Software Requirement Specification

February 2, 2023

Capstone Project for Java Recommended System

Adlina Binti Abdul Kadir

Table of Contents

Overview of the project

The project is on a Movie Recommendation System. The system would recommend similar related movie to the one that the user has choose by using Item Based Collaborative Filtering. The front end development is done using HTML5,CSS and Bootstrap and the database use is DB2.

Feature list

1.0   Feature List

These are the list of features that is in the system.

Validation
- Login Validation
- Registration Validation

Admin
- Add Movie
- Add User
- View User

User
- View Movie
- View Recommended Movie
- Add Favourite
- Give Rating

1.1   Validation
1.1.1   Login Validation
1.1.1.1 User does not exist

This validation occurs when the user try to login with a username that does not exist in the database.

Upon clicking the login button, the system will first find the username enter by the user in the database.

```
// Find user by username
String name = user.getName(); //get the username enter by user
HttpHeaders headers = new HttpHeaders();
HttpEntity<User> entity = new HttpEntity<>(headers);
User find = restTemplate.exchange( url: "http://localhost:8085/findbyname/"+name, HttpMethod.GET, entity, User.class).getBody(); // find the username in database
Optional<User> userdata = Optional.ofNullable(find);
```

Figure 1.1: Find username in user table

As shown in Figure 1.1, the username that has been input by the user is fetch using get and assign to String name. Then, String name added as part of http://localhost:8085/findbyname/ that will use user API in user service to find the username in user table.

```
if(userdata.isPresent()) //if username exist
{
```

Figure 1.2: If user exist

```
else
{
    model.addAttribute( attributeName: "invalid",  attributeValue: "USER NOT EXIST!"); //return message user not exist
    return "index"; //redirect to index
}
```
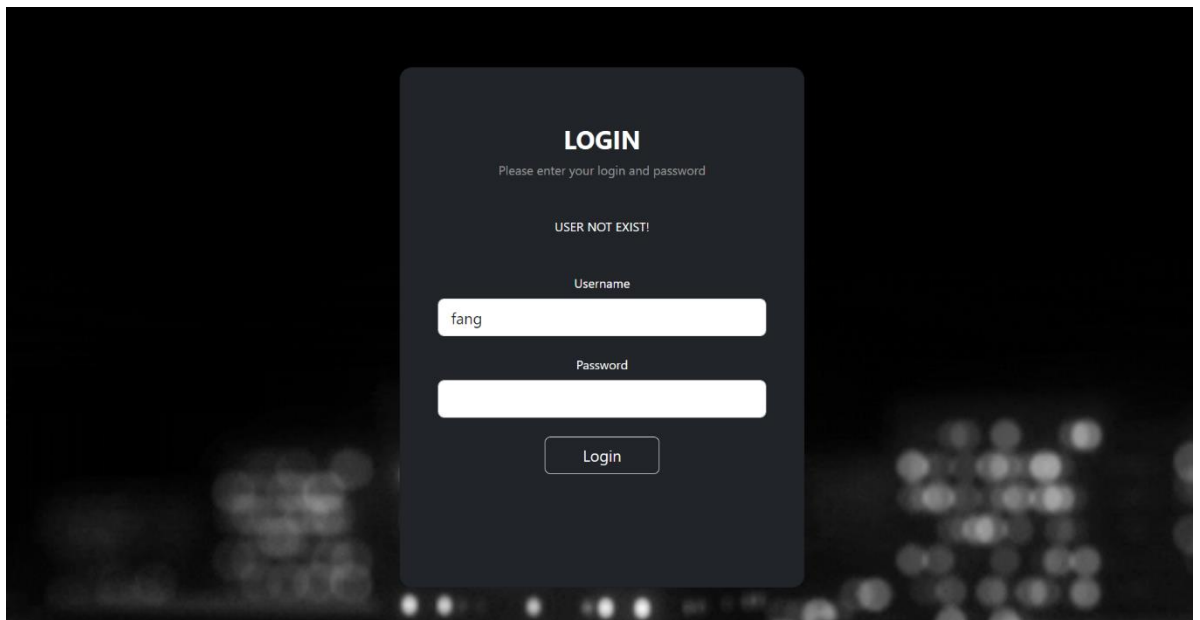
Figure 1.3: If the user not exist

Figure 1.4: User Not Exist message

The result of the code in Figure 1.1 is then use in if else where if the data with String name is present, it will proceed with the next code as shown in Figure 1.2 while if the user does not exist, it will return message of user not exist and redirect back to the index page.

