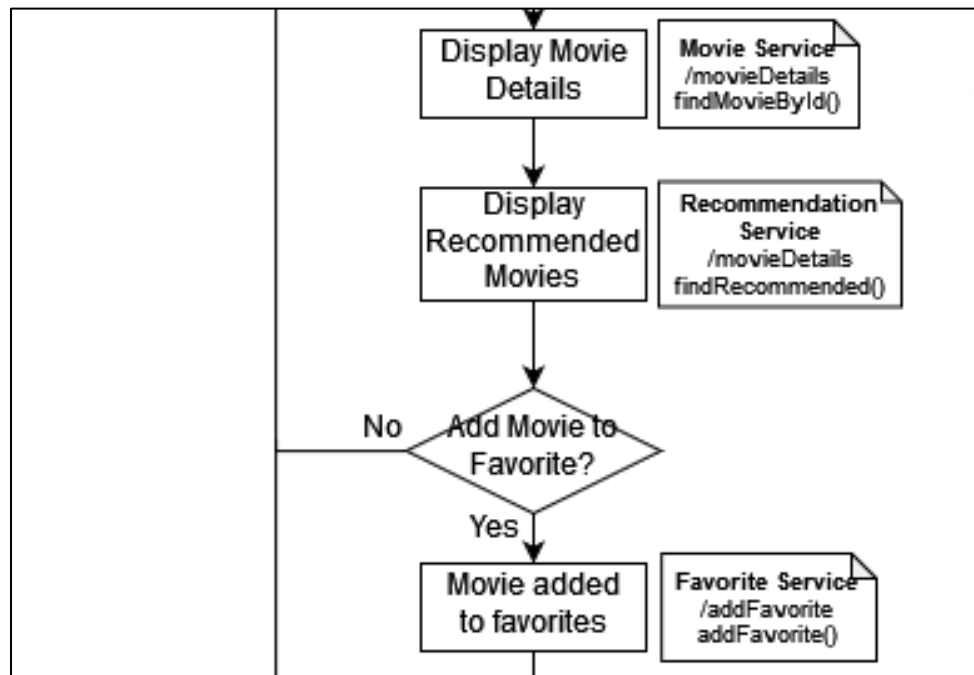


Figure 31: Add Movie to Favorite Flow

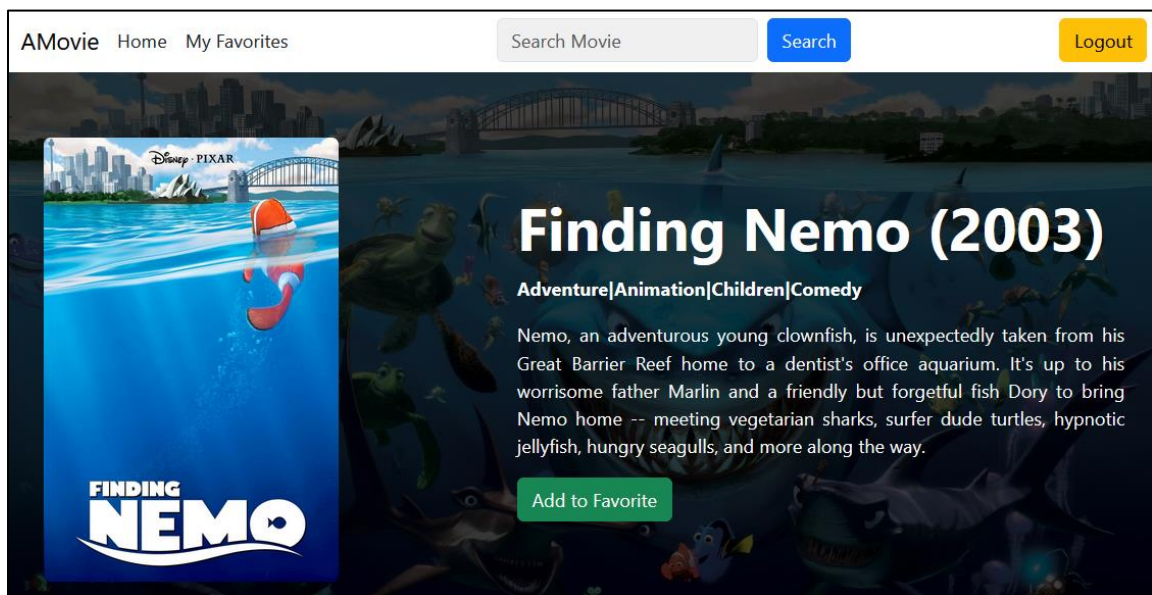


Figure 32: Add Movie to Favorite

User is able to add movie to their favorite list by clicking the button as shown in Figure 32.

```

// Endpoint for adding favorite movie
no usages
@PostMapping("/addFavorite")
public String addFavorite(@ModelAttribute("favorite") Favorite favorite, Model model)
{
    HttpEntity<Favorite> entity = new HttpEntity<>(favorite);
    // Call api to post data for adding new favorite movie
    restTemplate
        .exchange(url: "http://localhost:8085/favorite/addFavorite", HttpMethod.POST, entity, Favorite.class)
        .getBody();
    return "redirect:/movieDetails?id=" + favorite.getMovieId();
}

