

Overview of the project

This project is about a movie recommendation system. This system uses **Item Based Collaborative Filtering** to recommend similar movies to the user based on the movie chosen. Apart from recommending similar movies to the user, this system also allows user to give ratings to a movie and save movies to their favourite list. The front end of this system is developed using HTML, CSS, Bootstrap while the database used is DB2.

1.0 Feature List

Listed below are the features in this system.

Validation

- Login validation
- Registration validation

Admin

- View user
- Add user
- Add movie

User

- View movie (title, genre, description & poster)
- View recommended movie
- Add favourite
- Give rating

1.1 Login Validation

The validation occurs when the admin tries to login into the website. The figure below shows the login page for the admin.

Admin Login

Username:

Password:

Login

Login as user

Figure 1: Admin login page

```
@PostMapping("/adminLogin")
public String validateLogin(@ModelAttribute("admin") Admin admin, HttpServletRequest request) {
    String result = adminService.validateAdmin(admin, request);
    if (result.equals("dashboard")) {
        Admin admindata = adminRepo.findByUsername(admin.getUsername()).get();
        request.getSession().setAttribute("username", admindata.getUsername());
    }
    return result;
}
```

Figure 2: adminLogin method in controller class

```

public String validateAdmin(Admin admin, HttpServletRequest request) {
    Optional<Admin> admindata = adminRepo.findByUsername(admin.getUsername());
    if (admindata.isPresent()){
        if(BCrypt.checkpw(admin.getPassword(), admindata.get().getPassword())){
            return "dashboard";
        }
        else {
            return "invalidCredential";
        }
    } else {
        return "unauthorized";
    }
}

```

Figure 3: validateAdmin method in service class

This code block shows the login validation process. It takes in an 'Admin' object that includes the username and password entered by the user as well as a 'HttpServletRequest' object for retrieving information from the request.

First, the 'validateAdmin' method uses the 'adminRepo.findByUsername' method to find an 'Admin' object that matches the entered username. If such an 'Admin' object exists, the method checks if the entered password matches the stored password using the 'BCrypt.checkpw' method.

If the password matches, the method returns the string 'dashboard' indicating the login was successful. If the passwords don't match, the method returns 'invalidCredential' indicating that the login credentials were incorrect.

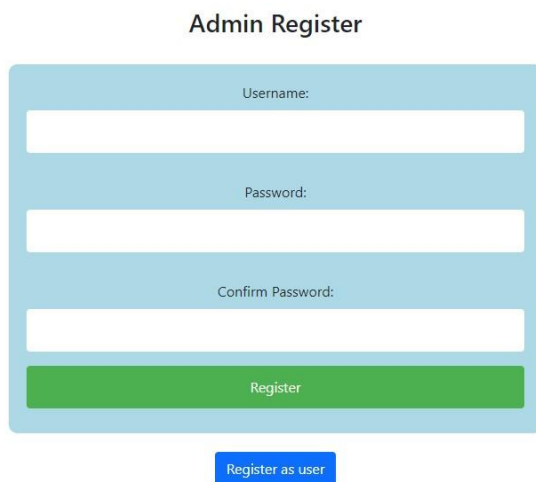
If the entered username doesn't match any 'Admin' object in the database, the method returns 'unauthorized' indicating the user is not authorized to access the system (since the admin is not yet registered).

The 'validateLogin' method then calls the 'validateAdmin' method and checks the returned value. If the value is 'dashboard', the method retrieves the 'Admin' object from the database using the entered username and sets the username as a session attribute in the 'HttpServletRequest' object.

Finally, the method returns the value returned by 'validateAdmin'. This approach ensures the admin's login credentials are validated and redirected to the correct page based on the validation result. This approach works similar to the user.

1.2 Registration validation

Similar to login, registration validation occurs when the admin tries to register into the system. The figure below shows the register page for admin.



The figure shows a web form titled "Admin Register". It is enclosed in a light blue rounded rectangle. Inside, there are three input fields: "Username:", "Password:", and "Confirm Password:", each with a white input box. Below these fields is a green button labeled "Register". Below the main form container is a separate blue button labeled "Register as user".

Figure 4: Admin register page

```
@PostMapping("/AdminRegistration")
public String registerAdmin(@ModelAttribute("admin") Admin admin) {
    if (adminService.registerAdmin(admin)) {
        return "registrationSuccess";
    } else {
        return "registrationError";
    }
}
```

Figure 5: registerAdmin method in controller class

```

public boolean registerAdmin(Admin admin) {
    Optional<Admin> existingAdmin = adminRepo.findById(admin.getUsername());
    if (existingAdmin.isPresent()) {
        return false;
    } else {
        String password = admin.getPassword();
        //store hashed password using BCrypt
        String hashedPassword = BCrypt.hashpw(password, BCrypt.gensalt());
        admin.setPassword(hashedPassword);
        admin.setConfirmPassword(hashedPassword);
        adminRepo.save(admin);
        return true;
    }
}

```

Figure 6: registerAdmin method in service class

This code block shows the validation process for registering a new admin user. The registerAdmin method takes an Admin object as an input parameter, which includes the username, password, and confirmPassword entered by the user during registration.

The method first checks if an Admin object with the same username already exists in the database using the adminRepo.findById method. If such an Admin object exists, the method returns false, indicating that the registration was unsuccessful.

If there is no existing Admin object with the same username, the method proceeds to store the user's password in a secure way. It uses the BCrypt.hashpw method to hash the password using a randomly generated salt value. The hashed password is then stored in the Admin object. Finally, the method saves the new Admin object in the database using the adminRepo.save method and returns true, indicating that the registration was successful.

The registerAdmin method is then called from the registerAdmin method of the AdminService class. If the registerAdmin method returns true, the method returns the string "registrationSuccess". If the method returns false, the method returns the string "registrationError". This approach ensures that the new admin user is only registered if the entered username is unique, and the password is stored securely in the database.

1.3 Admin

1.3.1 View user

After successful login, the admin is redirected to the dashboard. The user can view users by clicking the 'Users' navigation link. In this page, the admin will be able to view all users registered to the system.