1.1.1.2 Password Match

This validation occurs when the user tries to login but with the wrong password. This validation is check after the validation that data with username input by user exist.

```
//encrypted password
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder( strength: 14);
```

Figure1.5:BCrypt Encoder

For password validation, first a BCrypt Encoder object is created.

```
if (encoder.matches(user.getPassword(), userdata.get().getPassword())) //if password enter match with database
{
```

Figure1.6:If password match

Then the encoder is user to check whether the password input by user match the password in user table. userdata from Figure 1.1 is use to get the password of the user from the user table in database. Password that user enter is fetch using user.getPassword(). Then the encoder match this two data. If the match returns true then it will proceed with the next code.

```
else
{
    model.addAttribute( attributeName: "invalid", attributeValue: "LOGIN INVALID!"); //return message login invalid
    return "index"; //redirect to index
}
```

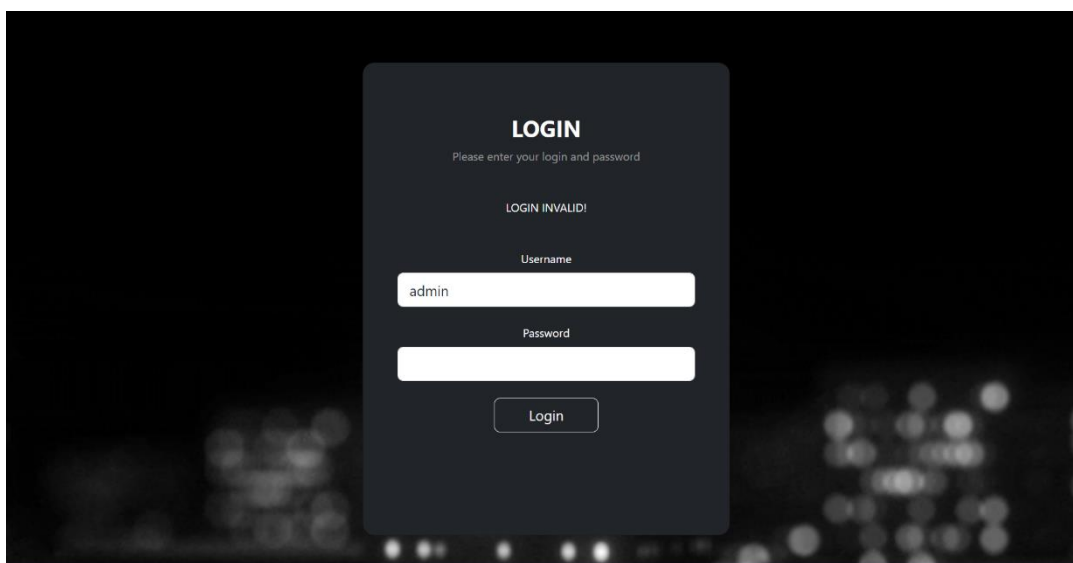Figure 1.7: If password not match

Figure 1.8: Login Invalid Message

If the encoder match returns false, then it will return the message login invalid and return to index page.

1.1.1.3 User position

This validation occurs after the password match return true. This is use to check whether the user is an Admin or not.

```
if (userdata.get().getPosition().equals("Admin")) //check the position type of use
{
    session.setAttribute( s: "username", user.getName()); //session with name
    return "homeadmin"; //redirect to home for admin
}
```

Figure 1.9: If position is Admin

Using userdata from Figure 1.1 to get the position of the user from the database, the data is then compare as shown in Figure 1.9. If the position of the user is Admin, the session for username starts and the user will be redirect to home page for admin.

```
session.setAttribute( s: "userid", userdata.get().getUser_Id()); //session for id
session.setAttribute( s: "username", user.getName()); //session for username

//get the list of all movie in database
HttpHeaders headers3 = new HttpHeaders();
headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
HttpEntity<Movie> entity3 = new HttpEntity<Movie>(headers3);
List<Movie> dataList = restTemplate.exchange( url: "http://localhost:8083/movies", HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>() {}).getBody();
model.addAttribute( attributeName: "dataList", dataList);
return "home"; // redirect to home for user
```

Figure 1.10: If position is not Admin

If the user position is not equal to Admin, then the user will be redirect to home page. Session for user id and username start and before redirecting to home page, list of movies from movie table is called using movie API.