```

Figure 33: Web Endpoint for Adding Favorite Movie

```

1 usage
public Favorite addFavorite(Favorite favorite)
{
    return repo.save(favorite);
}

```

Figure 34: Favorite service for Adding Favorite Movie

Based on Figure 33, the add favorite movie endpoint calls the API of Favorite Service to add favorite movie. It will then be redirected back to the movie details page once the movie has been added. In Figure 34, the method addFavorite() in Favorite service uses JPA to store the data in database.

3.11 Remove Movie from Favorite

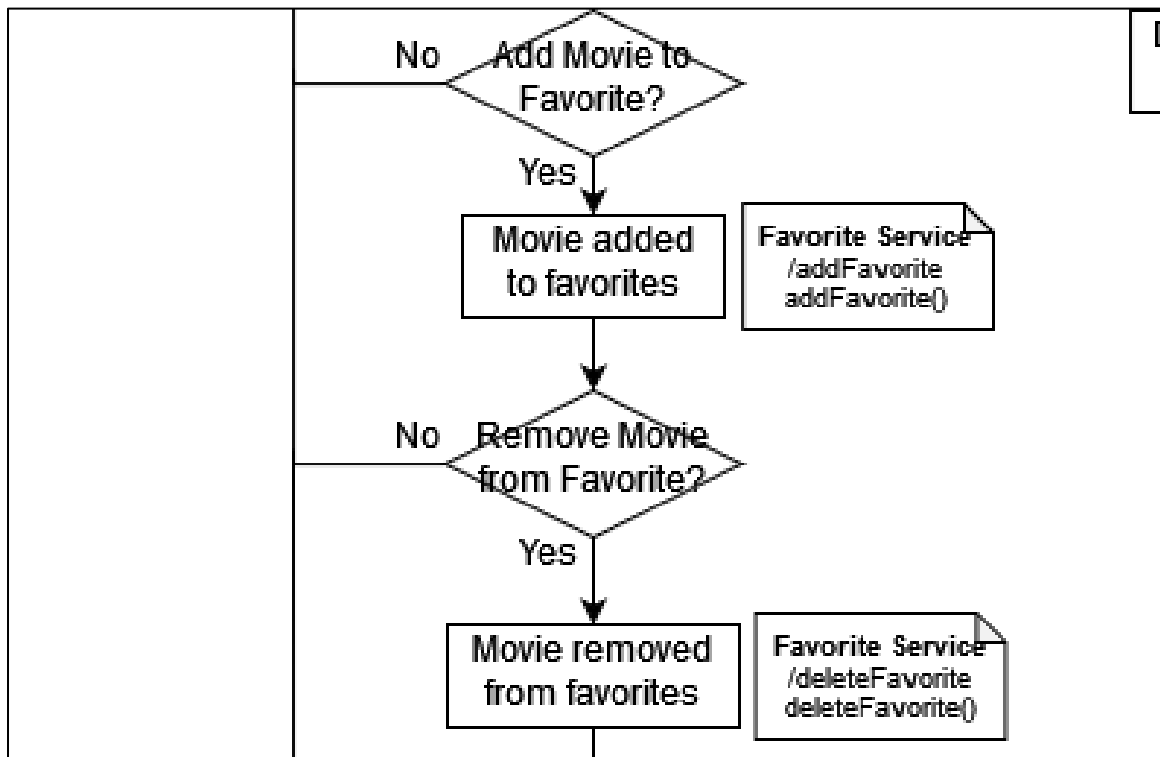


Figure 35: Remove Movie from Favorite Flow



Figure 36: Remove Movie from Favorite

As illustrated in Figure 36, if the movie is already added into favorite, the button will be disabled. However, user can remove the movie from favorite by clicking the trash icon button.

```
// Endpoint for deleting favorite movie
no usages
@PostMapping("/deleteFavorite/{id}")
public String deleteFavorite(@ModelAttribute("favorite") Favorite favorite, @PathVariable(value = "id") int id, Model model)
{
    HttpEntity<Integer> entity = new HttpEntity<>(id);
    // Call api to delete specific movie
    restTemplate.exchange(url: "http://localhost:8085/favorite/deleteFavorite/" + id, HttpMethod.POST, entity, String.class).getBody();
    return "redirect:/movieDetails?id=" + favorite.getMovieId();
}
```