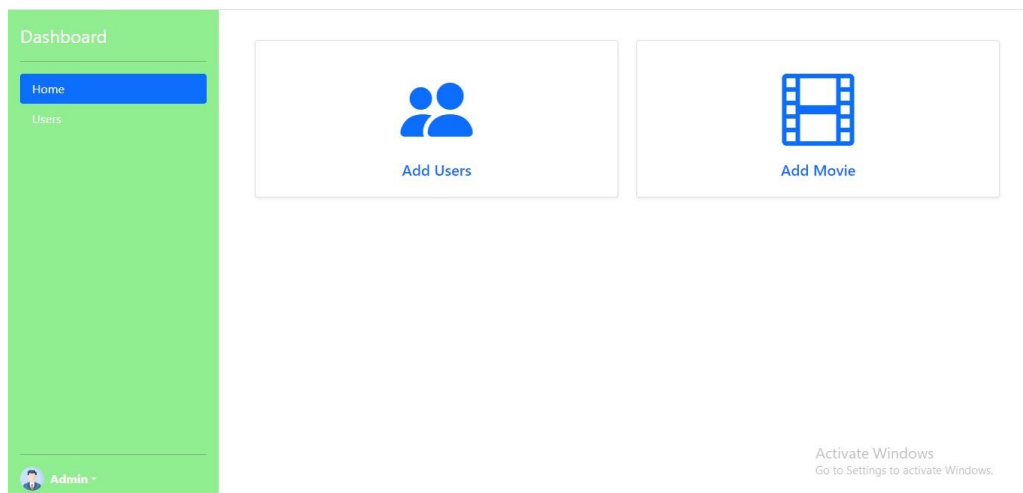


Figure 7: Admin Dashboard

Users	
UserID	Username
1	james
952	ake
1002	lex

Figure 8: Users page

1.3.2 Add user

As shown in the figure below, the admin can add new user by clicking 'Add Users' in the dashboard page.

User Register

Username:

Password:

Confirm Password:

Register

Figure 9: Add users page

1.3.3 Add movie

Add Movie

Movie Name:

Movie Genre:

Movie Description:

Movie Poster:

Choose File No file chosen

Add Movie

Figure 10: Add movie page

As shown in the figure above, the admin can add new movies by clicking 'Add movies' in the dashboard page.

```
@GetMapping("/getUsers")
public String viewUsers(Model model) {
    List<User> userList = userRepo.findAll();
    model.addAttribute("userList", userList);
    return "userView";
}
```

Figure 11: viewUsers method in controller class

When the admin clicks 'View users', the admin is redirected to the userView page as shown in the Figure 8. This method retrieves all users from the user repository and returns "userView" to be used to display the users.

```
@PostMapping("/UserRegistration")
public String userRegister(@ModelAttribute("user") User user) {
    if (userService.registerUser(user)) {
        return "success";
    } else {
        return "unsucessfull";
    }
}
```

Figure 12: userRegistration method in controller class

As shown in Figure 12, the registerUser method in the service class is expected to return a boolean value - true if the user registration was successful, and false if it was not.

If the registration is successful, the method returns "success" which means the user is successfully registered. If the registration is not successful, the method returns "unsucessfull" which means user is not registered.

```
@PostMapping("/addMovie")
public String addMovie(@ModelAttribute("movie") Movie movie) {
    HttpEntity<Movie> entity = new HttpEntity<>(movie);
    restTemplate.exchange("http://localhost:8086/addMovie", HttpMethod.POST, entity, Movie.class).getBody();
    return "dashboard";
}
```

Figure 13: addMovie method in controller method

This method allows users to add new movies by making a HTTP POST request to the "/addMovie" endpoint and then returning the "dashboard" view.

1.4 User

1.4.1 View movie

After successful login, the user will be redirected to the page as shown in the figure below. The user will be redirected to a welcome page. After clicking the button 'View movies', the user will be redirected to the page where they will be able to see all movies from the database which includes movie titles, posters & ratings.