1.1.2 Registration Validation

1.1.2.1 User Exist

This validation is created to check is the username to be added has already exist or not. This is to avoid duplicating user that could cause an issue when trying to login into the correct account.

```
String name = user.getName(); // get username input
String email = user.getEmail(); // get email input
String position = user.getPosition(); // get position input
int id = user.getUser_Id(); // get id

/*Encrypt the Password*/
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder( strength: 14);
var myPassword = user.getPassword(); // get password input
var encodedPassword = encoder.encode(myPassword); //encrypt the password
```

Figure 1.11: Get data input

When the user click register, the data that has been enter is fetch. For password, using BCryptPasswordEncoder, the password is encrypted and this encrypted version will be inserted into the database.

```
/*Find data with String name*/
HttpHeaders headers = new HttpHeaders();
HttpEntity<User> entitycheck = new HttpEntity<User>(headers);
User find = restTemplate.exchange( url: "http://localhost:8085/findbyname/"+name, HttpMethod.GET, entitycheck, User.class).getBody();
Optional<User> userdata = Optional.ofNullable(find);
```

Figure 1.12: Find data with String name

Using String name from Figure 1.9, data with similar input as String name is search in user table.

```
if (userdata.isPresent()) // if the data for String name exist
{
    model.addAttribute( attributeName: "exist", attributeValue: "USERNAME ALREADY EXIST!"); // return message username exist
    return "register"; // redirect to register page
}
```

Figure 1.13: If user exist

Figure 1.12: Username Exist Message

Then, the data from Figure 1.11 is use in an if else statement where if userdata.isPresent() is true, it will be redirect to register page and the message username already exist will appear

7

```
else
{
    user = new User(id, name, encodedPassword, email, position); // get all the input data
    HttpEntity<User> entity = new HttpEntity<>(user);
    restTemplate.exchange( url: "http://localhost:8085/register", HttpMethod.POST, entity, User.class).getBody(); // add data to user table
    return "useradmin"; // redirect to useradmin page
}
```

Figure 1.13: If user does not exist

If the user data does not exist in user table, the data is then added to user table and redirect to user admin page.

1.1.2.2 Email format

```
<input  id="email" type="email" placeholder="Email" name="email" class="form-control form-control-lg" required>
```

Figure 1.14: Email format

This is a validation to make sure the email enter by the user is in the correct format. This is done by simply set the type of input to email.

Figure 1.15: Email message

If the user does not input the right email format, a popup will appear.

