Software Requirements Specification

For

Movie Recommendation System (MRS)

Prepared By:

Muhammad Hairulwafiq bin Hairunizam

Collabera,

7 February 2023

Table of Contents

Table of Figures

1. Introduction

1.1 Document Purpose

The purpose of this Software Requirements Specification (SRS) is to outline the requirements for Movie Recommendation System (MRS). In addition to said requirements, the document also provides the specifications on the scope, features, operating environment as well as design and implementation constraints. Hence, this document should act as a guideline for efficient and well-managed project completion and serve as a reference in the future. The primary audience of this SRS document will be the development team followed by the stakeholder of the project. Therefore, this SRS should convey the required functionality and represent an agreement between the involved parties.

1.2 Product Scope

In this part, the SRS will cover the system of the MRS. The objective of this project is to create and implement a dedicated movie recommendation system using Pearson's Correlation Matrix to calculate and get the recommended movies. This project will be a web-based system. The system will be used by the users and admin. Users are required to create an account to access the system for them to start browse through movies, view the details of the movies, see recommended movies and add movies to their favourite. In addition, the admin is able to login to the system, manage movies by adding a new movie into the database, promoting a user to admin, and delete users.

1.3 Intended Audience and Document Overview

The target audiences for this document are the developers and the users. The purpose of this document is to describe the SRS in detail. Several models have been described in the documentation including the Use Case Diagram (UCD), Flow Diagram and Entity Relation Diagram (ERD).

2. Overall Description

2.1 Product Perspective

This product is a web-based application system designed to help users discover new movies, learn more about the details of the movie, and getting recommendations based on the movie they choose. The user is able to see through the list of movies available in the system. When the user chooses a movie, the details of the movie will be shown to the customer, and they can add the movie to their favourite. In each movie page, a recommended movie will be shown based on the movie selected.

2.2 Product Functionality

Create Account – The create account function shall allow new users to register their account for the system by entering the details needed.

Log In – The login function shall provide the admin and user with the ability to log in to the system.

View Movie – The view movie function shall provide the user with the ability to view the list of movies that is available in the system.

View Movie Details – The view movie details function shall provide the user with the ability to view the details of the movie.

Get Movie Recommendation – The get movie recommendation function shall provide the user with the ability to view recommended movies based on the movie they choose.

Add To Favourite – The add to favourite function shall allow the user to add a movie they choose to their favourite movie.

Remove From Favourite – The remove from favourite function shall allow the user to remove a movie from their favourite movie list.

Rate Movie – Rate movie function shall provide the user with the ability to give a rating to the movie.

Add Movie – Add movie function shall provide the admin with the ability to add a new movie by entering their title, poster path, description, and genres.

Delete Movie – Delete movie function shall provide the admin with the ability to delete a movie from the database.

Delete User – Delete user function shall provide admin with the ability to delete a user.

Promote To Admin – Promote to admin function shall provide admin with the ability to promote a normal user to admin.

2.3 Users and Characteristics

User Type	User	Admin
Characteristics	A user of the system that can view and movie to favourite	Has the authority to manage the content and user of the system
Frequency of Use	High Frequency	Medium Frequency
Limitation	Medium Limitation	Low Limitation
Skill	No skill needed	Partially skilled. Should be able to understand the system as well as handling the system interface, elements, and components
Experience	No special experience required. Just the basic ability to use the system.	Expected to have experience in handling user and data.

Table 1: Users and Characteristics

2.4 Operating Environment

Minimum System Requirements:

CPU	Intel Pentium 4 processor or later that's SSE3 capable
RAM	4 GB
Operating System	Windows 8 / 10 / 11 MacOS Linux
Browser	Microsoft Edge / Firefox / Chrome
Network	Wired / Wireless with speed at least 512kbps.
Video	AMD Radeon Xpress 1200 Series or NVIDIA GeForce FX 5200

Table 2: Minimum System Requirements

2.5 User Documentation

1. User manual

   Contain a written guide and associated images to give assistance for people using the system.

2. Installation manual

   A written guide containing details and steps with associated images to give assistance for developers that is going to continue maintaining the system.

3. System Features

3.1 System Architecture

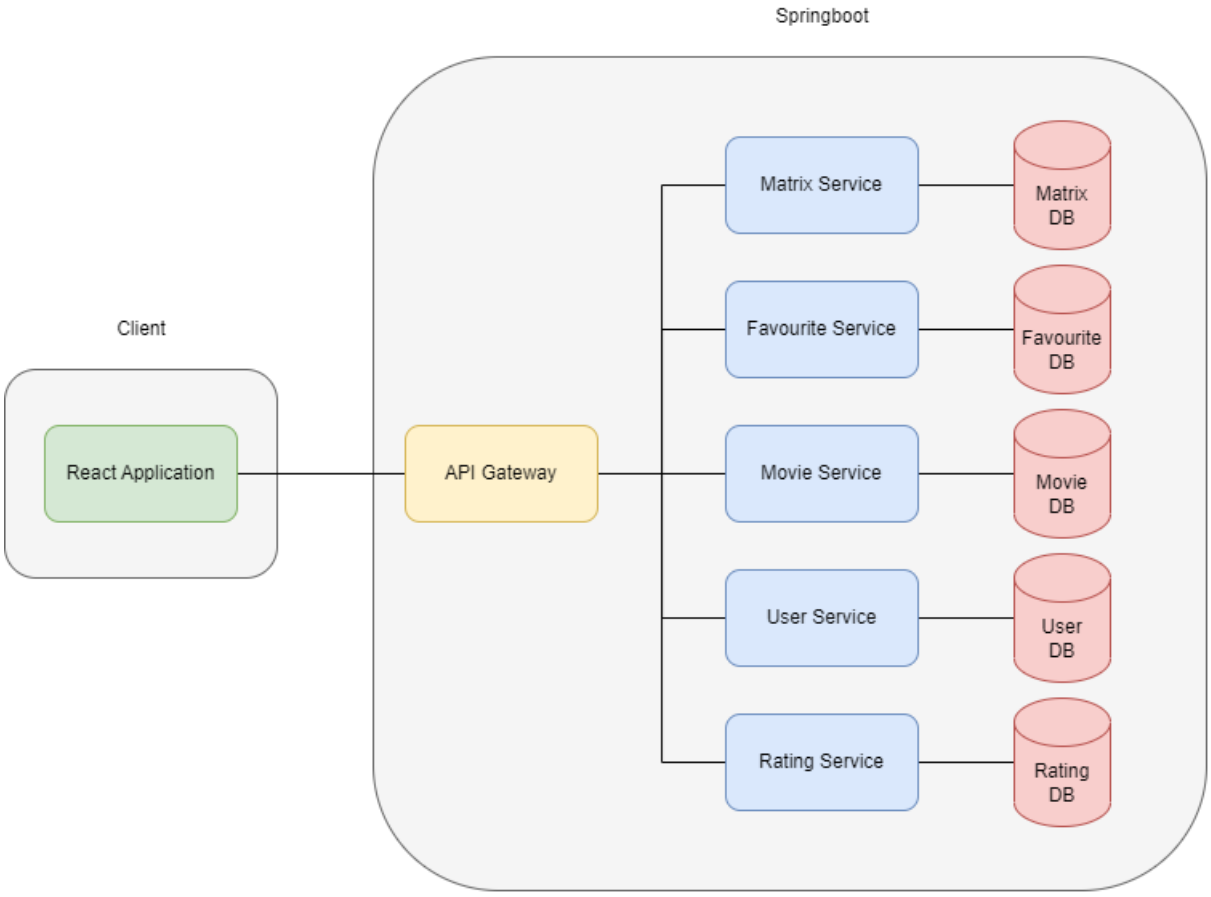


Figure 1: System Architecture

The system architecture for the Movie Recommendation System consists of client side and server side. Figure 1 shows the connection between the client and server-side system. The client side is written in ReactJS, HTML, CSS, Bootstrap, and JavaScript. A user of the application interacts with the react application using a browser. The API request are made through the API gateway using specific API call for specific process. The connection between the client side and the server side is established using JDBCOperations, and Java Spring Boot. Oracle is used as the database to handle the database create, read, update, and delete task.

3.2 Use Case Diagram

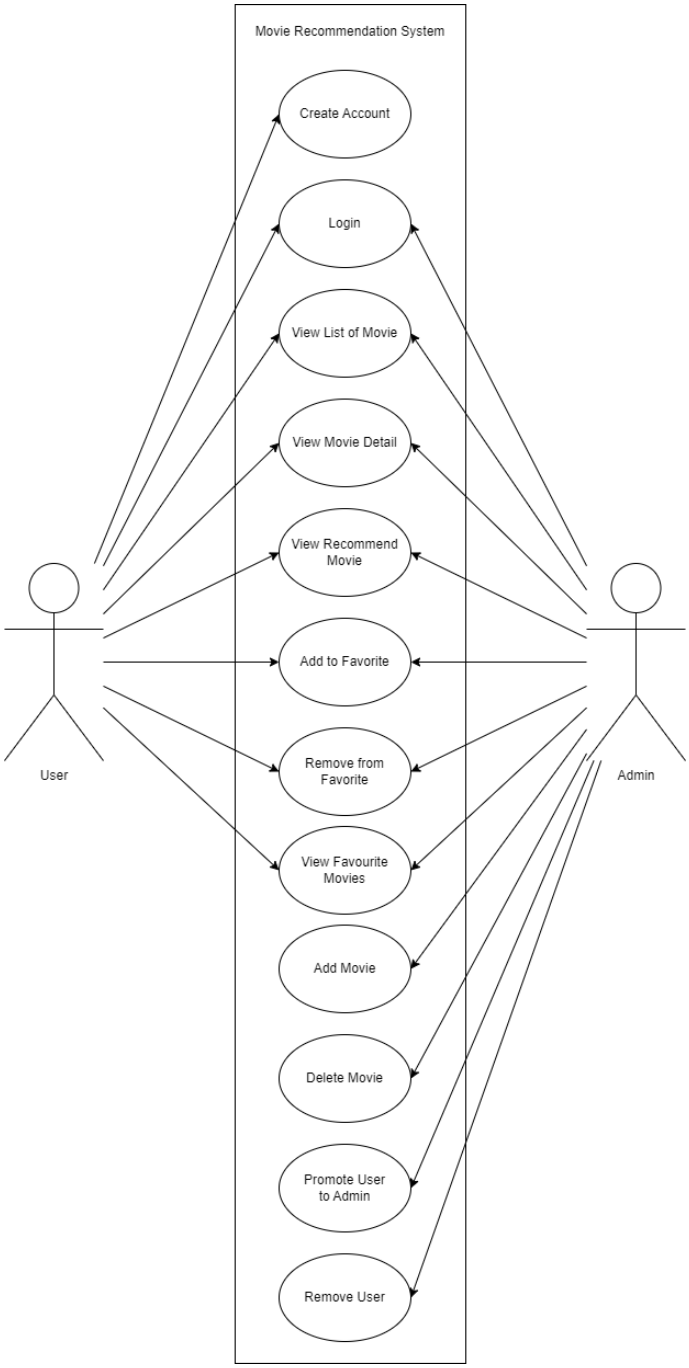


Figure 2: Use Case Diagram

Figure 2 shows the use case diagram of MRS. There are 12 use cases in the system with user have 8 use case excluding add movie, delete movie, promote user to admin, and remove user. While admin have 11 use case excluding only create account.