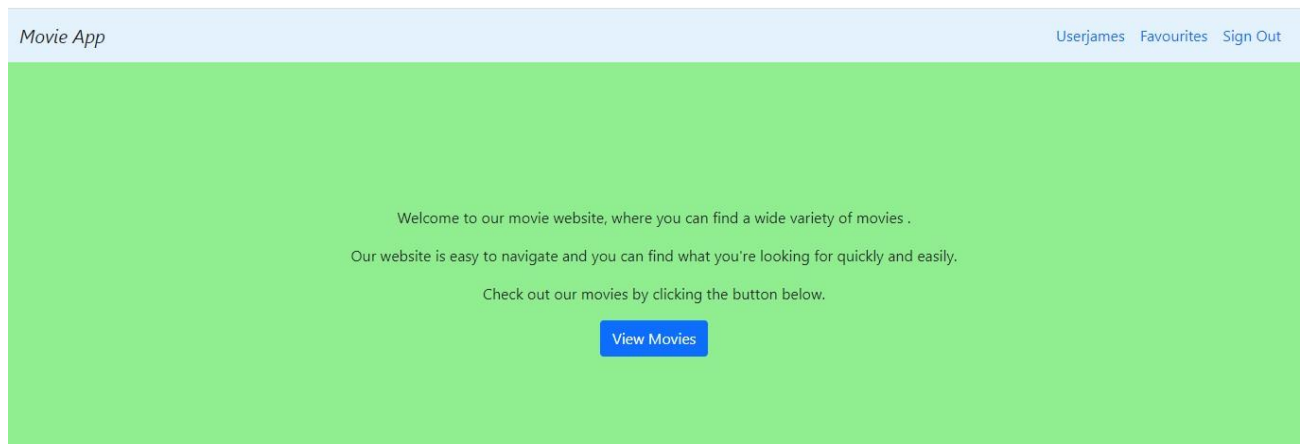


Figure 14 : User welcome page

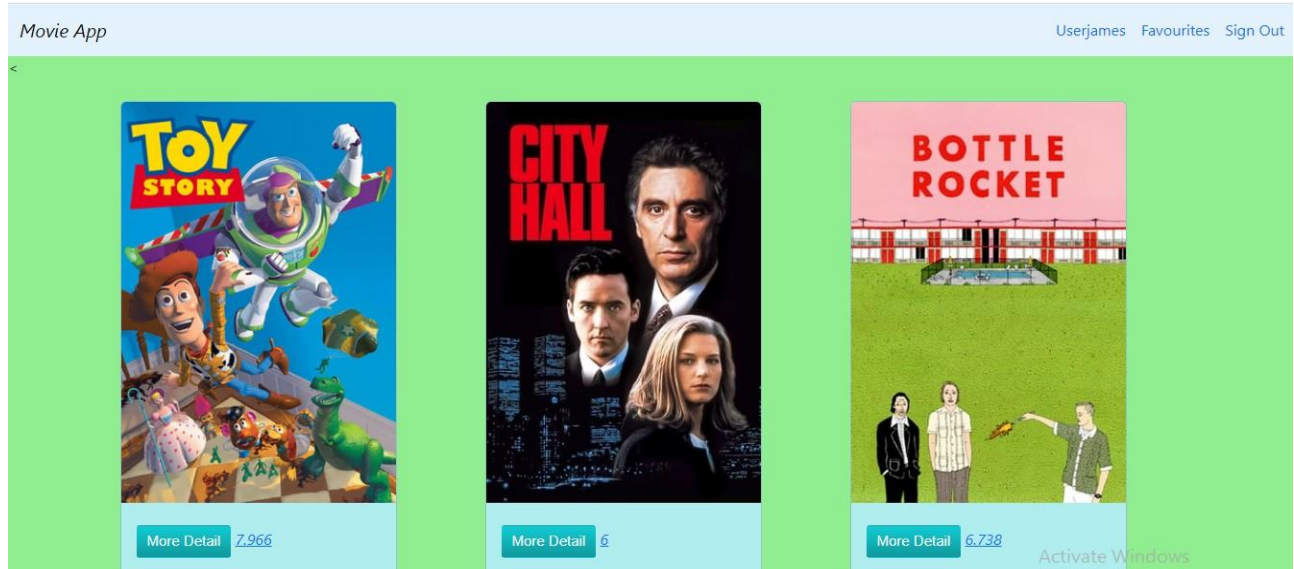


Figure 15 : User movies view page

If the user wants to view more details of a particular movie, then they can select the 'More Detail' button. This will redirect the user to another page where more details of the movie is displayed which include the description of the movie as well.

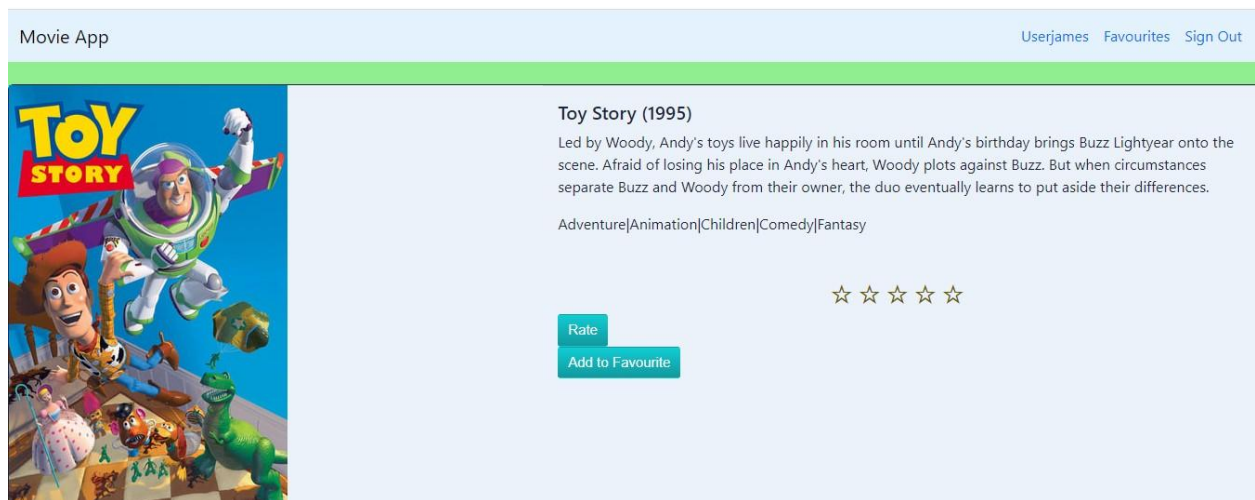


Figure 16 : User movie more details page

```
//redirect id to movie info controller method
no usages
@GetMapping("/redirectToFindMovieById")
public String redirectToFindMovieById(@RequestParam("id") int id) { return "redirect:/movieinfo/" + id; }
```

Figure 17: redirectToFindMovieById method in controller method

When the user clicks the 'More Details' button, the id of the movie selected will be taken by the movieinfo method using the @RequestParam annotation. This method is used to redirect the user to the movieinfo method that displays information about a particular movie, based on the ID of the movie.

```
/user view movie more details by its id
no usages
@GetMapping("/movieinfo/{id}")
public String movieInfoPage(@PathVariable int id, Model model) {

    HttpHeaders headers = new HttpHeaders();
    HttpEntity<Movie> entity = new HttpEntity<>(headers);
    Movie dataList = restTemplate.exchange(URI.create("http://localhost:8086/findMovieById/" + id), HttpMethod.GET, entity, Movie.class).getBody();
    model.addAttribute("dataList", dataList);

    HttpHeaders headers2 = new HttpHeaders();
    headers2.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Recommendations> entity2 = new HttpEntity<>(headers2);
    List<Recommendations> dataList = restTemplate.exchange(URI.create("http://localhost:8086/recommendations/" + id), HttpMethod.GET, entity2, new ParameterizedTypeReference<List<Recommendations>>() {}).getBody();

    HttpHeaders headers3 = new HttpHeaders();
    headers3.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Movie> entity3 = new HttpEntity<>(headers3);
    List<Movie> dataListMovie = new ArrayList<>();
    for (Recommendations recommend : dataList) {
        int movieId = recommend.getMovieId();
        List<Movie> recommendedMovies = restTemplate.exchange(URI.create("http://localhost:8086/movies/" + movieId), HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>() {}).getBody();
        dataListMovie.addAll(recommendedMovies);
    }
    model.addAttribute("dataListMovie", dataListMovie);
    return "moreDetails";
}
```

Figure 18: movieInfoPage method in controller class

The first block of code in this method is responsible for displaying detailed information about a specific movie to the user. It uses the restTemplate object to make a GET request to the findMovieById endpoint on the server running on http://localhost:8086. The id parameter is appended to the endpoint URL to specify which movie to retrieve.

The response received from the findMovieById endpoint is stored in a dataList variable of type Movie. The Model object is used to pass this dataList to the view to be rendered by the web application. The addAttribute method is used to add the dataList object to the model as an attribute with the key "dataList". The view can then access this dataList object to display detailed information about the movie to the user.

1.4.2 View Recommended Movies

In the user movie more details page, there is also a list of recommended movies displayed to the user which are similar to the movie selected by the user as shown in figure below.