1.1.2.3 Password Matching

To make sure the user are sure with the password set, a password matching validation is added.

```
<script>
    var password = document.getElementById("password") // call input id password
  , confirm_password = document.getElementById("password2"); // call input id password2

<!--Function to Validate the password-->
    function validatePassword(){
    //check if value is not the same
      if(password.value != confirm_password.value) {
        confirm_password.setCustomValidity("Passwords Don't Match"); //message if true
      } else {
        confirm_password.setCustomValidity('');
      }
    }

    password.onchange = validatePassword;
    confirm_password.onkeyup = validatePassword;
</script>
```
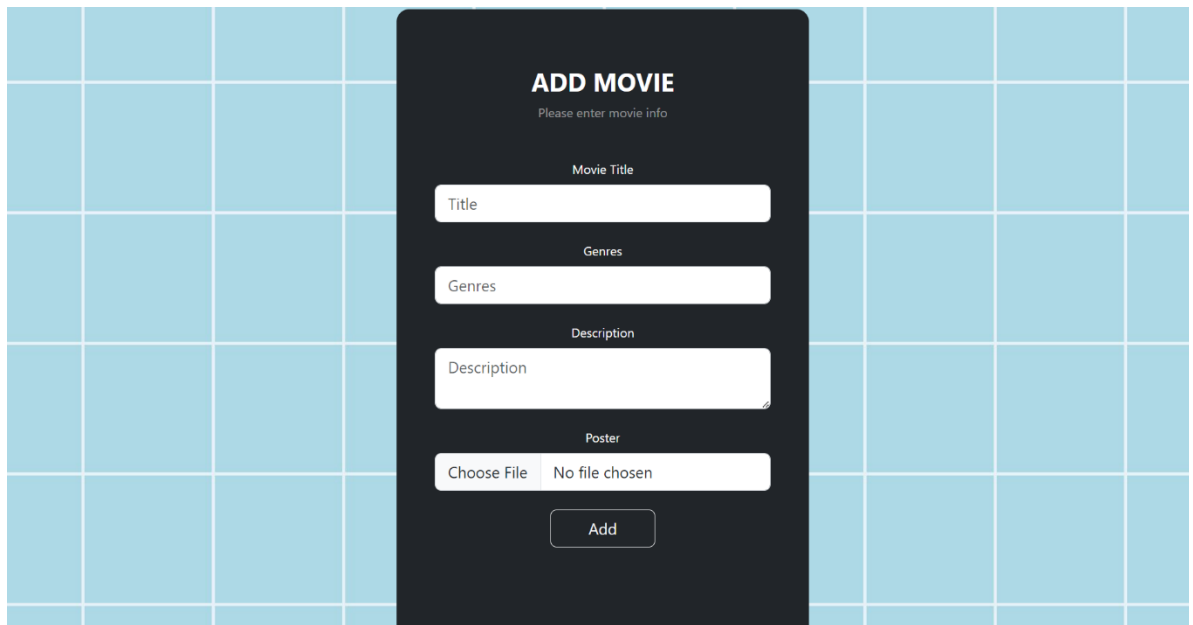
Figure 1.16: Password Validation

REGISTER
Please enter your info

Username
alia

Email
alia@gmail.com

Position
User

Password
...

Confirm Password
....

⚠ Passwords Don't Match

Register

Figure 1.17: Password Don't Match message

This validation is done by using javascript. The input of first password is name password and the input of confirm password is name password2. Figure 1.16 shows the function of validatePassword() where if the input of password and password2 is not the same, a message password don't match will appear like in Figure 1.17.

9

1.2   Admin
1.2.1   Add Movie

Figure 1.18: Add Movie

Admin can add new movie by inserting the title, genres, description and upload the poster for the movie.

Figure 1.19: Add Movie Service

Figure 1.20: Add Movie Controller

Figure 1.21: Add Movie Web

Figure 1.19 and Figure 1.20 is the coding in movie-service. This code is the code to save data into movie table. For Figure 1.21, this is the code for the web application

controller. When admin click add, the data will be added to the database through the movie API in movie-service. After that, the page will be redirect to movieadmin.
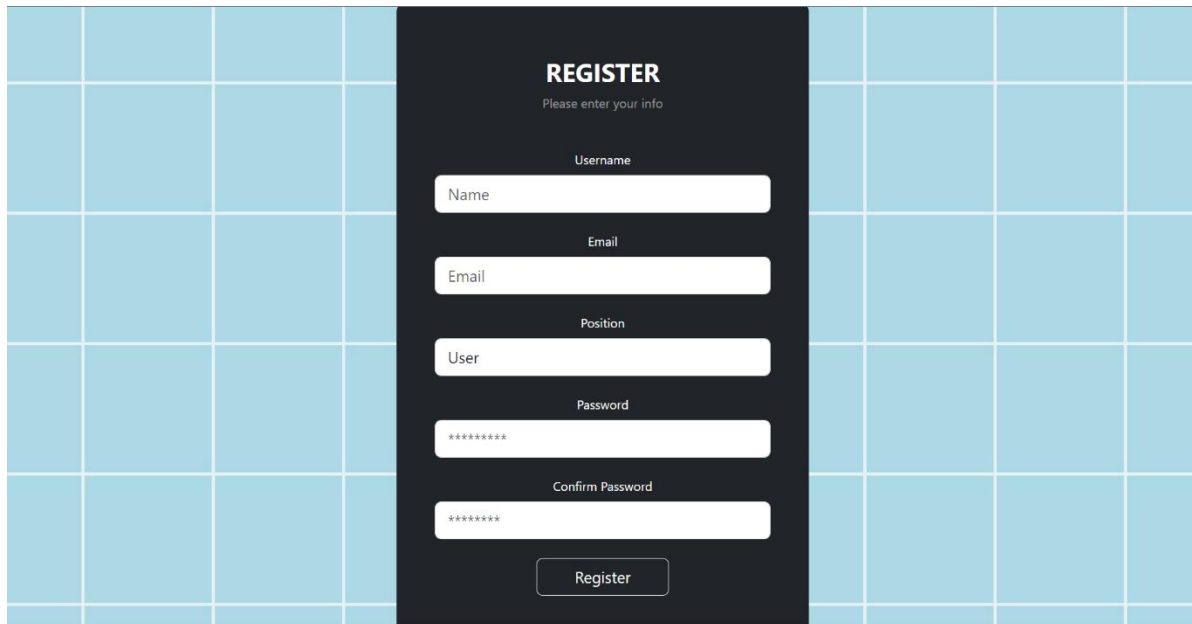
1.2.2 Add User