3.3 Flow Diagram

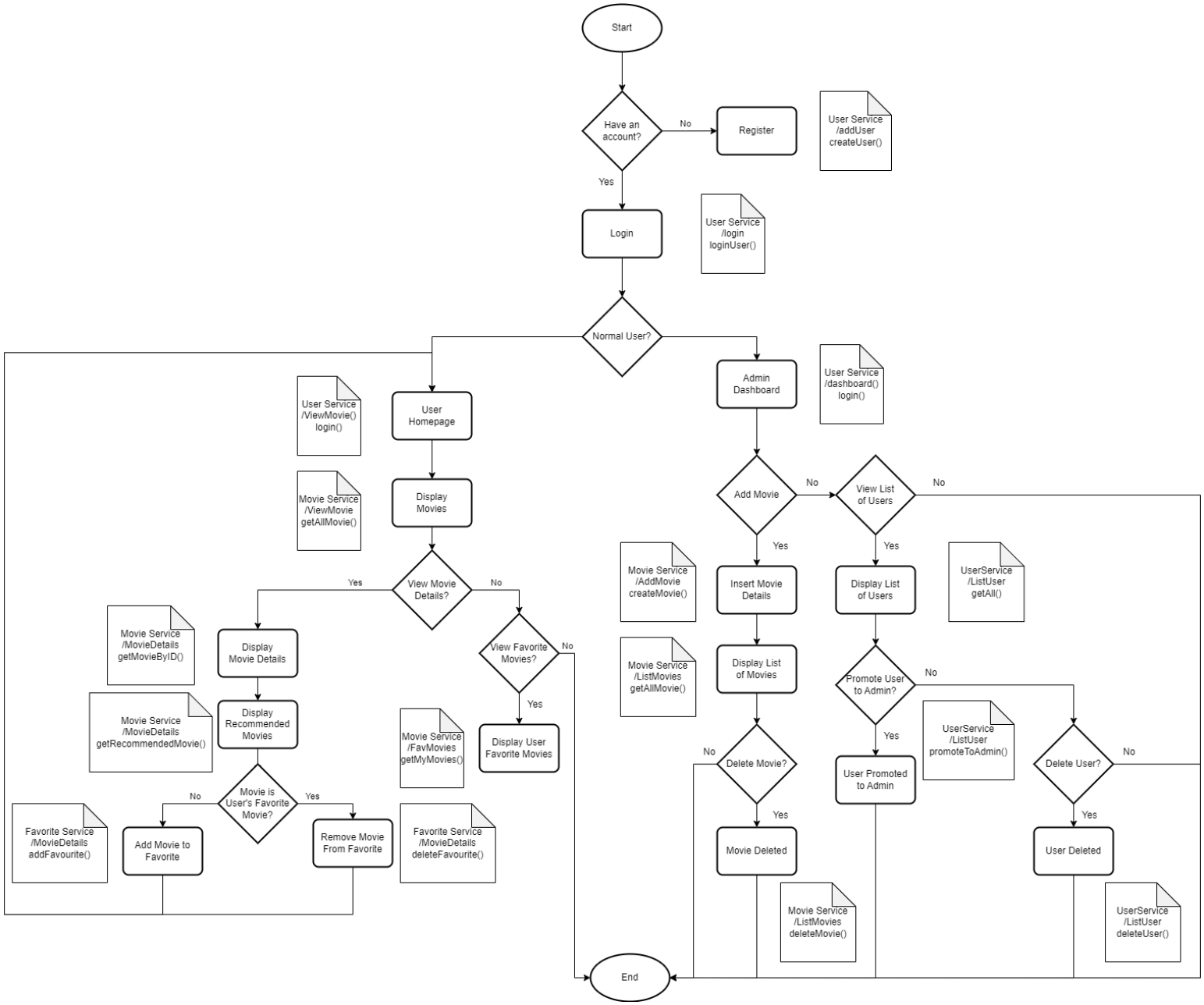


Figure 3: Flow Diagram for Whole System

3.4 Entity Relationship Diagram

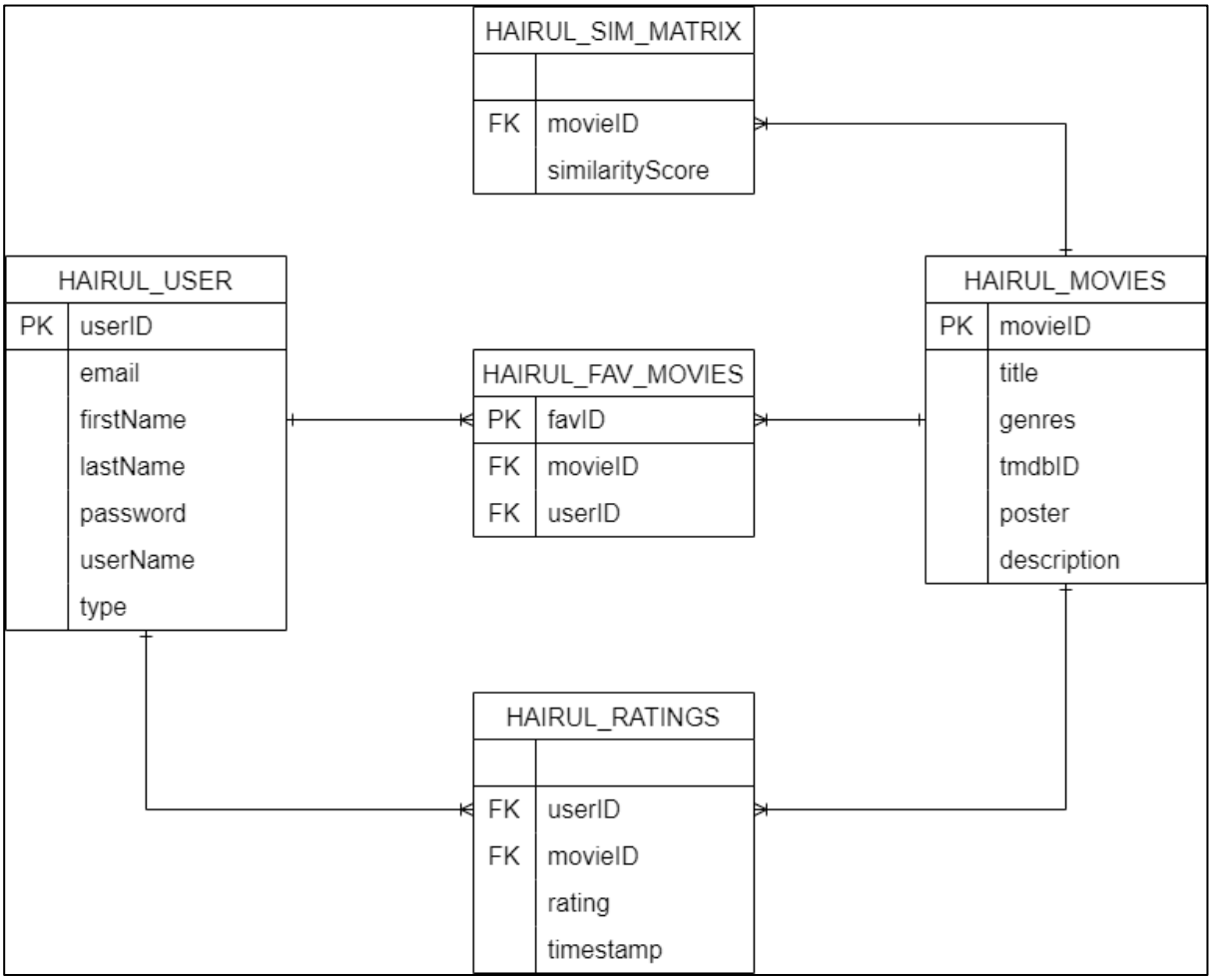


Figure 4: Entity Relationship Diagram of MRS

Figure 4 shows the entity relationship diagram of the Movie Recommender System which shows the connection between each table in the database.

These are the relationship that exist in the table:

- One user can have one or many favourite movies.
- One movie can be in one or many users' favourite movies list.
- One user can give one or many ratings.
- One movie can receive one or many ratings by the user.
- One movie can be in many similarity matrix calculations.