Figure 37: Web Endpoint for Removing Favorite Movie

```
1 usage
public String deleteFavorite(int id)
{
    repo.deleteById(id);
    return "Movie deleted from favorites";
}
```

Figure 38: Favorite service for Removing Favorite Movie

Based on Figure 37, the delete favorite movie endpoint calls the API of Favorite Service to remove favorite movie. It will then be redirected back to the movie details page. In Figure 38, in the deleteFavorite() method in Favorite service, by using JPA, the movie will be deleted from database.

3.12 View Favorite Movies List

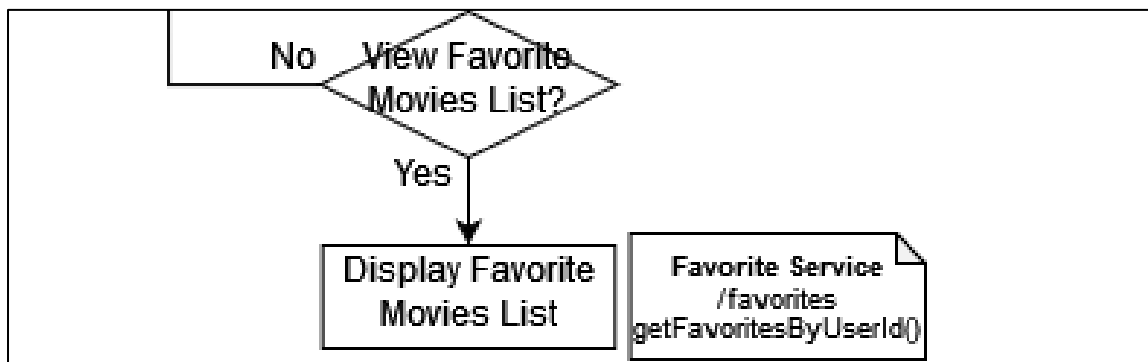


Figure 39: View Favorite Movies Flow

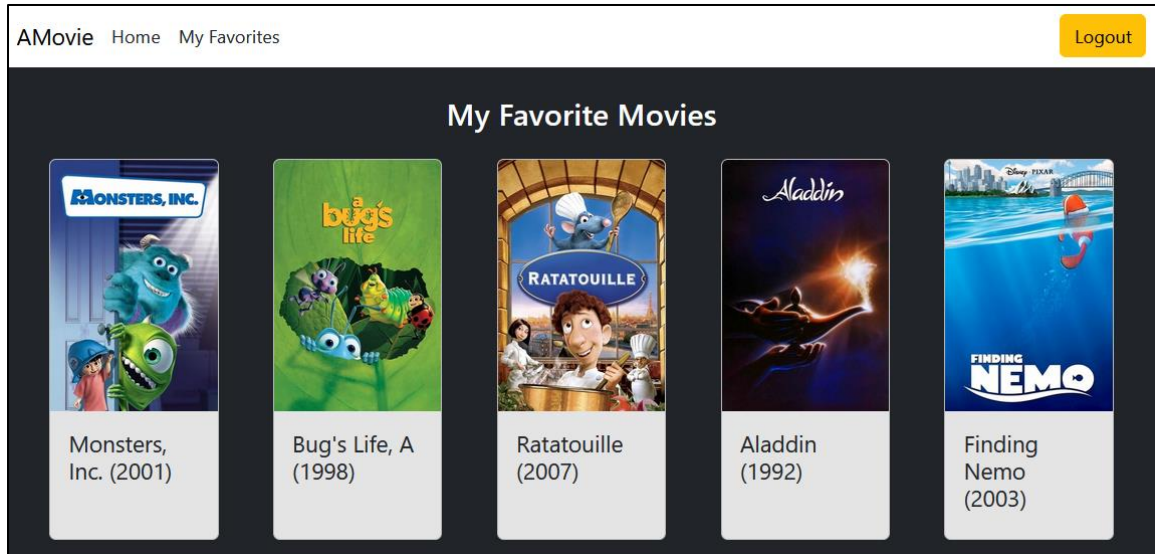


Figure 40: List of Favorite Movies

User is able to view his/her list of favorite movies. Besides, the user is able to click on each of the movie cards to view more details about the movie.

```
// Endpoint for getting all favorite movies
no usages
@GetMapping("/favorites")
public String getFavorites(@ModelAttribute("favorite") Favorite favorite, Model model, HttpSession session)
{
    HttpEntity<Favorite> entity = new HttpEntity<>(favorite);
    // Declare object mapper to deserialize JSON object into Java object