Figure 1.22: Add User

Admin can add new user by inserting the data username, email, position, password and confirm password.

Figure 1.23: Register Service

Figure 1.24: Register Controller

Figure 1.23 and Figure 1.24 is in user-service. The code save data to user table. Figure 1.13 shows the code in web application to add the user.

1.2.3 View User

Figure 1.25: View User

Admin can view the all the user that has been register.

```java
public List<User> getUsers() { return repo.findAll(); }
```
Figure 1.26: View user Service

```java
@GetMapping("/users")
public List<User> findallUsers() { return service.getUsers(); }
```
Figure 1.27: View User Controller

```java
@GetMapping("/users")
public String findallUser(Model model)
{
    //fetch data of all user from user table via user-service
    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<User> entity = new HttpEntity<~>(headers);
    List<User> dataList = restTemplate.exchange( url: "http://localhost:8085/users", HttpMethod.GET, entity, List.class).getBody();
    model.addAttribute( attributeName: "dataList", dataList); // data is insert here to be use by thymeleaf in html
    return "viewalluser";
}
```
Figure 1.28: View User Web

Figure 1.26 and Figure 1.27 is the coding in user-service. This is a code to list all data in user table. Figure 1.28 is the code in web application. The list is data is fetch from user-service. This data is then put to model.addAttribute and is display using thymeleaf in the html.
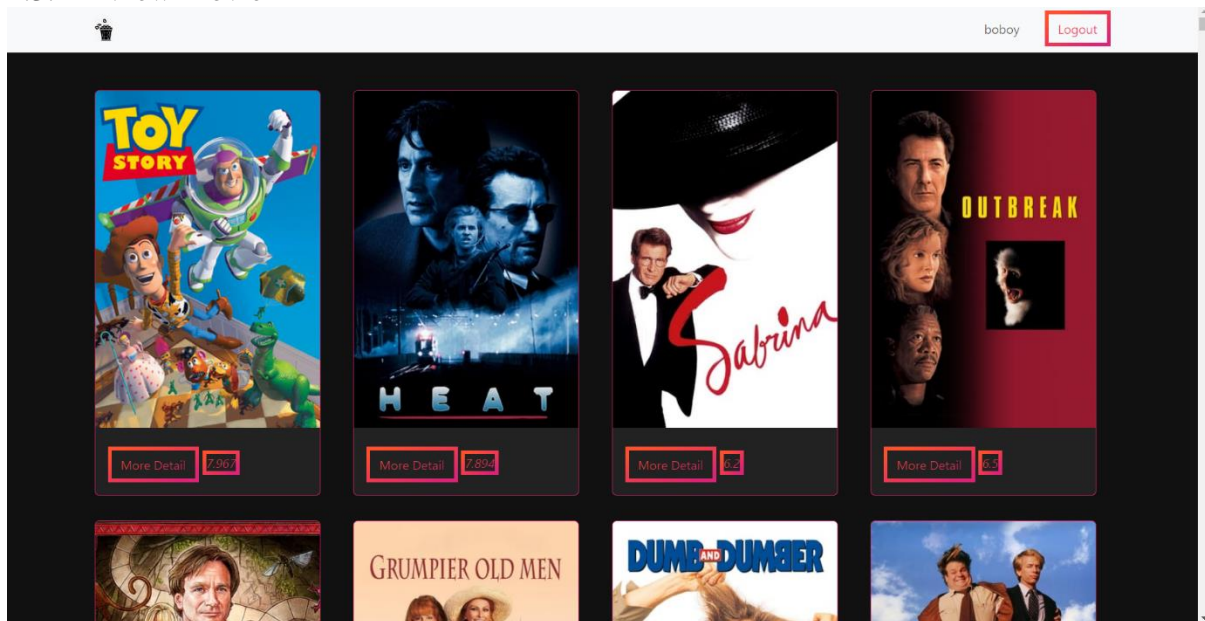
12