3.5 Use Case Description

3.5.1 UC 001 – Login

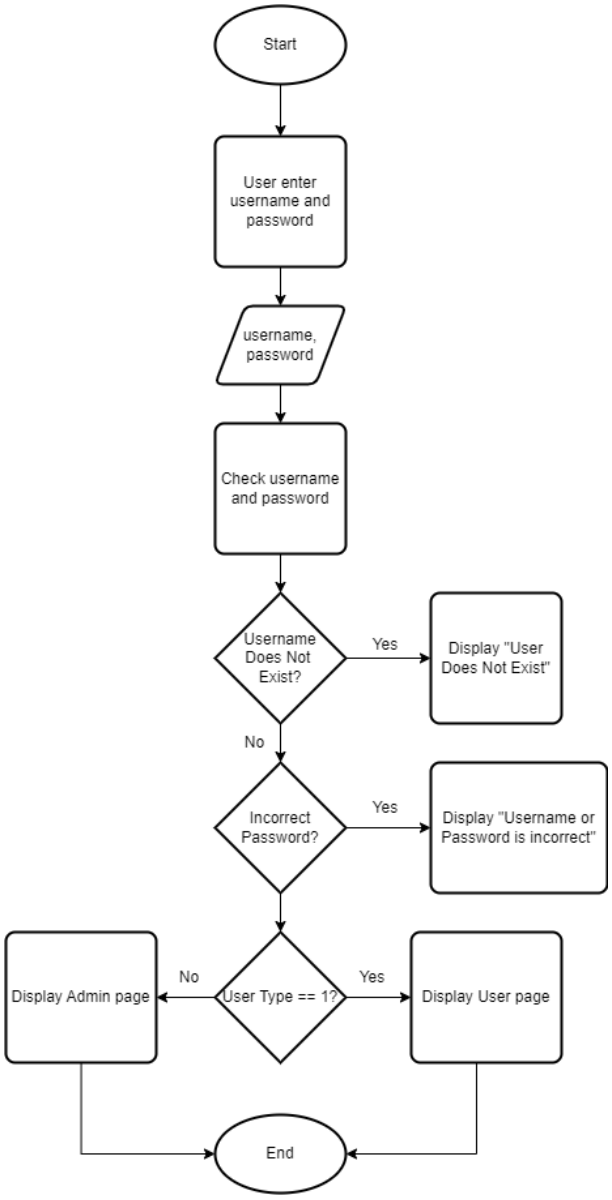


Figure 5: Login Flow

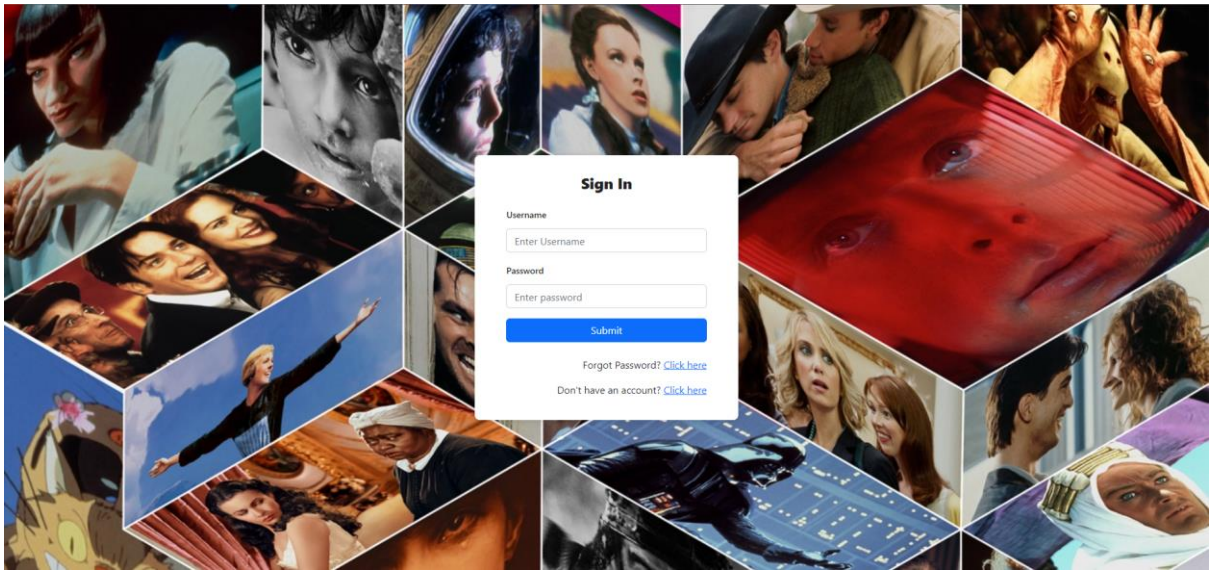


Figure 6: Index Page (Login Form)

Figure 5 shows the login flow for the login function. The process starts with user enters the login page which is shown in Figure 6. User then have to enters their username and password to proceed with the login process. If a user enter the wrong username or password, an error message will be shown to the user. Figure 7 below shows the error message user received.

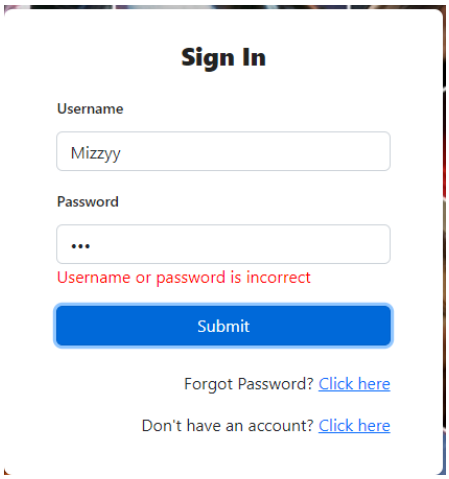


Figure 7: Error Message

After a user has successfully enters the valid username and password, they will be redirected to a page based on the type of their account. If they are an admin, they will go to the admin dashboard, and if they are a normal user, they will go straight to the movie list page.

```
//Sending the user object to the API call and getting the result
await UserService.login(user).then((res) => {

  //If successfull show an alert saying login successfull
  alert('Login Successful')
  const data = res.data //Getting the result from the api
  console.log(res.data)

  //Checking if the user type is 1 redirect to admin page
  if(data.type == "1")
  {
    window.location.href ="/Dashboard";

    //Storing the current user data into session
    localStorage.setItem('id', data.userid)
    localStorage.setItem('username', data.username)
  }
  //Checking if the user type is 2 redirect to homepage which is ViewMovie
  else
    window.location.href ="/ViewMovie";

    //Storing the current user data into session
    localStorage.setItem('id', data.userid)
    localStorage.setItem('username', data.username)
```

Figure 8: Sending User's username and password to API from the Client-Side

```
@PostMapping("/login") //Method call to log in
public ResponseEntity<?> login(@RequestBody User user)
{
    return serviceU.loginUser(user);
} //Pass data from the form into a user object
```

Figure 9: Server-side API for Login

Based on Figure 8, the client side will take the username and password then store it in a user object for it to be pass to the server side using the API call. The API then will return the result received from the server-side based on Figure 9. Back in the client-side it will then check if the user type is 1 (User) or 2 (Admin). If there is an error, the API will return the error message and the client-side will display the message to the user.

3.5.2 UC 002 – Register

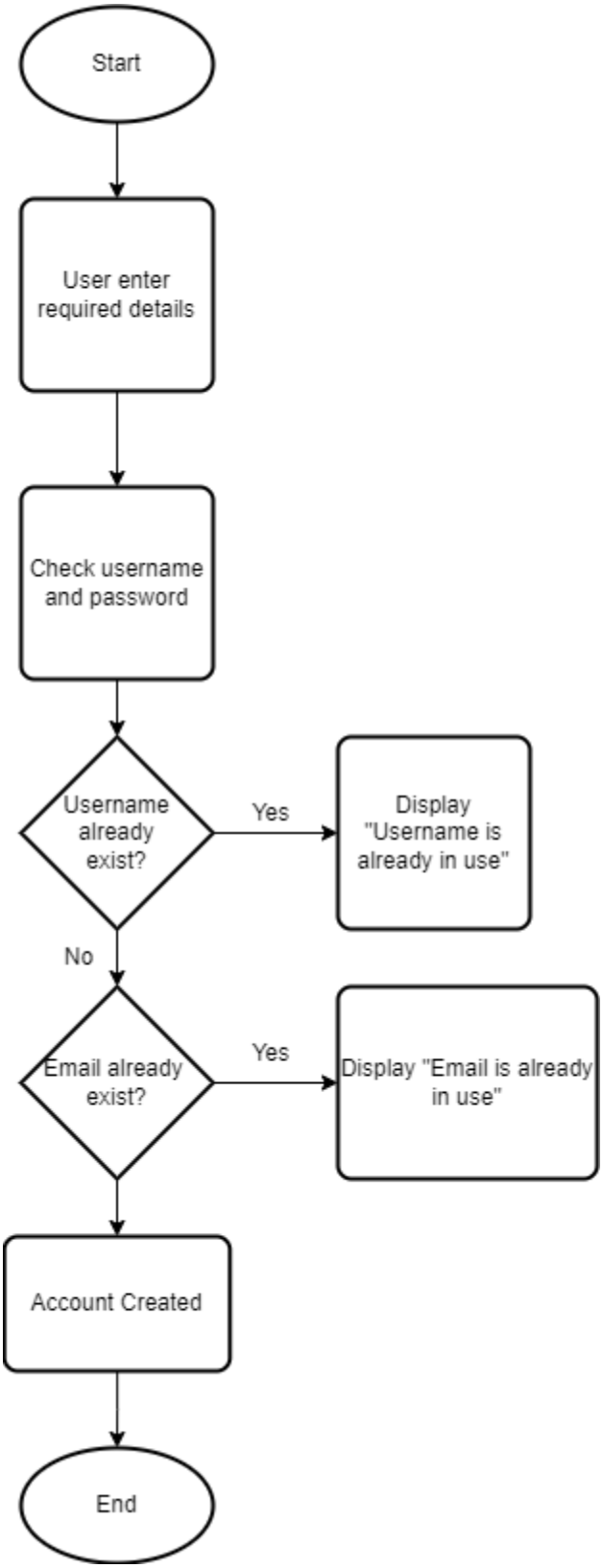


Figure 10: Register Flow

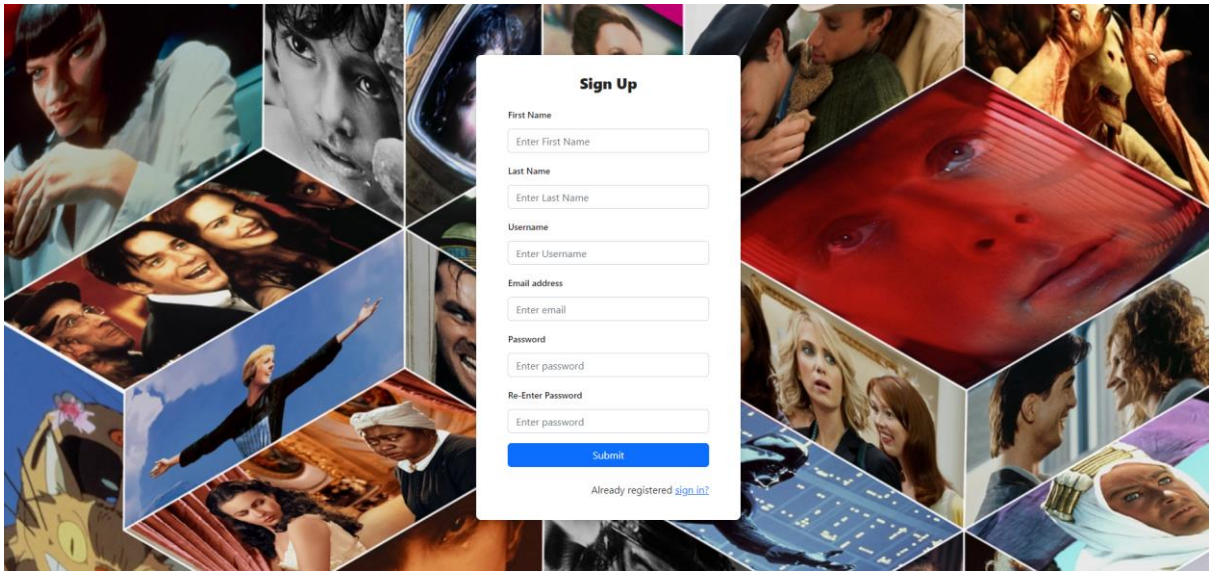


Figure 11: Register Page

Figure 10 shows the register flow for the register function. The process starts with user enters the register page which is shown in Figure 11. User then have to enter the required details to proceed with the register process. If a user enters an existing username or email, an error message will be shown to the user. Figure 12 below shows the error message user received.