    ObjectMapper mapper = new ObjectMapper();
    // Get favorite movies IDs
    List<Favorite> getFavoriteIDs = mapper.convertValue(restTemplate
        .exchange( url: "http://localhost:8085/favorite/favorites/" + session.getAttribute(s: "userId"),
            HttpMethod.GET, entity, List.class)
        .getBody(), new TypeReference<List<Favorite>>() {});

    // Find favorite movie details based on ID to store in a List
    List<Movie> getFavoriteMovies = new ArrayList<>();
    for (Favorite data : getFavoriteIDs) {...}
    model.addAttribute( attributeName: "favorites", getFavoriteMovies);
    return "/user/favorites";
}
```

Figure 41: Web Endpoint for View Favorite Movies

Based on Figure 41, a list of favorite movie ids will be fetched using the API of Favorite Service by providing the user's id. Once the list is ready, it will then pass each of the ids to store in a new list of movies that contains all the details of the movie.

3.13 Search Movie

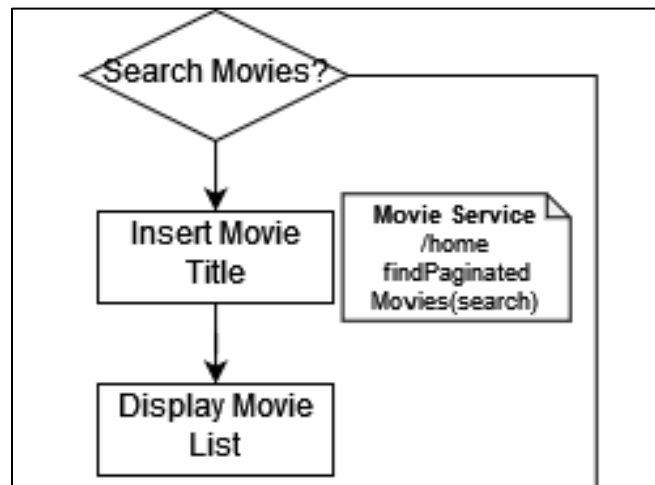


Figure 42: Search Movie Flow

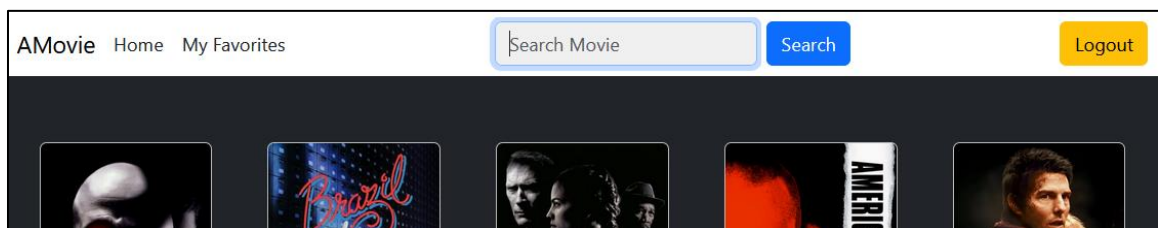


Figure 43: Search Feature

User can also search for specific movie by entering movie title on the search bar as shown in Figure 43.