1.3   User

1.3.1   View Movie

Figure 1.29: View Movie

After login, user can view all of the movie with its poster and rating. If they want to see more detail on the movie, they can click the More Detail button.

```
public List<Movie> getMovies() { return repo.findAll(); }
```

Figure 1.30: List Movie Service

```
@GetMapping("/movies")
public List<Movie> findallMovies() { return service.getMovies(); }
```

Figure 1.31: List Movie Controller

```
//get the list of all movie in database
HttpHeaders headers3 = new HttpHeaders();
headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
HttpEntity<Movie> entity3 = new HttpEntity<>(headers3);
List<Movie> dataList = restTemplate.exchange( url: "http://localhost:8083/movies", HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>() {}).getBody();
model.addAttribute( attributeName: "dataList", dataList);
```

Figure 1.32: List Movie Web

Figure 1.30 and Figure 1.31 is the code in movie-service. This code list all the movie from movie table in database. Figure 1.32 is the code in web application controller that fetch all the movies data and put it to model.addAttribute to be display in the html.
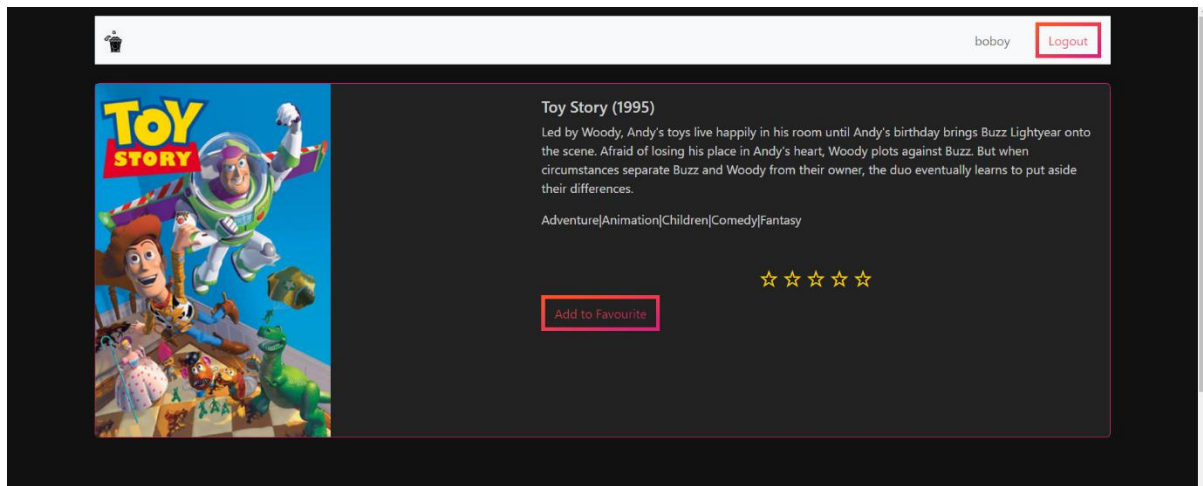
Figure 1.33: View Movie Detail