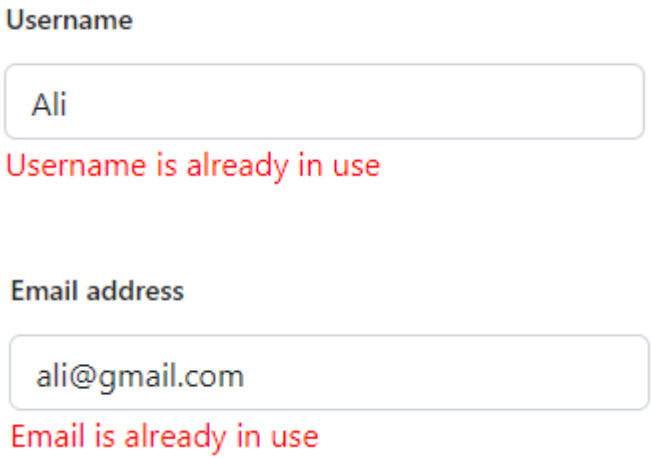


Figure 12: Error Message During Registration

When a user enters a valid username and password, a popup message will be shown saying “User Registered” and they will be redirected back to the login page.

```
const user = {
    username,
    email,
    firstname,
    lastname,
    password
}

//Checking if password 1 and password 2 from the form match
if(password === password2)
{
    //If match, send the user object to the API call
    await UserService.register(user).then((res) =>{

        alert('Account Registered')
        window.location.href ="/Login";

      //Catching error
    }).catch(error =>{

        //Checking if the error message is email
        if(error.response.data == "Email is already in use")
          setErrEmail(true)
        //Checking if the error message is username
        else if(error.response.data == "Username is already exist")
          setErrUser(true)

        console.log(error)
    })
}
```

Figure 13: Sending User Details to the API from Client-Side

```
@PostMapping("/addUser")  //Method call to add a new user in the database
public ResponseEntity<?> addUser(@RequestBody User user){
    return serviceU.createUser(user);
} //Pass user object into the service
```

Figure 14: Server-Side API for Register

Based on Figure 13, the user details are stored in a user object. The system first check if the password entered matched is each other and will proceed to send the user object to the API call. In Figure 14, the API will receive the object from the client-side and send a response to the client-side. If there is an error, the API will return the error message and the client-side will display the message to the user.

3.5.3 UC 003 – View List of Movies

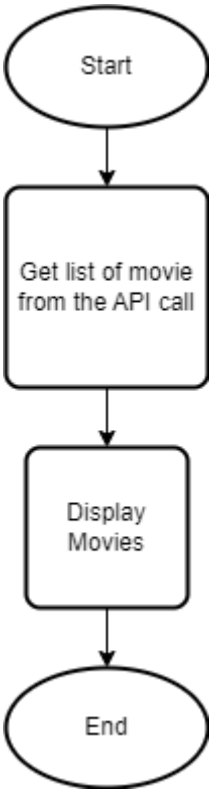


Figure 15: View Movies Flow

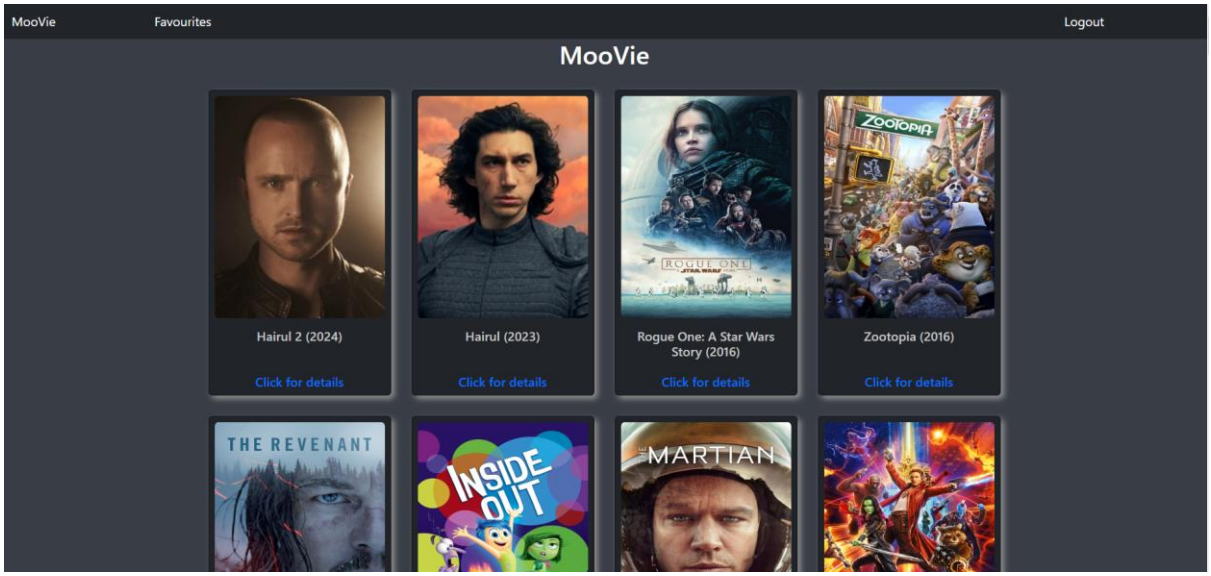


Figure 16: View Movies Page

Figure 15 shows the flow to view the list of movie page. Upon entering the movie page, the page will load for a while and then display all the movie available from the database. Figure 16 shows the page of view movies. When a user clicks on the movie, they will be redirected to the movie details page.

```
//Getting all the movie from the API call
MovieService.getAllMovie().then((res) => {

    //Inserting the result to movies array and set idLoading to false
    this.setState({ movies: res.data, isLoading: false });

    //Looping through each movie
    this.state.movies.forEach((movie) => {

    //URL to get the every movie API
      let URL = 'https://api.themoviedb.org/3/movie/' + movie.tmdbid + '?api_key=3fd2be6f0c70a2a598f084ddfb75487c&language=en-US';

      //After fetching the result from the url
      fetch(URL)
        .then((res) => res.json()) //Convert it into JSON
        .then((data) => {

        //Setting the poster of each movie
          this.setState(prevState => ({
            movies: prevState.movies.map(m => {

                //Check if the poster in the database is not empty
                if(m.poster != null){
                    return{
                        ...m,
                        poster: m.poster
                    }
                }
                //Check if the movie has a tmdbid  and use the data from the api
                else if (m.tmdbid === movie.tmdbid) {
                return {
                  ...m,
                  poster: data.poster_path ? this.state.IMG_PATH + data.poster_path : this.state.noImg
                };
              }
              return m;
```

Figure 17: Client-Side for Getting All Movie

```
@GetMapping("/getMovies")  //Method call to get all movies
public List<Movies> getAll() { return serviceM.getAll(); }
```

Figure 18: Server-Side API for Getting Movies

Based on Figure 17, the code first call the API to get data from every movies in the database. From the server-side in Figure 18, a response is sent to the client. Then from every result, the tmdbid of each movie is use and send it to the API of TMDB API to get the movie poster, banner, and description. The system then checks if the movie already has a poster or description stored in the database then they will not fetch the data from TMDB API.