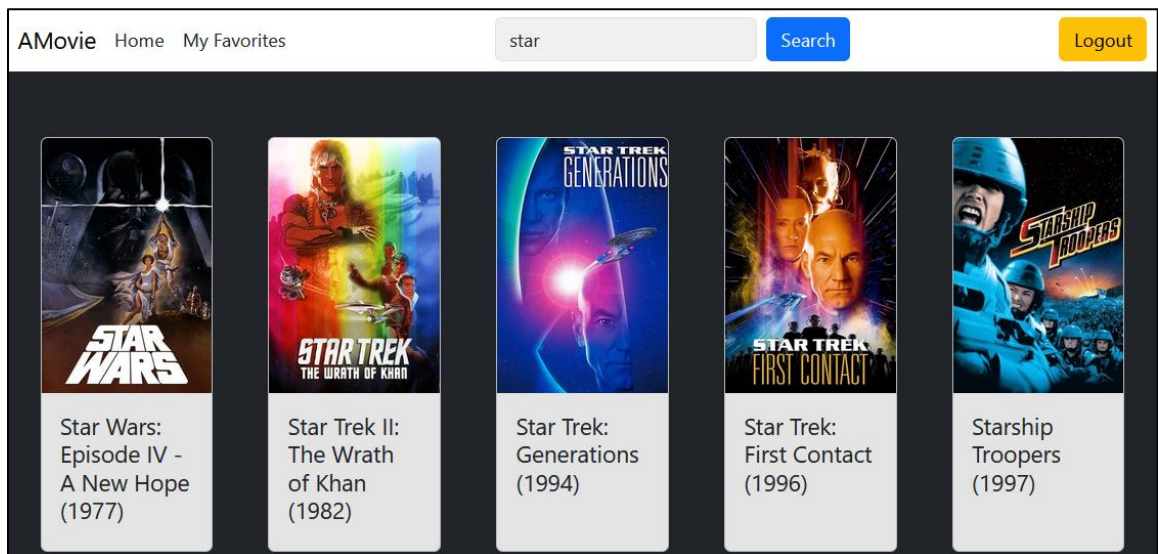


Figure 44: Search Result

Upon searching, the list of movies containing the searched title will be displayed.

```

// Endpoint for movies pagination
1 usage
@GetMapping("/page/{pageNo}")
public String findPaginated(@PathVariable(value = "pageNo") int pageNo,
    @RequestParam(value = "search", required=false) String search,
    Model model) {
    HttpEntity<Integer> entity = new HttpEntity<>(pageNo);
    // Call api to fetch movies
    RestPage<Movie> page = restTemplate
        .exchange(uri, "http://localhost:8083/movie/page/" + pageNo + "?search=" + search, HttpMethod.GET, entity, RestPage.class).getBody();
    List<Movie> listMovies = page.getContent();
    // Add model attribute to be use on the front-end
    model.addAttribute("currentPage", pageNo);
    model.addAttribute("totalPages", page.getTotalPages());
    model.addAttribute("totalItems", page.getTotalElements());
    model.addAttribute("search", search);
    model.addAttribute("listMovies", listMovies);
    return "/user/home";
}

```

Figure 45: Web Endpoint for Paginating Movies

The search feature uses the same endpoint as paginating movies but by specifying the search field. It will then display the list of movies.