User can also view the detail of the movie by clicking the More Detail button.
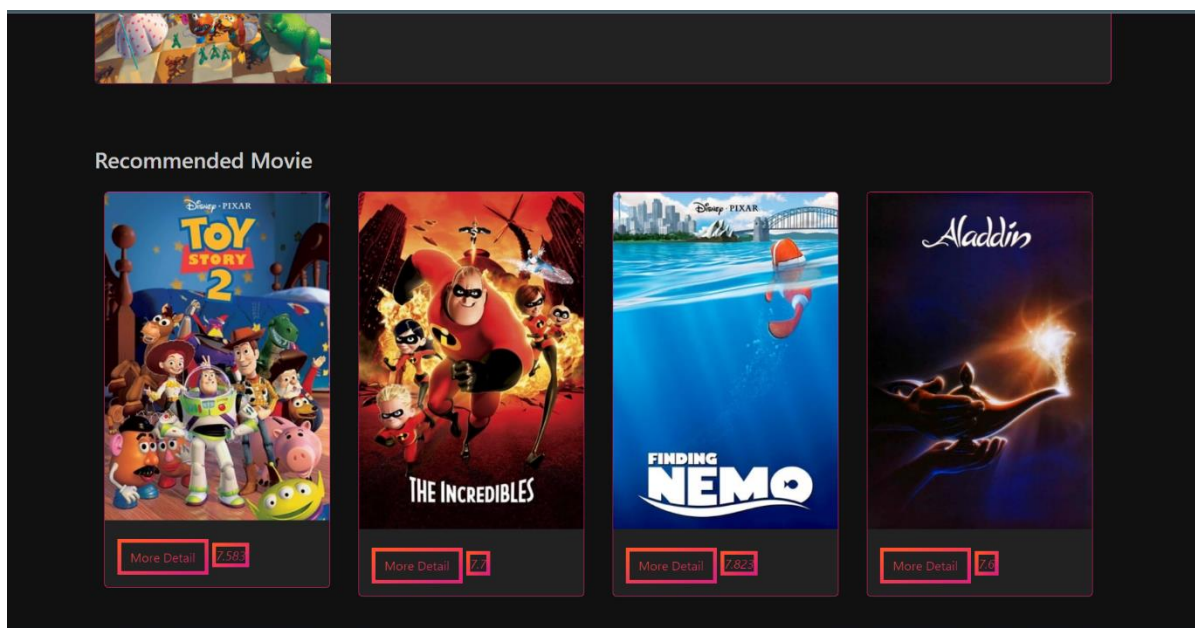
1.3.2 View Recommended Movie

Figure 1.34: Recommended Movie

User can see the recommended movie similar to the movie that they have chosen.

```
@Autowired
JdbcTemplate jdbcTemplate;

/*Map the list of result*/
1 usage
private RowMapper<Recommend> rowMapper = (ResultSet rs, int rowNum) -> {
    Recommend matrix = new Recommend();
    matrix.setMovieId(rs.getInt( columnIndex: 1));

    return matrix;
};

/*Query to find movie_ID with similarity more than 0.5 and fetch the first 5 only*/
1 usage
public List<Recommend> 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 1.35: Recommend Movie Repository

Figure 1.35 shows the repository of recommend-service. The code is the query to fetch movie id that has a similarity of more than 0.5 with the id inserted from the Matrix table with contain pearsons correlation matrix.

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

Figure 1.36: Recommend Service

```
@PostMapping("/register")
public User register(@RequestBody User user) { return service.register(user); }
```

Figure 1.37: Recommend Controller

Both Figure 1.36 and Figure 1.37 are the code in service and controller for the repository in Figure 1.35.

```
public List<Movie> getMoviesbyId(int id) { return repo.findByMovieId(id); }
```

Figure 1.38: List movie by Id service

```
@GetMapping("/movies/{id}")
public List<Movie> findallMoviesbyId(@PathVariable int id) { return service.getMoviesbyId(id); }
```

Figure 1.39: List movie by Id Controller

Figure 1.38 and Figure 1.39 are the code in movie-service that list all the movies by id.