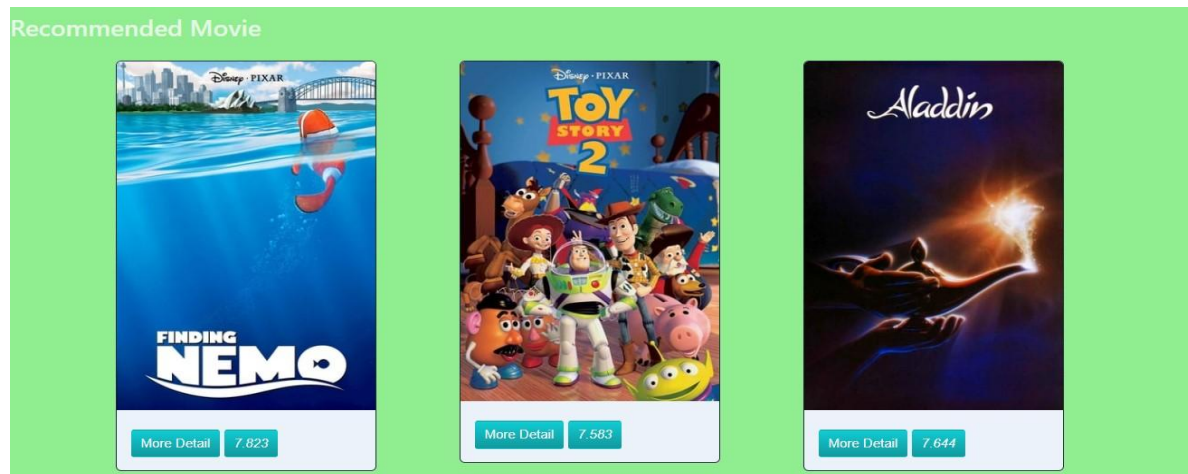


Figure 19 : Movie recommendations

In the remaining two code blocks, two HTTP GET requests are made to the server running on `http://localhost:8086` using the `restTemplate` object.

The first request is made to the `/recommendations/{id}` endpoint, where `id` is the id of the movie that the user is viewing more details about. The response received from this endpoint is a list of movie recommendations based on the movie with the specified id.

The response received from the `/recommendations/{id}` endpoint is stored in a `dataId` variable of type `List<Recommendations>`. The `HttpHeaders` object is used to set the `Accept` header to `application/json` to indicate that the client is expecting a JSON response from the server.

The second request is made inside a for loop that iterates through each recommendation in the `dataId` list. For each recommendation, a GET request is made to the `/movies/{movieId}` endpoint, where `movieId` is the id of the recommended movie. The response received from the `/movies/{movieId}` endpoint is stored in a `recommendedMovies` list of type `List<Movie>`.

The `dataListMovie` list is used to store all the recommended movies received from each request made inside the for loop. After all the requests have been made and the `dataListMovie` list is populated with the recommended movies, it is added to the `Model` object as an attribute with the key "`dataListMovie`".

Finally, the `moreDetails` view is rendered and displayed to the user. The view has access to both the detailed information about the movie being viewed and the list of recommended movies.

```
@GetMapping("/recommendations/{id}")
public List<Recommendations> findAllMovies(@PathVariable int id) { return service.findAllMovies(id); }
```

Figure 20 : findAllMovies method in controller class

```
public List<Recommendations> findAllMovies(int id) { return repo.findAll(id); }
```

Figure 21 : findAllMovies method in service class

```
private RowMapper<Recommendations> rowMapper = (ResultSet rs, int rowNum) -> {
    Recommendations matrix = new Recommendations();
    matrix.setMovieId(rs.getInt( columnIndex 1));

    return matrix;
};

1 usage
public List<Recommendations> 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 21: findAll method in repository class

The `findAll` method of `RecommendationsRepository` class is responsible for fetching the top 5 recommended movies for the given movie id from the database. It uses `JdbcTemplate` to execute a SQL query that selects the `MOVIE_ID` column from the `MATRIX` table where the

similarity score of the movie id with other movies is greater than 0.5, and orders the result by the similarity score of the movie id in descending order. It then returns the top 5 rows of the result set as a list of Recommendations objects, where each object contains the MOVIE_ID of a recommended movie.

The SQL query also excludes the movie id from the result set, so that the recommended movies are not the same as the input movie.

1.4.3 Add Favourite

On the navigation bar of each page, there is a navigation link which redirects users to save their favourite movies as shown in the figure below.

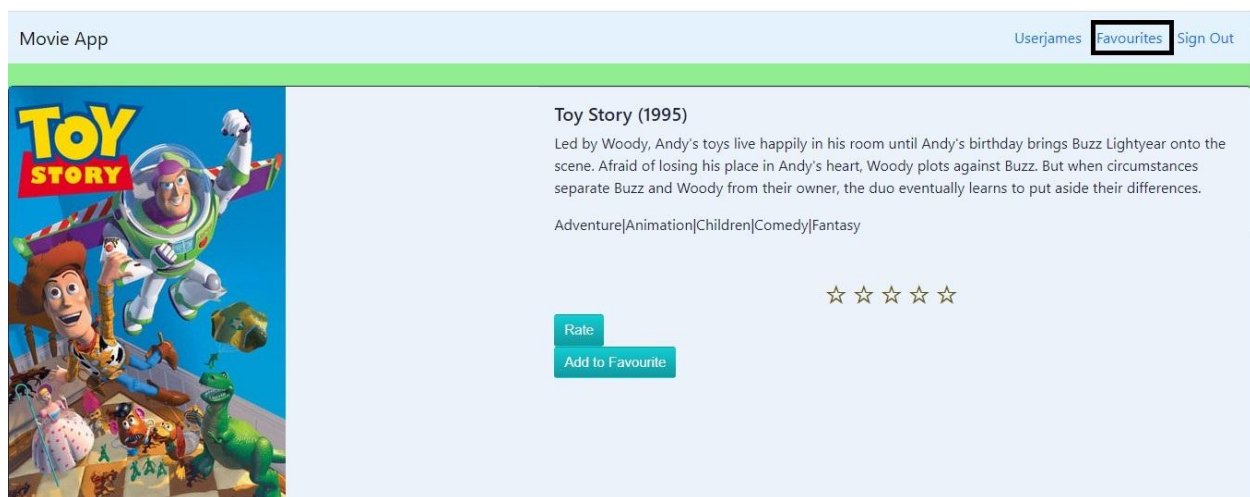


Figure 22: Add to Favourite button

The user is redirected to a favorite page which contains list of movies added to favourites by user.

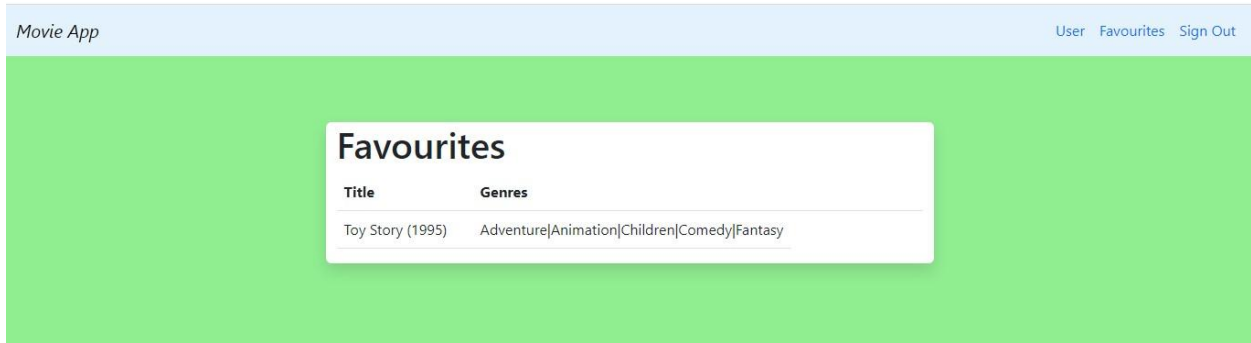


Figure 23: Favourites page