3.14 Admin Dashboard

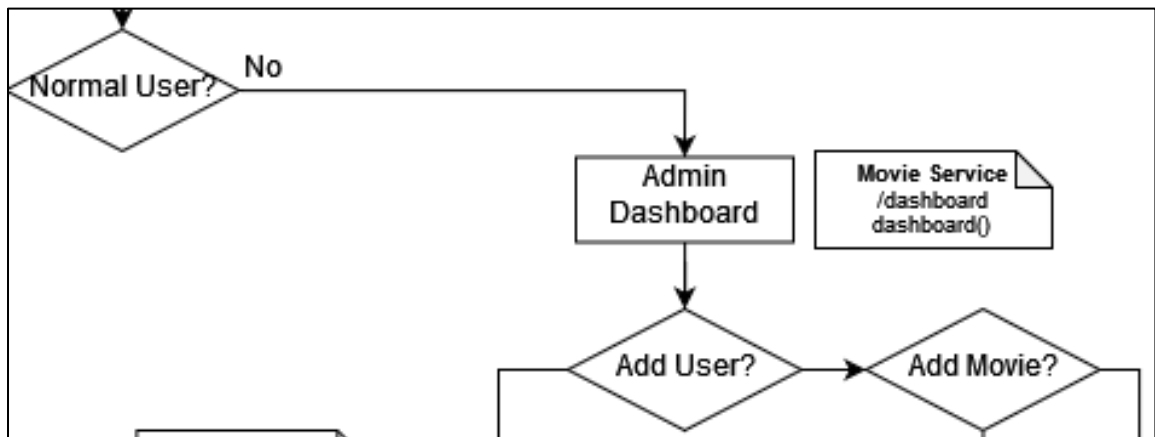


Figure 46: Admin Dashboard Flow

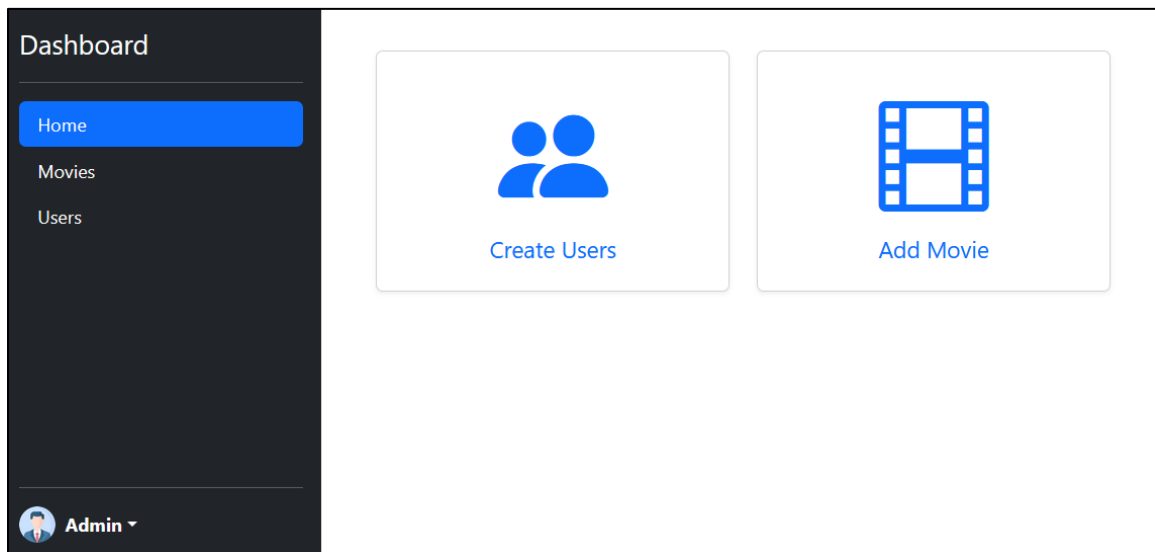


Figure 47: Admin Dashboard

The admin dashboard allows admin to perform actions or tasks like add new users, add movie, view list of movies and list of users.

3.15 Add User

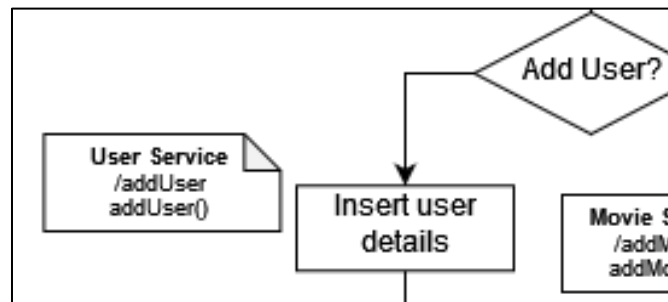


Figure 48: Add User Flow

The screenshot shows a web application interface. On the left is a dark sidebar with a 'Dashboard' header and three menu items: 'Home' (highlighted in blue), 'Movies', and 'Users'. At the bottom of the sidebar is a user profile for 'Admin'. The main content area is white and features a form titled 'Add User'. The form contains the following fields: 'First Name' (text input), 'Last Name' (text input), 'Username' (text input), 'Email' (text input), 'Role' (dropdown menu with 'Choose...' selected), 'Password' (text input), and 'Confirm Password' (text input). A green 'Add' button is located at the bottom of the form.

Figure 49: Add User Form

As illustrated in Figure 49, admin is required to enter the user details using the form to add user.

```

// Endpoint for admin to add user
no usages
@PostMapping("/addUser")
public ResponseEntity<?> addUser(@RequestBody User user)
{
    return userService.addUser(user);
}

```

Figure 50: Java API for Adding User

```

// Add User method for admin
1 usage
public ResponseEntity<?> addUser(User user)
{
    // Check existing username if exist
    if (repo.findByUsername(user.getUsername()) != null) {
        return new ResponseEntity<>( body: "Username already exists!", HttpStatus.BAD_REQUEST);
    }
    // Check existing email if exist
    else if (repo.findByEmail(user.getEmail()) != null) {
        return new ResponseEntity<>( body: "Email already exists!", HttpStatus.BAD_REQUEST);
    } else {
        // Check if password and confirm password match
        if (user.getPassword().equals(user.getConfirmPassword())) {
            // hash password using bcrypt
            user.setPassword(passEncoder.encode(user.getPassword()));
            // Save transaction
            repo.save(user);
            return new ResponseEntity<User>(user, HttpStatus.OK);
        } else {
            return new ResponseEntity<>( body: "Password does not match!", HttpStatus.BAD_REQUEST);
        }
    }
}
}

```

Figure 51: User service for Adding User

Based on Figure 50, the add user API is responsible for adding a user into the system. In Figure 51, addUser() method will do the checking of username and email if it exists and return alert response. Bcrypt is used to encode the password to store it in database. If there is an error, the API will return the error message accordingly.