3.5.4 UC 004 & UC 005 – View Movie Details & Recommended Movies

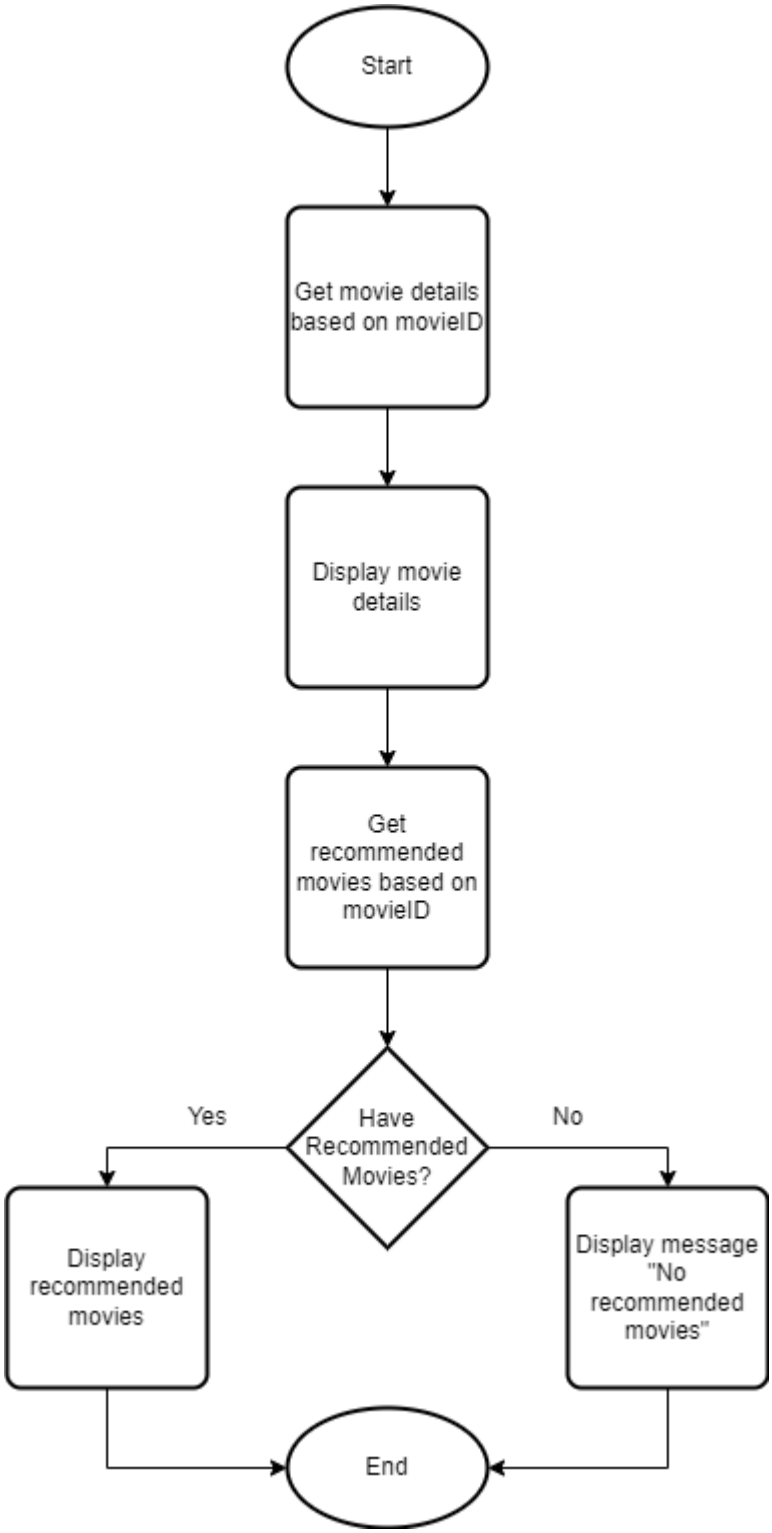


Figure 19: View Movie Details & Recommended Movies Flow

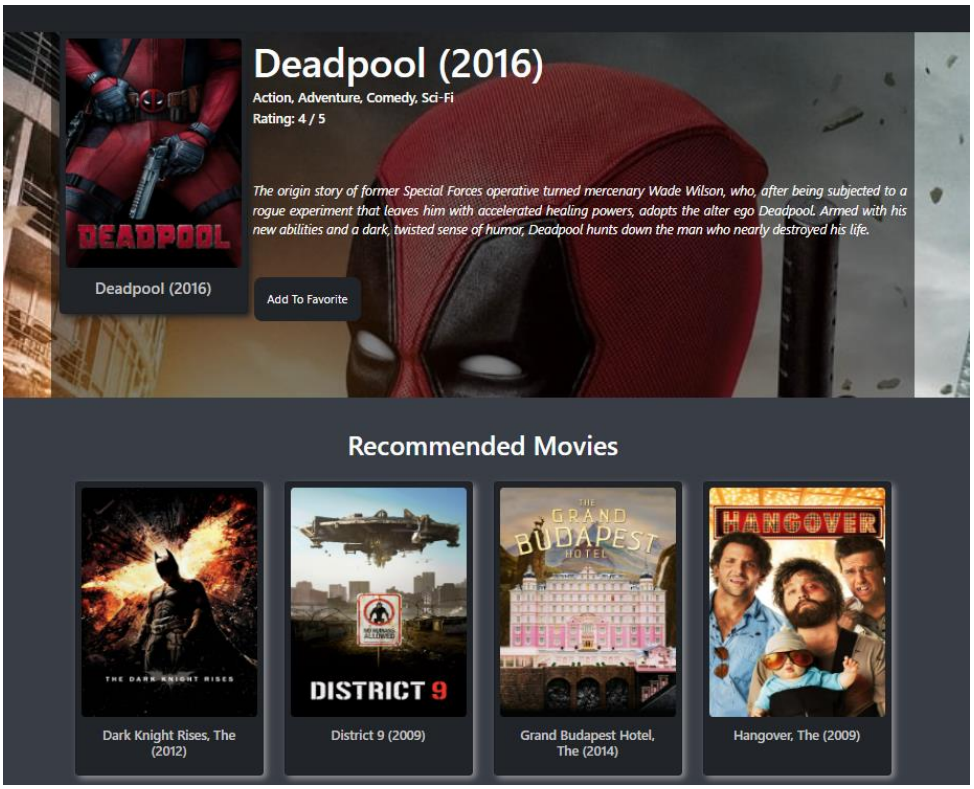


Figure 20: Movie Details & Recommended Movies Page

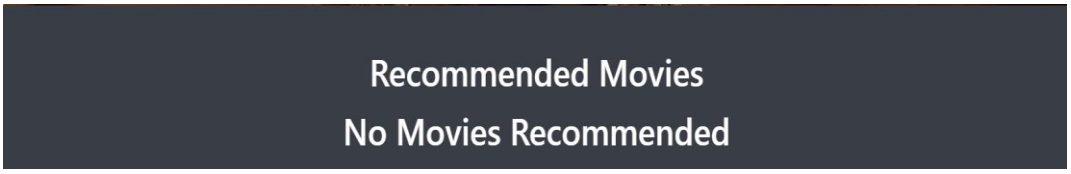


Figure 21: No Recommendation Message

Figure 19 shows the flow of view the movie details and recommended movies. The process starts with the system fetch all the data from the API call and show the details of the movie to the user. If the movie doesn't have a recommended movie, a message saying no movies recommended will be displayed.

```
no usages
@GetMapping("/getMyMovies/{id}") //Method call to get users fav movie by passing id
public List<Movies> getMyMovie(@PathVariable int id){return serviceM.getMyMovie(id);}
```

Figure 22: Server-Side of Get Movie Details

The movie details use the same code as shown in Figure 17, but instead of using MovieService.getMovie() this process uses MovieService.getMovieByID() with movieID as the passing parameter. The system then fetches the data receive from the server side in Figure 22.

```
MovieService.getRecommendedMovies(this.state.id).then((res) =>{

    this.setState({recMovies: res.data}); //Store movieid in recMovies

    //Looping through each movie
    this.state.recMovies.forEach((movie) => {

        //URL to get the current movie API
        let URL = 'https://api.themoviedb.org/3/movie/' + movie.tmdbid + '?api_key=3fd2be6f0c70a2a598f084ddfb75487c&language=en-US';

        //After fetching the result from the url
        fetch(URL)
          .then((res) => res.json()) //Convert it into JSON
          .then((data) => {

            //Setting the poster of the movie
            this.setState(prevState => ({
              recMovies: prevState.recMovies.map(m => {

                //Check if the poster in the database is not empty
                  if(m.poster != null){
                      return{
                          ...m,
                          poster: m.poster
                      }
                  }
                //Check if the movie has a tmdbid  and use the data from the api
                  else if (m.tmdbid === movie.tmdbid) {
                  return {
                    ...m,
                    poster: data.poster_path ? this.state.IMG_PATH + data.poster_path : this.state.noImg
                  };
                }
                return m;
              })
            }));
        });
    });
```

Figure 23: Client-Side Get Recommended Movies

```
@GetMapping("/movies/{id}") //Get calling method with passing id in url
public ResponseEntity<?> getSimilarMovie(@PathVariable int id)
{
    return serviceM.getSimilarMovie(id);
} //Use pathVariable to access id in url
```

Figure 24: Server-Side Get Recommended Movies

Figure 23 shows how the system get the recommended movies. The system pass the movie ID to the API call to get movies recommended based on current movie view. In the server-side, the system will take the movieID and find the closest movie to it using the recommendation matrix table. Then the system will return the movies recommended to the API.

3.5.5 UC 006 & UC 007 – Add to Favourite & Remove from Favourite

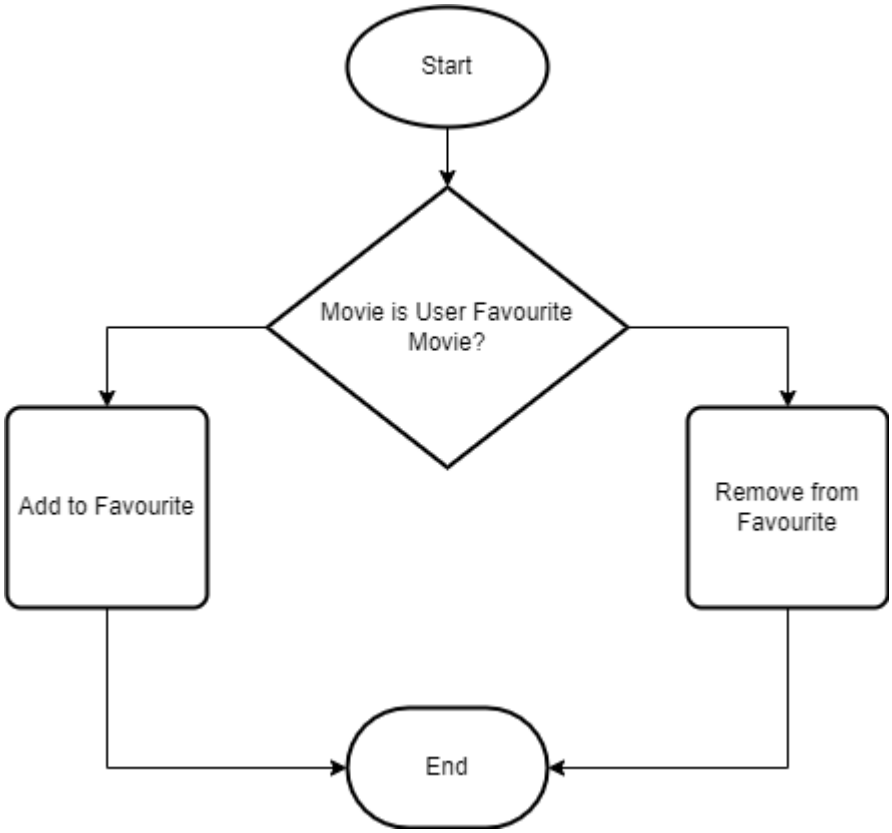


Figure 25: Add & Remove Favourite Flow

Figure 26: Add to Favourite Button

Figure 27: Remove from Favourite Button

Based on Figure 25, the flow of this process starts by the system checking the movie favourite status. If the movie is not in the user's favourite list, the add to favourite button from Figure 26 will be shown, else the remove from favourite button from Figure 27 will be shown.

```
//Sending current user id to the API call
FavouriteService.getMyFav(this.state.curUserID).then((res) =>{
    this.setState({favMovie: res.data}) //Get fav movie result

    //Checking if the current movie is in the user fav movie list
    this.state.favMovie.forEach((fav) =>{

        if(fav.movieID == this.state.id)
            this.setState({favState: true}) //Setting the state of favourite to true

    });
});
```

Figure 28: Client-Side for Getting Movie Favourite Status

```
//Add Movie to Favourite Function and passing movieid into the function
addFavourite(idMovie){

    //Creating a favMovie object containing userID and movieID
    const favMovie = {
        userID: this.state.curUserID,
        movieID: idMovie
    }

    //Sending the favMovie object to the API
    FavouriteService.addToFav(favMovie).then((res) =>{

        alert('Movie Added To Favorite')
        this.setState({favState: true}) //Setting favState to true
    })

}

//Remove Movie from Favourite Function and passing movieid into the function
deleteFavourite(idMovie){

    //Creating a favMovie object containing userID and movieID
    const favMovie = {
        userID: this.state.curUserID,
        movieID: idMovie
    }

    //Sending the favMovie object to the API
    FavouriteService.removeFav(favMovie).then((res) => {

        alert('Movie Remove From Favorite')
        this.setState({favState: null}) //Setting favState to null
    })

}
```

Figure 29: Add to Favourite & Remove from Favourite Function

```
@PostMapping("/addFavourite") //Post calling method name
public String addFav(@RequestBody Favourite fav) { return service.addFav(fav); }

no usages
@PostMapping("/removeFavourite") //Post method to remove fav
public String removeFav (@RequestBody Favourite fav) { return service.deleteFav(fav); }

no usages
@GetMapping("/getFavourite/{id}") //Get calling method with passing id in url
public List<Favourite> getMyFav(@PathVariable int id){return service.getByID(id);} //Use
```

Figure 30: Server-Side for Add & Remove Favourite

Figure 28 shows how the system check the favourite status of the movie. The process sends the user ID and movie ID to the API call. The API in the server-side in Figure 30 will then send the result and then the client-side will check the status. If the status is true, the system will use the addFavourite function in Figure 29 and allow user to add the movie to favourite. Else, the system uses the deleteFavourite function in Figure 29 and allow the user to remove the movie from his favourite list.