```
@GetMapping("/{fav}")
public String addFav(@RequestParam("id") int id, @RequestParam("userId") int userId, @ModelAttribute("favourite") Favourite favourite, Model model)
{
    int idf = favourite.getId();

    favourite = new Favourite(idf, id, userId);
    HttpEntity<Favourite> entity = new HttpEntity<>>(favourite);
    restTemplate.exchange("http://localhost:8086/addFav", HttpMethod.POST, entity, Favourite.class).getBody();

    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Favourite> entity2 = new HttpEntity<>>(headers);
    List<Favourite> favList = restTemplate.exchange("http://localhost:8086/favs/" + userId, HttpMethod.GET, entity2, new ParameterizedTypeReference<List<Favourite>>(){}).getBody();

    HttpHeaders headers3 = new HttpHeaders();
    headers3.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Movie> entity3 = new HttpEntity<>>(headers3);
    List<Movie> dataListMovie = new ArrayList<>();
    for (Favourite favourite1 : favList) {
        int movieId = favourite1.getMovieId();
        List<Movie> favMovies = restTemplate.exchange("http://localhost:8086/movies/" + movieId, HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>(){}).getBody();
        dataListMovie.addAll(favMovies);
    }
    model.addAttribute("dataList", dataListMovie);
    return "favourite";
}
```

Figure 24: addFav method in controller class

This method adds a movie to a user's favorites list and then return the updated list of favorite movies for that user. The method receives two parameters: the id of the movie to add to favorites and the userId of the user who wants to add the movie. The favourite object is used to create a new instance of the Favourite class. The idf attribute of the favourite object is retrieved and stored in idf. The favourite object is updated with the received id and userId. An HttpEntity is created with the updated favourite object. A POST request is made to the addFav endpoint with the HttpEntity. This endpoint adds the new favorite movie to the database. A GET request is made to the favs/{userId} endpoint to retrieve the updated list of favorite movies for the user. An HttpHeaders object is created with the MediaType set to APPLICATION_JSON. Another HttpEntity object is created with the HttpHeaders. A for loop iterates over the favList object to retrieve the movieId for each favorite movie. A GET request is made to the movies/{movieId}

endpoint to retrieve the details of each favorite movie. The retrieved movie details are stored in the `dataListmovie` object. The `dataListmovie` object is added to the model object with the key `dataList`. The method returns the favourite view.

```
@GetMapping("/viewFavs")
public String findallFav(Model model, HttpSession session)
{
    Integer userId = (Integer) session.getAttribute("userid");
    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Favourite> entity2 = new HttpEntity<>(headers);
    List<Favourite> favList = restTemplate.exchange("http://localhost:8084/favs/" + userId, HttpMethod.GET, entity2, new ParameterizedTypeReference<List<Favourite>>(){}).getBody();
    List<Integer> dataListid = new ArrayList<>();
    for (Favourite favourite : favList) {
        int favid = favourite.getId();
        dataListid.add(favid);
    }
    model.addAttribute("dataid", dataListid);

    HttpHeaders headers3 = new HttpHeaders();
    headers3.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Movie> entity3 = new HttpEntity<>(headers3);
    List<Movie> dataListmovie = new ArrayList<>();
    for (Favourite favourite1 : favList) {
        int movieId = favourite1.getMovieId();
        List<Movie> favMovies = restTemplate.exchange("http://localhost:8084/movies/" + movieId, HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>(){}).getBody();
        dataListmovie.addAll(favMovies);
    }
    model.addAttribute("dataList", dataListmovie);
    return "favourite";
}
```

Figure 25: *findAllFav method in controller class*

This `findAllFav` method retrieves all the favourites of a user from the server using a GET request to `/favs/{id}` where `id` is the user ID. Then it uses the retrieved list of favourites to extract the IDs of movies that the user has added to their favourites. Next, it sends a GET request to `/movies/{id}` for each movie ID in the retrieved list to get the details of the movies. Finally, it adds the retrieved list of movies to the model and returns the favourite view.

1.4.4 Give Rating

As shown in figure above, user can give ratings to the movie by selecting the stars.

```
@PostMapping("/rate")
public String addRating(@ModelAttribute("rating") Rating rating, Model model)
{
    String timestamp = rating.getTimestamp();
    int movieId = rating.getMovieId();
    int userId = rating.getUserId();
    double movierating = rating.getMovierating();
    int id = rating.getId();
    rating = new Rating(id,userId,movieId,movierating,timestamp);
    HttpEntity<Rating> entity = new HttpEntity<>>(rating);
    restTemplate.exchange( url = "http://localhost:8086/addRating", HttpMethod.POST, entity, Rating.class).getBody();