3.16 View List of Users

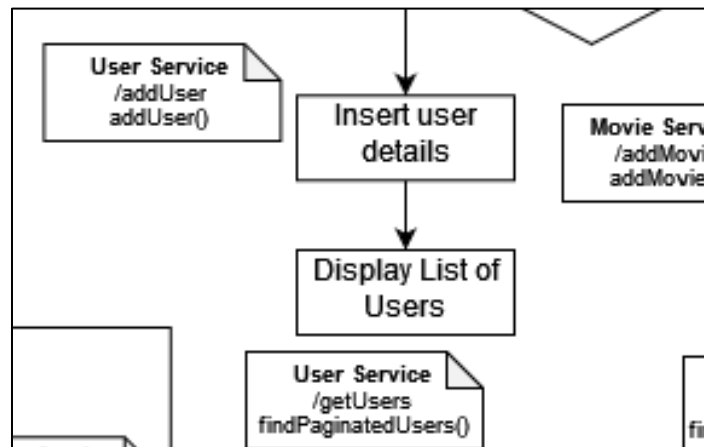


Figure 52: View List of Users Flow

Dashboard

Home

Movies

Users

Admin

List of Users

User ID	First Name	Last Name	Username	Email	Role
2052	System	Administrator	admin	admin@gmail.com	ADMIN
2053	Abu	Bakar	abubakar	abubakar@gmail.com	USER
3002	Walter	White	walter	walter@gmail.com	USER
3003	Hank	Schrader	hank	hank@gmail.com	USER
2952	Jesse	Pinkman	jesseyo	jesseyo@gmail.com	USER
3004	Tommy	Shelby	tommy	tommy@gmail.com	ADMIN

Figure 53: List of Users

Admin can view the list of users that are registered in the system.

```
// Endpoint to paginate list of users
no usages
@GetMapping("/page/{pageNo}")
public Page<User> findPaginatedUsers(@PathVariable int pageNo) {
    int pageSize = 15;
    Page<User> page = userService.getPaginatedUsers(pageNo, pageSize);
    return page;
}
```

Figure 54: Java API for Paginating List of Users

```
// Pagination method for list of user
1 usage
public Page<User> getPaginatedUsers(int pageNo, int pageSize) {
    Pageable pageable = PageRequest.of(page: pageNo - 1, pageSize);
    return this.repo.findAll(pageable);
}
```

Figure 55: User service for Paginating List of Users

In Figure 54, the API endpoint will utilize the method `getPaginatedUsers()` in User service as shown in Figure 55 to paginate the list of users.

3.17 Add Movie

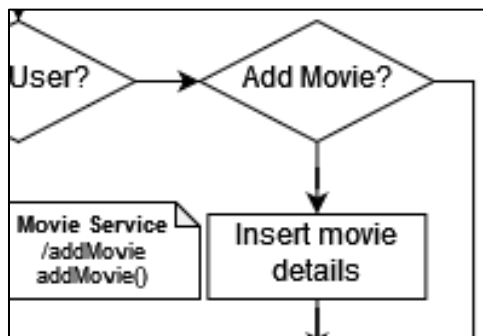


Figure 56: Add Movie Flow

The screenshot shows a web dashboard with a dark sidebar on the left. The sidebar contains a 'Dashboard' header, a 'Home' button (highlighted in blue), and links for 'Movies' and 'Users'. At the bottom of the sidebar is a user profile for 'Admin'. The main content area is white and features a form titled 'Add Movie'. The form has four input fields: 'Title', 'Poster', 'Genre', and 'Description'. Below these fields is a green 'Add' button.

Figure 57: Add Movie Form

Admin is required to enter the movie details like title, poster, genre, and description using the form to add movie as illustrated in Figure 57 above.

```
// Endpoint for adding new movie
no usages
@PostMapping("/addMovie")
public Movie addMovie(@RequestBody Movie movie)
{
    return movieService.addMovie(movie);
}
```

Figure 58: Java API for Adding Movie

```
1 usage
public Movie addMovie(Movie movie)
{
    return repo.save(movie);
}
```

Figure 59: User service for Adding Movie

Based on Figure 58, the add movie API is responsible for adding a movie into the system. In Figure 59, addMovie() method will save the movie in the database.