3.5.6 UC 008 – View Favourite Movies

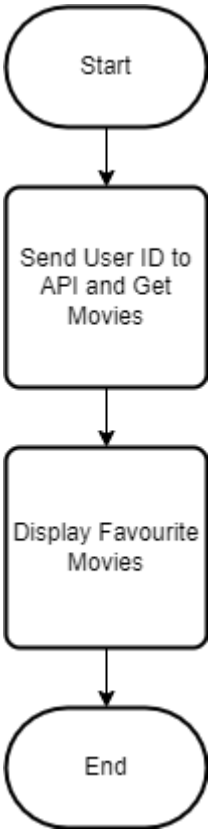


Figure 31: View Favourite Movies Flow

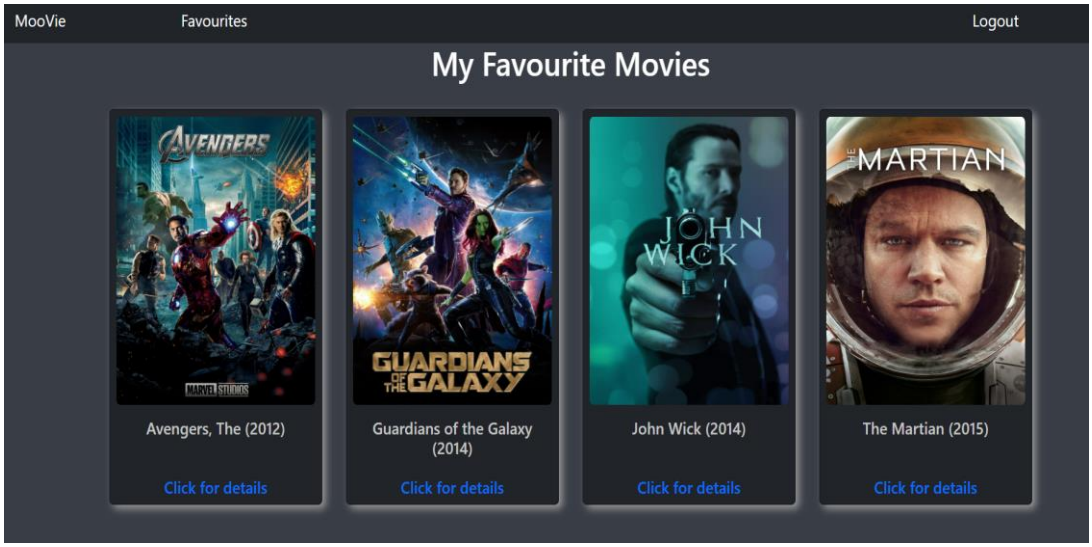


Figure 32: View Favourite Movies Page

Based on Figure 31, view favourite movies process starts by the system sending the user ID to the database and then display the list of movies based on the user ID.

```
componentDidMount(){

    //Getting the fav movies by passing current user id and store them in
    MovieService.getMyMovies(this.state.curUserID).then((res) => {
        this.setState({ movies: res.data, isLoading: false }); //Storing

        //Going to each movies api and fetching data from there
        this.state.movies.forEach((movie) => {
          let URL = 'https://api.themoviedb.org/3/movie/' + movie.tmdbid

          //After calling each api we get the poster for the movies
          fetch(URL)
            .then((res) => res.json())
            .then((data) => {
                //This method to use to preserve the object movies variab
              this.setState(prevState => ({
                movies: prevState.movies.map(m => {
                    //Checking if the movie has a poster link in the data
                    if(m.poster != null){
                        return{
                            ...m,
                            poster: m.poster
                        }
                    }
```

Figure 33: Client-Side View Favourite Movies

```
no usages
@GetMapping("/getMyMovies/{id}") //Method call to get users fav movie by passing id
public List<Movies> getMyMovie(@PathVariable int id){return serviceM.getMyMovie(id);}
```

Figure 34: Server-Side View Favourite Movies

Figure 33 shows the process of getting the user's favourite movies. The process is similar to getting all the movies from the database in UC 003 but in this case, it gets all the movies based on the user ID. The system passed the user ID to the favourite database and then returns the movie ID. It then displays the title and image of the movies with the ability to view the movie details.

3.5.7 UC 009 – Add Movie

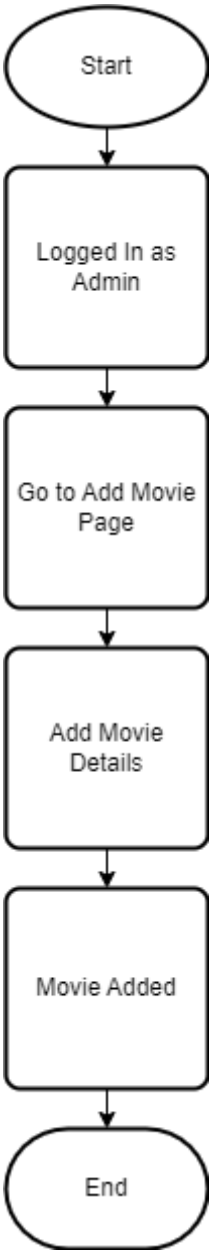


Figure 35: Add Movie Flow

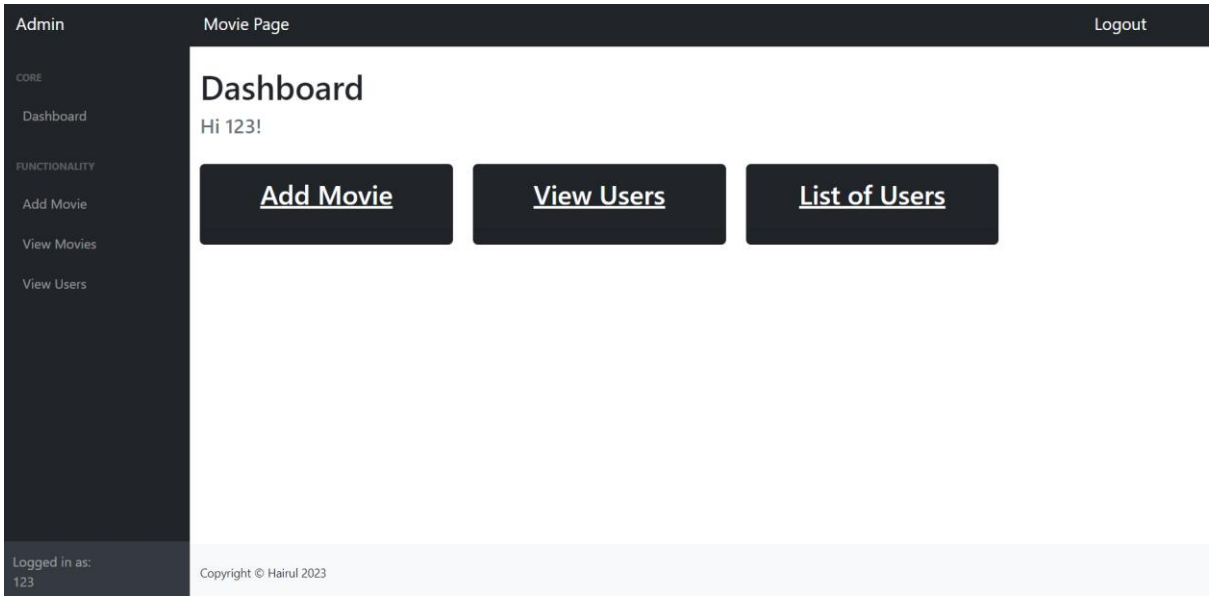


Figure 36: Admin Page

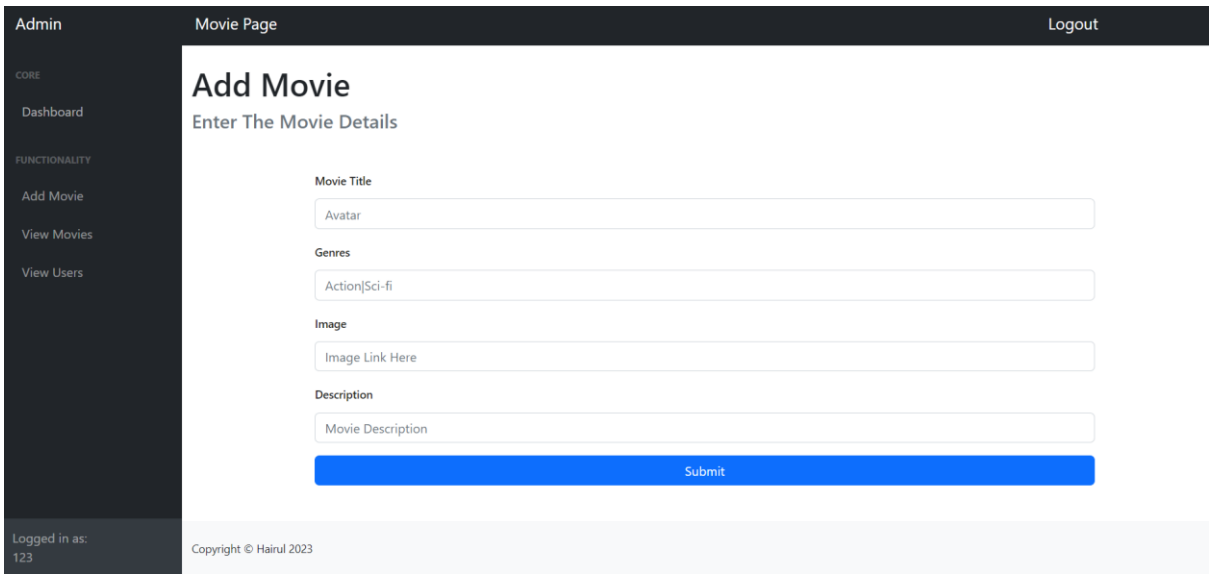


Figure 37: Add Movie Page

Based on Figure 35, the process of adding a movie starts by the user logging in as an admin. The user then will be redirected to admin page displayed in Figure 36. The admin page will display the current user's username. To add a movie the user can click on Add Movie button and they will be redirected to Add Movie page in Figure 37. User then need to enter the movie details and click submit. When the process is successful, a message will be displayed saying "Movie Successfully Added".

```
//Function to save movie when click submit button
const saveMovie = (e) =>{

    e.preventDefault(); //prevent page from loading on default

    const movie  = {title,genres,poster,description} //Creating a mov

    //Sending the movie object to axios post
    MovieService.createMovie(movie).then((response) => {

        alert('Movie Added')
        window.location.reload() //Reload the page to empty the form

    }).catch(error => {
        console.log(error)
    })
}
```

Figure 38: Client-Side Add Movie

```
no usages
@PostMapping("/addMovies") //Method call to add movies
public String addMovie(@RequestBody Movies movies){
    return serviceM.createMovie(movies);
} //Post data from body object
```

Figure 39: Server-Side Add Movie

Figure 38 shows the process of adding a movie to the database. The process starts by getting all the data from the form then store it in an object called movie. The object is then pass to the createMovie API call. In the server-side in Figure 39, the system takes the movie object from the API and then save it to the database.