    /*List all movie to display at home*/
    HttpHeaders headers3 = new HttpHeaders();
    headers3.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<Movie> entity3 = new HttpEntity<>>(headers3);
    List<Movie> dataList = restTemplate.exchange( url = "http://localhost:8086/movies", HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>() {} ).getBody();
    model.addAttribute( attributeName = "dataList", dataList);
    return "userMoviesView";
}
```

Figure 26: addRating method in controller class

This method handles a POST request to add a rating for a movie.

The method takes in a Rating object as a parameter with attributes such as movieId, userId, movierating, timestamp, and id. It then creates an HttpEntity object with the Rating object and uses a RestTemplate object to make a POST request to the endpoint "http://localhost:8086/addRating".

After the rating is added, the method retrieves a list of all movies from the same API endpoint "http://localhost:8086/movies" and adds it to the model. The method then returns a string "userMoviesView" which corresponds to the name of the view that will be rendered.

2.0 Module

2.1 Login Flowchart (Admin)

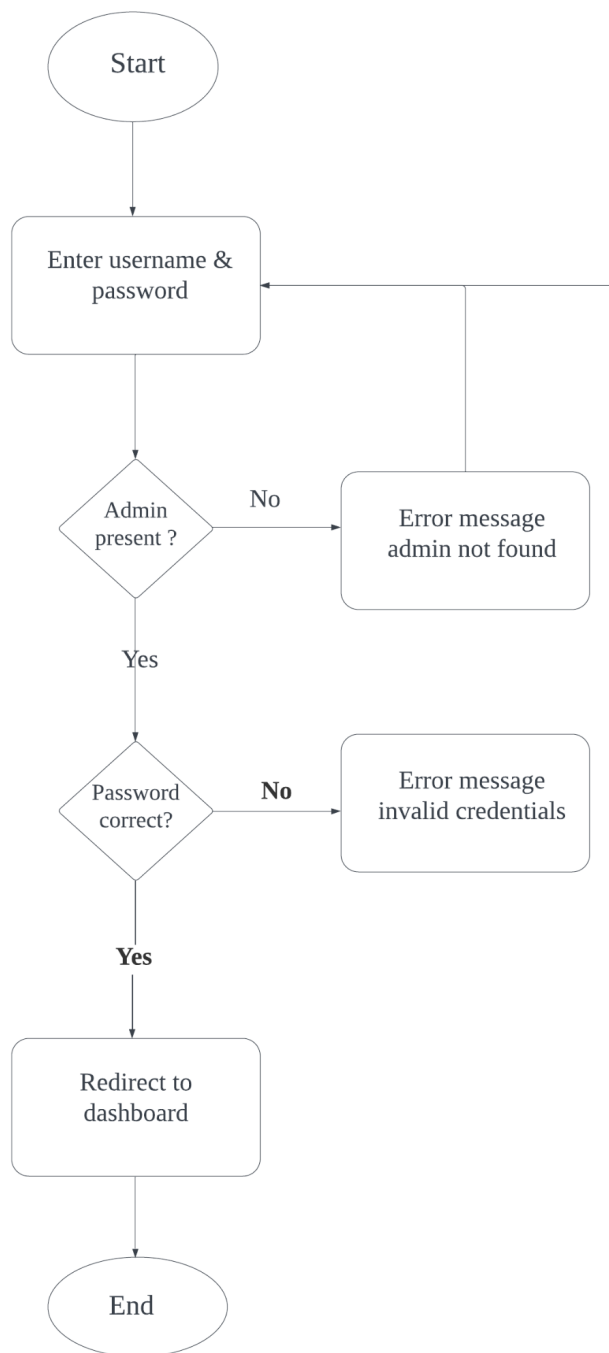


Figure 27: Admin login flowchart

Figure 27 is the flow chart for login page. The user will first enter the username and password. If the user does not exist, error message User Not Exist will appear and the page will be redirect to the login page back. If user exist but the password is wrong, error message login invalid will appear and the page will be redirected to the login page. If the password is correct then the user will be directed to the dashboard.

2.2 Login Flowchart (User)

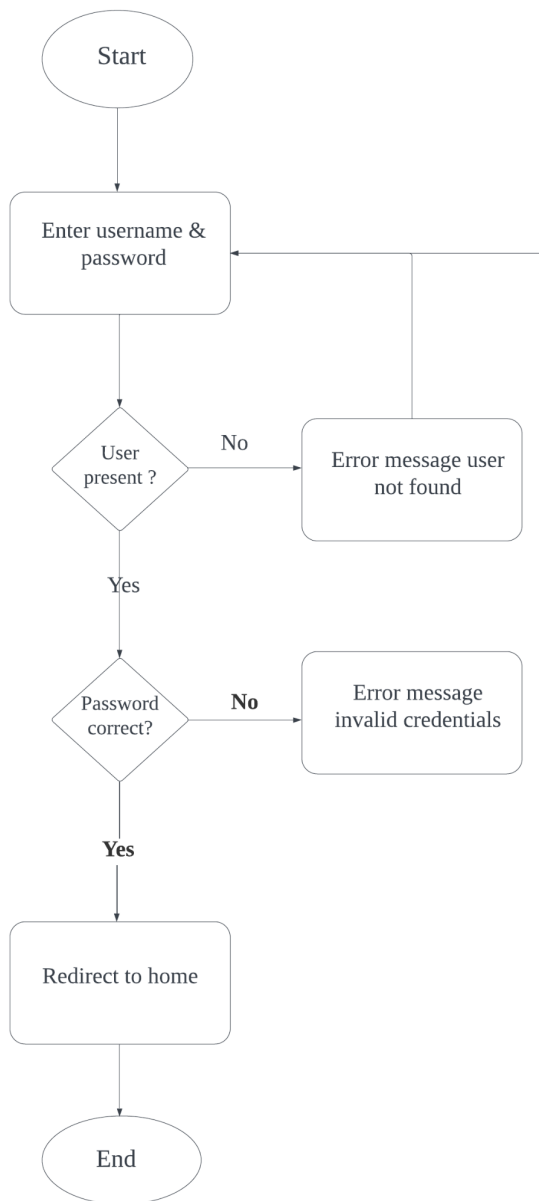


Figure 28: User login flowchart

Figure 28 is the flow chart for login page. The user will first enter the username and password. If the user does not exist, error message User Not Exist will appear and the page will be redirect to the login page back. If user exist but the password is wrong, error message login invalid will appear and the page will be redirected to the login page. If the password is correct then the user will be directed to the home page.

2.3 Admin (Flowchart)

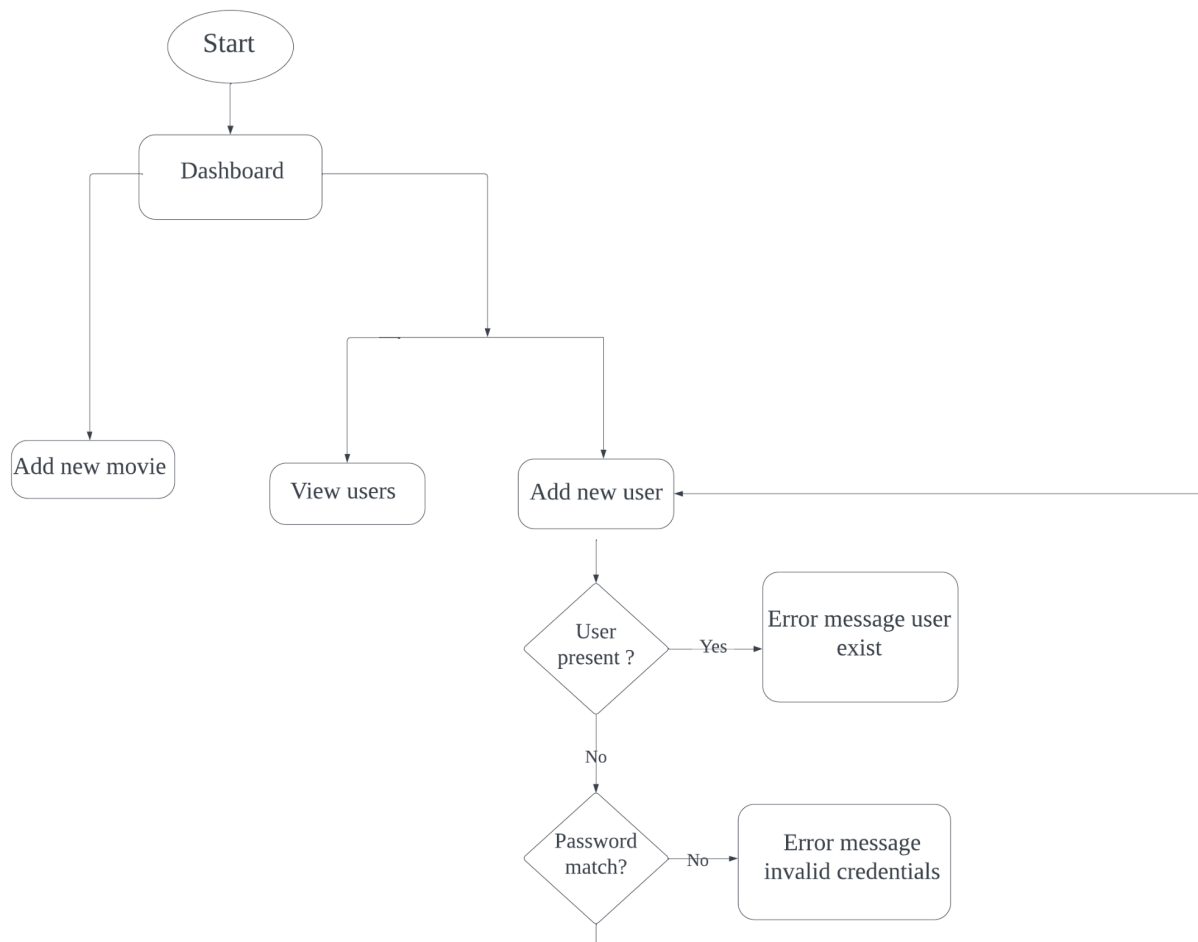


Figure 29: Admin flowchart

Figure 29 is the flowchart for admin page. Admin can view users, add new user or add new movie. In the movie page, the user can add new movie. When adding new user, if the username inserted exist, the error message User exist will appear and the page is redirected back. If the passwords do not match, an error message indicating invalid credentials is displayed and the page is redirected back.

2.4 User (Flowchart)

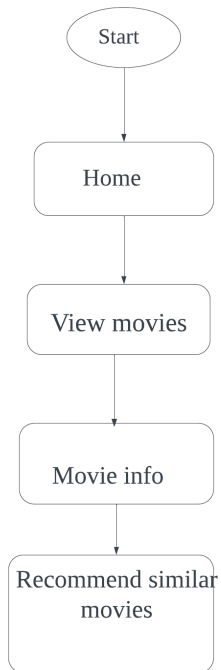


Figure 30: User flowchart