3.18 View List of Movies

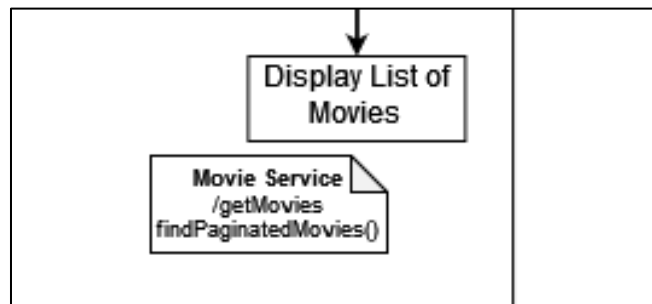


Figure 60: List of Movies Flow

<div>Dashboard</div> <div>Home</div> <div>Movies</div> <div>Users</div> <div>Admin</div>	List of Movies			
	Movie ID	Title	Genre	Description
	195561	Better Call Saul	Crime Comedy	Ex-con artist Jimmy McGill turns into a small-time attorney and goes through a series of trials and tragedies, as he transforms into his alter ego Saul Goodman, a morally challenged criminal lawyer.
	195562	Peaky Blinders	Crime	Tommy Shelby, a dangerous man, leads the Peaky Blinders, a gang based in Birmingham. Soon, Chester Campbell, an inspector, decides to nab him and put an end to the criminal activities.
	196561	Breaking Bad	Crime	Breaking bad test
	260	Star Wars: Episode IV - A New Hope (1977)	Action Adventure Sci-Fi	
	6377	Finding Nemo (2003)	Adventure Animation Children Comedy	
	356	Forrest Gump (1994)	Comedy Drama Romance War	
	2858	American Beauty (1999)	Drama Romance	
	923	Citizen Kane (1941)	Drama Mystery	
	1527	Fifth Element, The (1997)	Action Adventure Comedy Sci-Fi	
	6539	Pirates of the Caribbean: The Curse of the Black Pearl (2003)	Action Adventure Comedy Fantasy	
	6874	Kill Bill: Vol. 1 (2003)	Action Crime Thriller	
	1208	Apocalypse Now (1979)	Action Drama War	
	7361	Eternal Sunshine of the Spotless Mind (2004)	Drama Romance Sci-Fi	
	45722	Pirates of the Caribbean: Dead Man's Chest (2006)	Action Adventure Fantasy	
	924	2001: A Space Odyssey (1968)	Adventure Drama Sci-Fi	
	<div>First Prev 1 / 31 Next Last</div>			

Figure 61: List of Movies

Admin can view the list of movies that are available in the system as shown in Figure 61.

```
// Endpoint for paginating list of movies
no usages
@GetMapping("/page/{pageNo}")
public Page<Movie> findPaginatedMovies(@PathVariable int pageNo, @RequestParam(value = "search", required = false) String search) {
    int pageSize = 15;
    Page<Movie> page = movieService.getPaginatedMovies(pageNo, pageSize, search);
    return page;
}
```

Figure 62: Java API for Paginating List of Movies

```

1 usage
public Page<Movie> getPaginatedMovies(int pageNo, int pageSize, String search) {
    // Check for searching
    if (search == null || search == "")
    {
        Sort sort = Sort.by(...properties: "tmdbId").ascending();
        Pageable pageable = PageRequest.of( page: pageNo - 1, pageSize, sort);
        // Return all movies
        return this.repo.findAll(pageable);
    }
    else
    {
        Sort sort = Sort.by(...properties: "tmdbId").ascending();
        Pageable pageable = PageRequest.of( page: pageNo - 1, pageSize, sort);
        // Return movies based on search field
        return this.repo.findByTitleContainingIgnoreCase(search, pageable);
    }
}

```

Figure 63: Movie service for Paginating List of Movies

In Figure 62, the API endpoint will utilize the method `getPaginatedMovies()` in movie service as shown in Figure 63 to paginate the list of Movies.

4.0 Pearson Correlation

Pearson correlation measures the linear relationship between two variables, with a scale that ranges from -1 to 1. A negative correlation of -1 means a perfect negative relationship, a positive correlation of 1 means a perfect positive relationship, and a correlation of 0 means no relationship. The correlation coefficient is calculated by dividing the covariance of two variables by the product of their standard deviations.

Pearson correlation can be implemented in a movie recommender system to evaluate how similar users' movie choices are to one another. A high Pearson correlation value indicates similarity between two films. The Pearson correlation can be used to predict a similarity between movies based on the ratings of other users who have similar preferences.

The steps that are used in this movie recommender system to construct a movie similarity matrix based on user ratings are as follows:

1. User ratings data are gathered for a collection of movies from Grouplens dataset. This data is then transformed into a ratings matrix, where each row represents a user, and each column represents a movie. The ratings that various users have given to various movies are stored in the matrix's cells.
2. The Pearson's Correlation between two movies can be determined by comparing the ratings of the same users for both movies. Below is the formula for calculating Pearson's Correlation:

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

Figure 64: Pearson's Correlation Formula

x and y represent the scores for the two films, \bar{x} and \bar{y} represent their means and r represents the Pearson correlation coefficient.

3. The similarity matrix, which holds the similarity score between two movies in each cell, is constructed after the Pearson's correlation for each pair of movies is determined. By identifying the films that are most the same as a highly rated film by a user, the similarity matrix is used to recommend movies to user.