3.5.8 UC 010 – Delete Movie

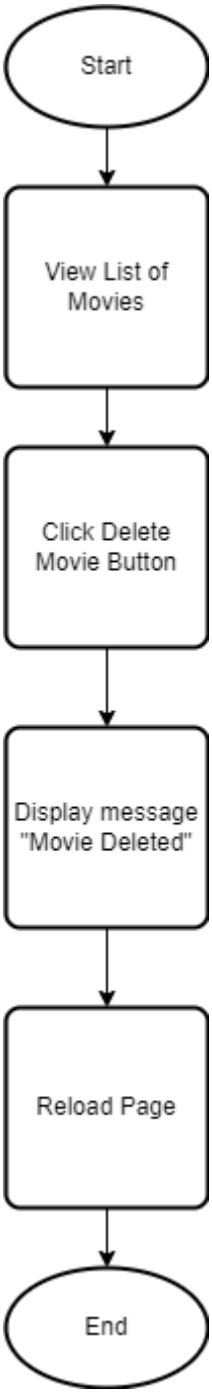


Figure 40: Delete Movie Flow

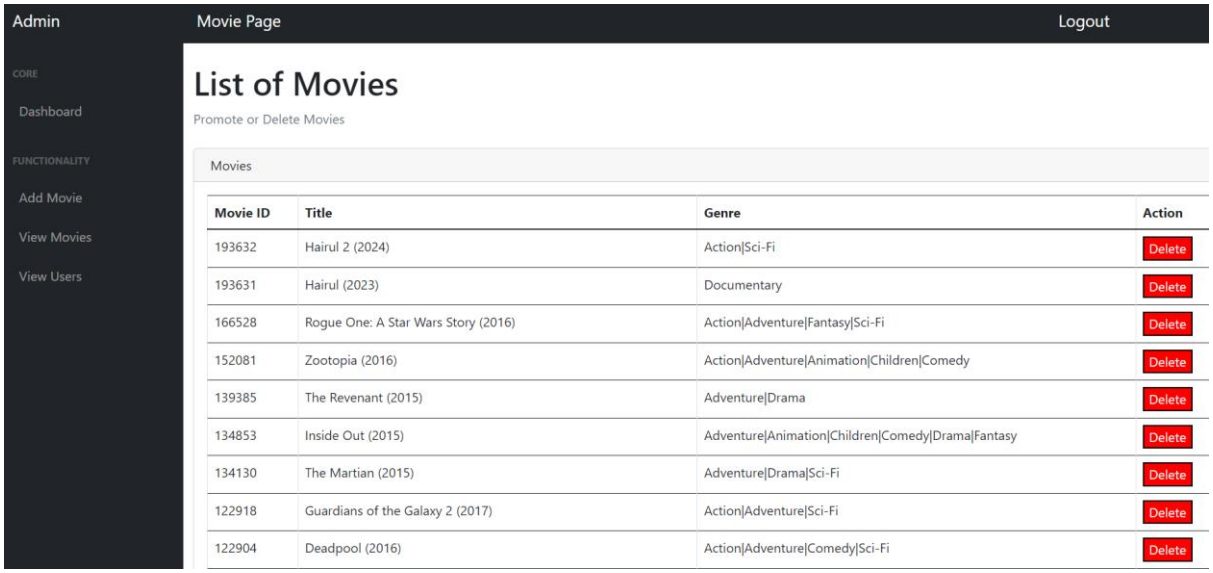


Figure 41: View and Delete Movie Page

Based on Figure 40, the process of deleting a movie starts with the user go to view list of movies page shown in Figure 41. In this page there is a button delete which let user to delete a movie of choosing. When the button is click, a message saying “Movie Delete” will be displayed and the page will be reloaded to show the update made.

```
    this.state = {
        movies: [], //Creating a movie array
        username: localStorage.getItem('username') //Getting th
    }

    this.deleteMovie = this.deleteMovie.bind(this); //To bind t
}

//Function delete movie by passing movie object into it
deleteMovie(movie){

    //Api call to delete movie, then if success display the ale
    MovieService.deleteMovie(movie).then(() =>{
        alert('Movie Deleted')
        window.location.reload();

        //Catching error and display it in the console
    }).catch(error =>{
            console.log(error)
    })

}

//On page load we run the api call to get all the movie and sav
componentDidMount(){
    MovieService.getAllMovie().then((res) => {
        this.setState({ movies: res.data});
    });
}
```

Figure 42: Client-Side Delete Movie

```
no usages
@PostMapping("/deleteMovie")  //Method call to delete the movie
public String deleteMovie(@RequestBody Movies movie){return serviceM.deleteMovie(movie);}
```

Figure 43: Server-Side Delete Movie

Figure 42 shows the process of deleting a movie. The process starts by the system fetch all the movie in the database and displayed it when the page load. When the delete button is clicked the deleteMovie function will run by sending the movieID to the API call. The API then will send the movieID to the server-side in Figure 43 and then the server will send back response to the client-side.

3.5.9 UC 011 & UC 012 – Promote User to Admin & Delete User

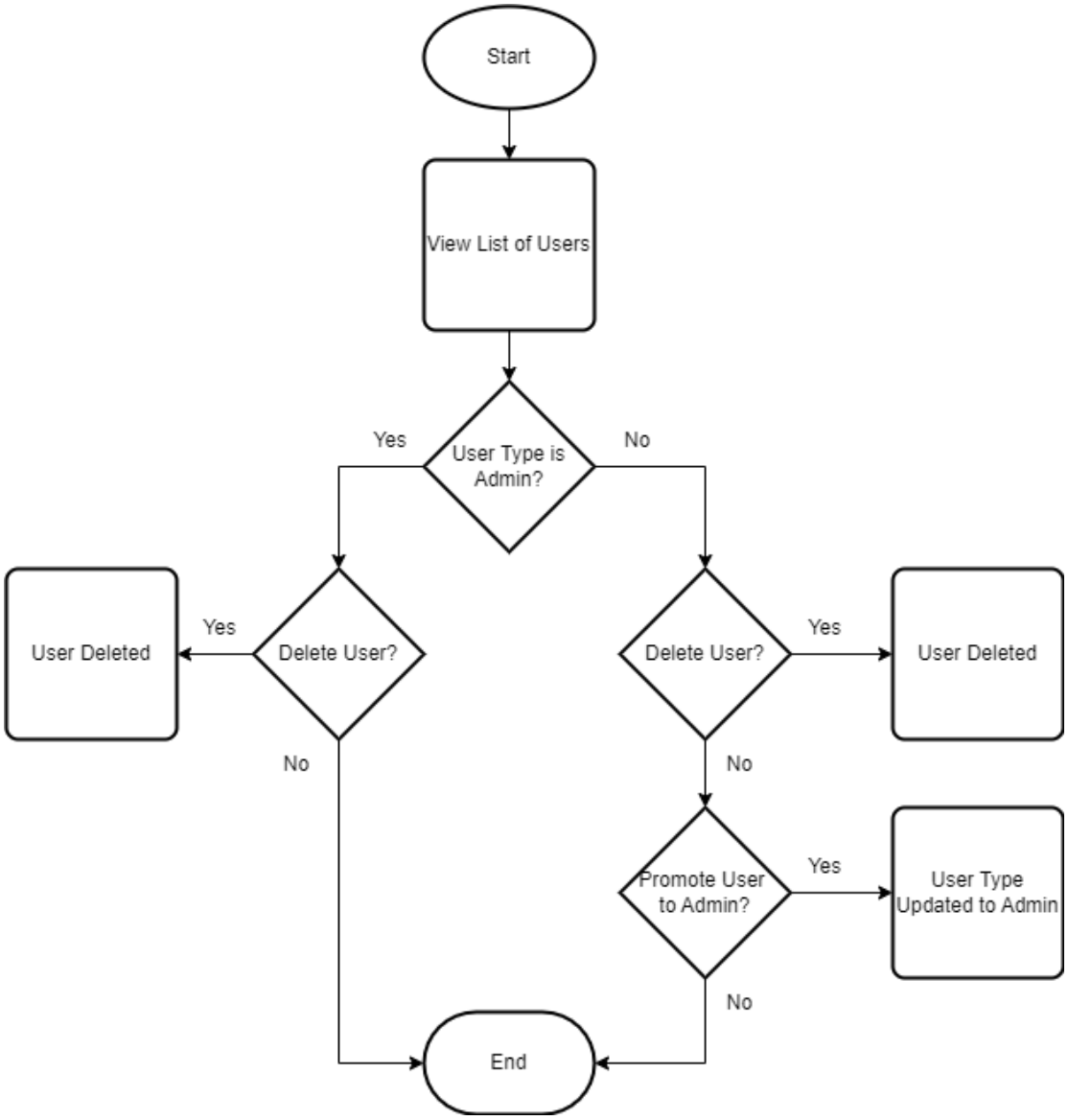


Figure 44: Promote User to Admin & Delete User Flow

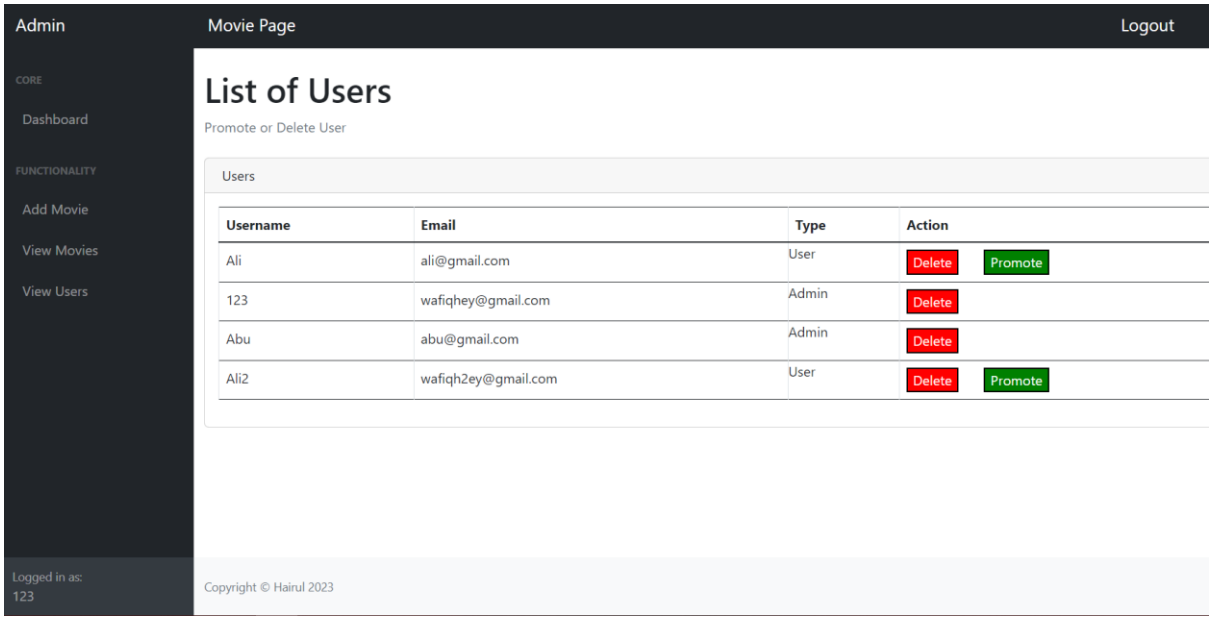


Figure 45: View List of Users Page

Based on Figure 44, the process of promoting a user to admin and delete a user starts with entering the view list of users page displayed in Figure 45. In this page, the system will show all the user exist in the database. If the user type is “USER” there will be a button Delete and Promote. Else, the button Delete is shown only. When the user click on the delete button, the user will be deleted and the page will reload to show changes made. If the user clicks on the promote button, the page will reload and show the user type has been changed to “ADMIN”.

```
//Function to promote a user to admin by passing the user
promoteToAdmin(user){

    //Api call to update the user to admin, show an alert
    UserService.updateToAdmin(user).then((res) =>{
        alert('User Updated')
        window.location.reload();

        //Catching error from the back end side
    }).catch(error =>{
            console.log(error)
    })
}

//Function delete user
deleteUser(user){

    //Api call to delete the user, show an alert that user
    UserService.delete(user).then((res) =>{
        alert('User Deleted')
        window.location.reload();

        //Catching error from the back end side
    }).catch(error =>{
            console.log(error)
    })
}

//Function to run when the page is load
componentDidMount(){

    //Getting all the user from API call and store it into
    UserService.getAll().then((res) => {
        this.setState({ users: res.data});
    });
}
```