Figure 30 is the flowchart for user. After login, user is direct to home page which displays a simple welcome message and a small paragraph of what the website is about. After clicking the view movies button, user is redirected to the page to view all movies. After clicking the more details button of a particular movie, user is redirected to view details about the chosen movie. In that page, user will also be able to view similar movies recommended to the user.

3.0 Db2

3.1 Entity Relationship Diagram (ERD)

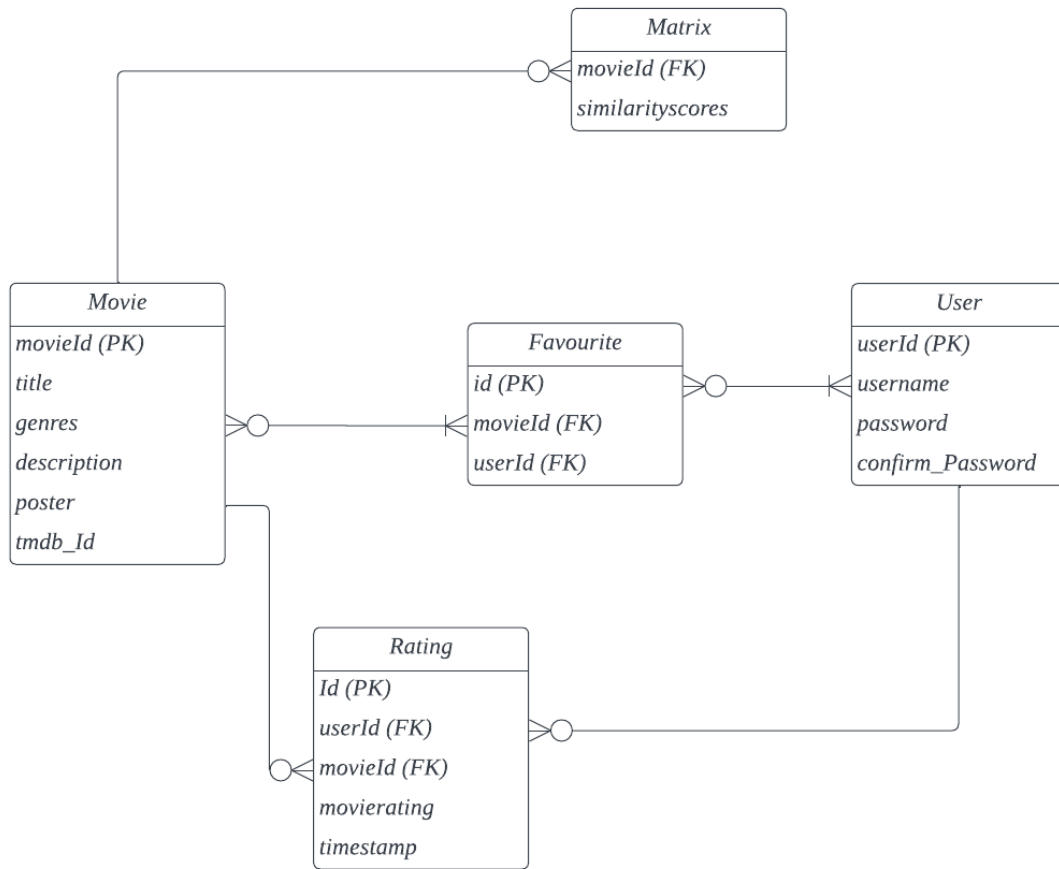


Figure 31: ERD

3.2 Data Dictionary

3.2.1 Movie

Field Name	Data Type	Field Length	Constraint
movieid	Int	4	Primary Key
title	Varchar	255	
genres	Varchar	255	
description	Varchar	255	
poster	Varchar	255	
tmdb_Id	Int	4	Not null

3.2.2 User

Field Name	Data Type	Field Length	Constraint
userId	Int	4	Primary Key
username	Varchar	255	
password	Varchar	255	
confirm_Password	Varchar	255	

3.2.3 Rating

Field Name	Data Type	Field Length	Constraint
Id	Int	4	Primary Key
userId	Int	4	
movierating	Int	4	
timestamp	Varchar	255	

3.2.4 Favourite

Field Name	Data Type	Field Length	Constraint
id	Int	4	Primary Key
movieid	Int	4	
userid	Int	4	

4.0 Monolithic architecture

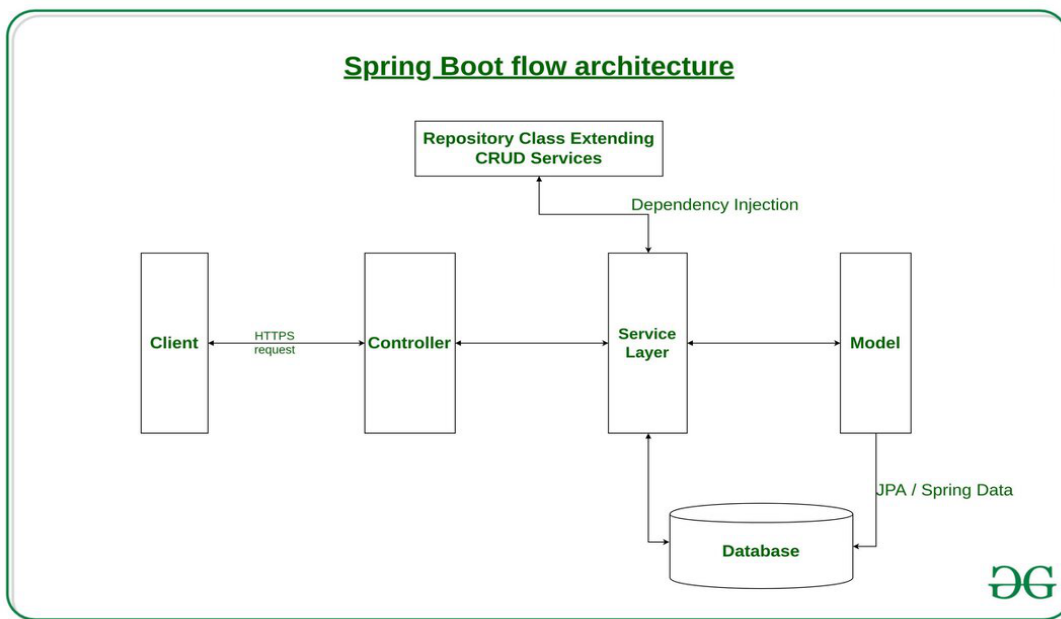


Figure 32: Monolithic architecture

Figure 32 shows the architecture of a typical monolithic Spring Boot application. In a monolithic architecture, all the application's functionalities are grouped together as a single, large application. Let's break down the image to understand the different components and how they interact with each other:

1. User Interface: The user interface (UI) is responsible for presenting the application to the user. It could be a web-based UI, a mobile app UI, or a desktop UI. In the image, the UI is represented by the "Web Browser" icon.

2. **Web Server:** The web server is responsible for handling incoming requests from the user interface and routing them to the appropriate components of the application. In the image, the web server is represented by the "Tomcat Server" icon.
3. **Spring Boot Application:** The Spring Boot application is the core of the application. It's responsible for handling business logic, data storage, and integrating with other components of the application. In the image, the Spring Boot application is represented by the "Spring Boot" icon.
4. **Spring Data:** Spring Data is a module of the Spring framework that provides a simple and consistent approach to data access. It provides support for a variety of data stores, including SQL databases, NoSQL databases, and in-memory data stores. In the image, Spring Data is represented by the "Database" icon.
5. **Spring Security:** Spring Security is a module of the Spring framework that provides authentication and authorization services for web applications. It provides a flexible and customizable framework for implementing security features in Spring applications. In the image, Spring Security is represented by the "Security" icon.