Figure 46: Client-Side of the UC 011 & UC 012

```
no usages
@PostMapping("/updateToAdmin")  //Method call to update admin
public String updateToAdmin(@RequestBody User user){
    return serviceU.updateToAdmin((user));
} //Pass the user object into the service

no usages
@PostMapping("/deleteUser") //Method call to delete a user
public String deleteUser(@RequestBody User user){
    return serviceU.deleteUser((user));
} //Pass user object from the body into the service
```

Figure 47: Server-Side of UC 011 & UC 012

Figure 46 shows the process of promoting and deleting a user. The process starts by getting all the users from database using getAll API call. Then the server will send a response containing the list of users in the database. When user click on delete button, the deleteUser function will be use. The function will send the selected user ID to the API and the server will receive the ID and delete the user. When a user clicks on the promote button, the system will send the selected user ID to the API and the server will receive the ID and update the user type from 1 to 2.

3.6 Data Dictionary

3.6.1 Table HAIRUL_USER

Column ID	Column Name	Data Type	Field Length	Constraint
1	USERID	NUMBER	(38,0)	PK
2	EMAIL	VARCHAR	255	
3	FIRSTNAME	VARCHAR	255	
4	LASTNAME	VARCHAR	255	
5	PASSWORD	VARCHAR	255	
6	USERNAME	VARCHAR	255	
7	TYPE	VARCHAR	255	

Table 3: HAIRUL_USER

3.6.2 Table HAIRUL_MOVIES

Column ID	Column Name	Data Type	Field Length	Constraint
1	MOVIEID	NUMBER	(38,0)	PK
2	TITLE	VARCHAR	255	
3	GENRES	VARCHAR	255	
4	TMDBID	NUMBER	(38,0)	
5	POSTER	VARCHAR	255	
6	DESCRIPTION	VARCHAR	255	

Table 4: HAIRUL_MOVIES

3.6.3 Table HAIRUL_RATINGS

Column ID	Column Name	Data Type	Field Length	Constraint
1	USERID	NUMBER	(38,0)	FK
2	MOVIEID	NUMBER	(38,0)	FK
3	RATING	NUMBER	(38,0)	
4	TIMESTAMP	NUMBER	(38,0)	

Table 5: HAIRUL_RATINGS

3.6.4 Table HAIRUL_FAV_MOVIES

Column ID	Column Name	Data Type	Field Length	Constraint
1	FAVID	NUMBER	(38,0)	PK
2	MOVIEID	NUMBER	(38,0)	FK
3	MOVIEID	NUMBER	(38,0)	FK

Table 6: HAIRUL_FAV_MOVIES

4. Pearson’s Correlation

4.1 Description of Pearson’s Correlation

Pearson's correlation is a statistical method used to measure the strength and direction of the linear relationship between two continuous variables. In the context of movie recommendation, it can be used to determine the similarity between different movies based on their ratings by different users.

Based on user ratings, the movie similarity matrix mathematically illustrates how similar two films are to one another. This resemblance is measured by the Pearson's correlation coefficient, with a range of -1 to 1, with -1 indicating a perfect negative correlation, 0 indicating no correlation at all, and 1 denoting a perfect positive correlation.

To develop a movie similarity matrix based on user ratings, the following steps can be followed:

1. **Collect user ratings**: The first step of generating a similarity matrix is to collect all the user ratings data for every set of movies. The data can be obtained by using the HAIRUL_RATINGS table in the database.
2. **Create a ratings matrix**: Once all the data from users is ready, it can be transformed into a ratings matrix where the rows represent users and the columns represent movies. Each cell containing the rating that a user has given to a movie. **For example**: movie1, user1: 3.5 (rating).
3. **Calculate the Pearson's correlation coefficient:** By comparing the ratings of the same users for the two movies, it is possible to calculate the Pearson's correlation coefficient between them. The formula for calculating Pearson's correlation is as follows:

   $$r = \sum (x - \bar{x})(y - \bar{y}) / (n-1) \sqrt{(\sum (x - \bar{x})^2 (\sum (y - \bar{y})^2))}$$

   x and y are the ratings for the two movies, $\bar{x}$ and $\bar{y}$ are the means of the ratings, n is the number of users who have rated both movies, and r is the Pearson's correlation coefficient.

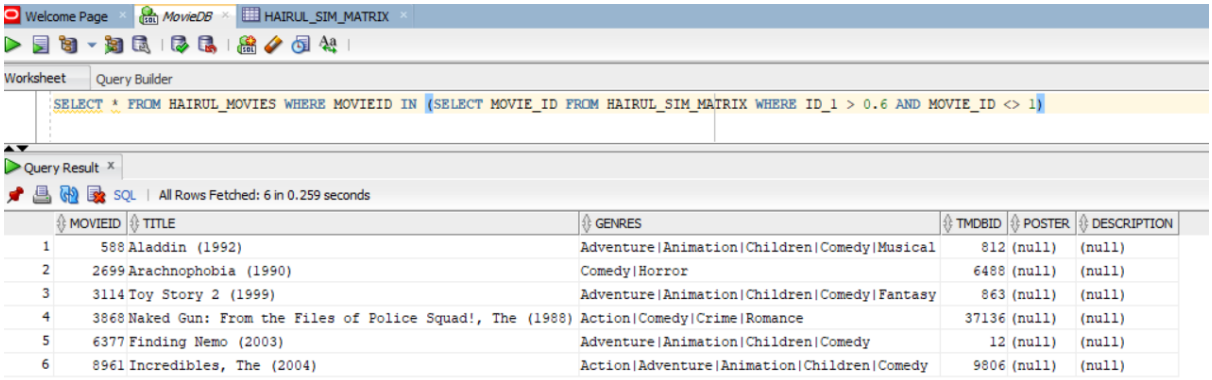
4. **Create and Store the similarity matrix:** Once the Pearson's correlation coefficient is calculated for all pairs of movies, a similarity matrix can be created, where each cell contains the similarity score between two movies.
   The matrix then will be stored in the table HAIRUL_SIM_MATRIX.

4.2 Example of Data in the Similarity Matrix

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	MOVIE_ID	ID_1	ID_2	ID_3	ID_5	ID_6	ID_7	ID_10	ID_11	ID_16	ID_17	ID_19	ID_21	ID_22	ID_24	ID_25
2	1	1	0.330978	0.487109	0.310971	0.106465	0.208402	-0.02141	0.206812	-0.16886	0.218245	0.325865	0.144401	0	0	0.207935
3	2	0.330978	1	0	0	0.16351	0	0.016626	0.466415	-0.31064	0	0.305796	0.112617	0	0	0
4	3	0.487109	0	1	0	0	0	0	0	0	0	0	0	0	0	0
5	5	0.310971	0	0	1	0	0	0	0	0	0	0	0	0	0	0
6	6	0.106465	0.16351	0	0	1	0	0.420222	0	0.524619	0	0.351882	0.047248	0	0	0.436732
7	7	0.208402	0	0	0	0	1	0	0	0	0	0	0	0	0	0
8	10	-0.02141	0.016626	0	0	0.420222	0	1	0.168661	0.384357	0	0.042841	0.042155	0		0.055526
9	11	0.206812	0.466415	0	0	0	0	0.168661	1	0	0	0	0.123299	0	0	0
10	16	-0.16886	-0.31064	0	0	0.524619	0	0.384357	0	1	0	0	-0.06473	0	0	0.307469
11	17	0.218245	0	0	0	0	0	0	0	0	1	0	0.293964	0	0	0.49529
12	19	0.325865	0.305796	0	0	0.351882	0	0.042841	0	0	0	1	0.131738	0	0	0
13	21	0.144401	0.112617	0	0	0.047248	0	0.042155	0.123299	-0.06473	0.293964	0.131738	1	0	0	0.24508
14	22	0	0	0	0		0	0	0	0	0	0		1	0	0
15	24	0	0	0	0	0	0	0	0	0	0	0		0	1	0
16	25	0.207935	0	0	0	0.436732	0	0.055526	0	0.307469	0.49529	0	0.24508	0	0	1

Figure 48: Movie Rating Similarity Matrix

Figure 48 shows the movie rating similarity matrix data in the database. For each movie, it will be compared the correlation between every other movie. 1 indicating perfect correlation, 0 indicates no correlation at all. In the movie recommendation system, the movie that has a similarity matrix rating higher than 0.6 will be selected and display to the user.



	MOVIEID	TITLE	GENRES	TMDBID	POSTER	DESCRIPTION
1	588	Aladdin (1992)	Adventure|Animation|Children|Comedy|Musical	812	(null)	(null)
2	2699	Arachnophobia (1990)	Comedy|Horror	6488	(null)	(null)
3	3114	Toy Story 2 (1999)	Adventure|Animation|Children|Comedy|Fantasy	863	(null)	(null)
4	3868	Naked Gun: From the Files of Police Squad!, The (1988)	Action|Comedy|Crime|Romance	37136	(null)	(null)
5	6377	Finding Nemo (2003)	Adventure|Animation|Children|Comedy	12	(null)	(null)
6	8961	Incredibles, The (2004)	Action|Adventure|Animation|Children|Comedy	9806	(null)	(null)

Figure 49: Using the Matrix to Get Similar Movies

Figure 49 shows how the recommended movies are retrieved from the table. By using the queries in Figure 49, we find the movie that has the closest similarity matrix to movieID 1 which is Toy Story. We get the results of Toy Story 2, Finding Nemo, The Incredibles, and more.