6. **External Services:** External services are services that the Spring Boot application interacts with to perform its functions. These could be third-party APIs, cloud services, or other services within the same network. In the image, external services are represented by the "Third Party APIs" icon.

In a monolithic architecture, all the components of the application are packaged and deployed together. The Spring Boot application is typically deployed as a single executable JAR file that contains all the dependencies and configurations needed to run the application. This approach provides a simple and easy-to-manage deployment process, but it can also lead to some challenges, such as scaling issues and longer deployment times.

Overall, the monolithic architecture of Spring Boot applications provides a simple and easy-to-manage approach to building and deploying applications. However, as applications grow in complexity and scale, it may become necessary to adopt a microservices architecture to address some of the challenges associated with monolithic architectures.

5.0 Pearson Correlation

Pearson correlation is a measure that quantifies the linear relationship between two variables, using a scale ranging from -1 to 1. If the correlation value is -1, it indicates a perfect negative relationship, while a value of 1 implies a perfect positive relationship, and 0 means no relationship between the variables. The correlation coefficient is computed by dividing the covariance of two variables by their standard deviations' product.

To determine movie similarities in a recommender system, Pearson correlation can be applied to assess how similar users' movie preferences are. The higher the correlation value, the more similar two films are. This correlation method can forecast the similarity between movies based on other users' ratings, which have comparable preferences.

The recommender system for constructing a movie similarity matrix based on user ratings follows these steps:

1. Ratings data are collected from Grouplens dataset for several movies, then converted into a ratings matrix, where each row represents a user, and each column represents a movie. The matrix cells store the various ratings given by different users for different movies.
2. Pearson's correlation between two films can be determined by comparing the same users' ratings for both movies. The formula for calculating Pearson's correlation is provided below:

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

Figure 33: Pearson Correlation Formula

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

3. Once the Pearson's correlation coefficient for every pair of movies is calculated, the similarity matrix is built, which contains the similarity score for each pair of movies in its cells. This similarity matrix is then utilized to recommend movies to users by identifying the movies that are most similar to a highly rated movie by the user.