```
/*call the movie id requested*/
HttpHeaders headers = new HttpHeaders();
HttpEntity<Movie> entity = new HttpEntity<~>(headers);
Movie dataList = restTemplate.exchange( url: "http://localhost:8083/findbyid/"+id, HttpMethod.GET, entity, Movie.class).getBody();
model.addAttribute( attributeName: "dataList", dataList);
```

Figure 1.40: Selected Movie

```
/*call similar id*/
HttpHeaders headers2 = new HttpHeaders();
headers2.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
HttpEntity<Recommend> entity2 = new HttpEntity<~>(headers2);
List<Recommend> dataid = restTemplate.exchange( url: "http://localhost:8086/recommends/" + id, HttpMethod.GET, entity2, new ParameterizedTypeReference<List<Recommend>>() {}).ge
```

Figure 1.41: Similar Id

```
/*call movie with similar id*/
HttpHeaders headers3 = new HttpHeaders();
headers3.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
HttpEntity<Movie> entity3 = new HttpEntity<~>(headers3);
List<Movie> dataListmovie = new ArrayList<>();
for (Recommend recommend : dataid) {
    int movieId = recommend.getMovieId();
    List<Movie> recommendedMovies = restTemplate.exchange( url: "http://localhost:8083/movies/" + movieId, HttpMethod.GET, entity3, new ParameterizedTypeReference<List<Movie>>()
    dataListmovie.addAll(recommendedMovies);
}
model.addAttribute( attributeName: "dataListmovie", dataListmovie);
return "movieinfo";
```

Figure 1.42: List of movie by Id

Figure 1.40, Figure 1.41 and Figure 1.42 are the code in web application. The code in Figure 1.40 is to fetch the movie with the id of movie selected by the user. Figure 1.41 also use the id of movie selected by the user to find list of similar movie id from the matrix table. The result of this is then send to the code in Figure 1.42 to list all the movie detail with the id from the result.
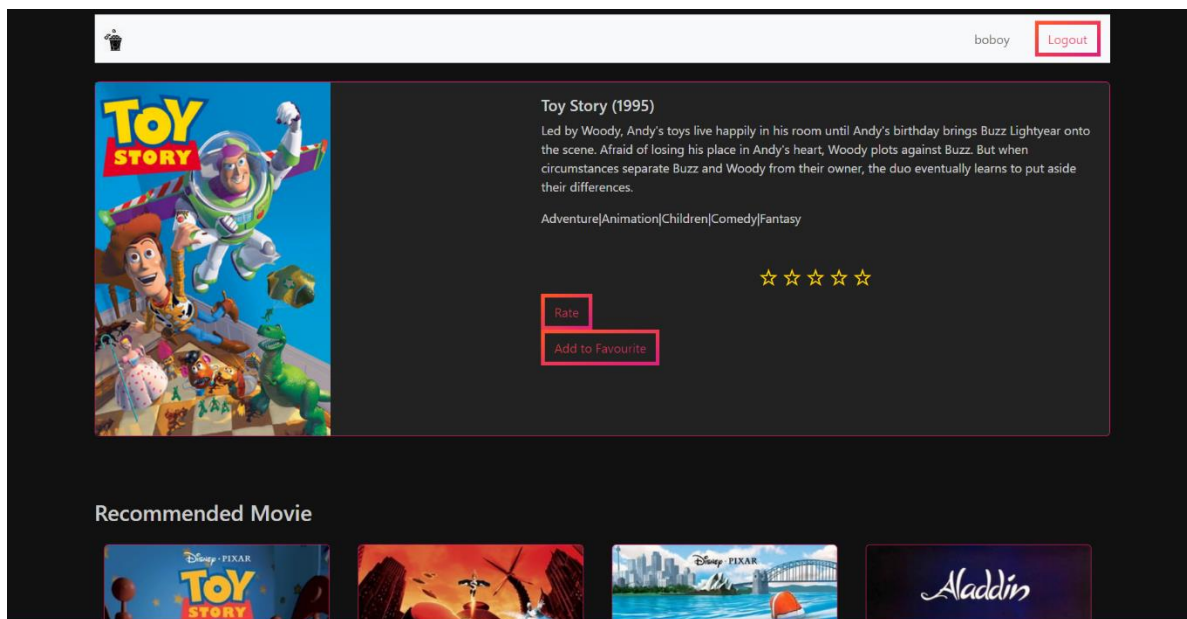
1.3.3 Add Favorite

Figure 1.43: Add to favourite

User are able to add their chosen movie to favourite by clicking the Add to Favourite button.

```
public Favourite saveFav(Favourite favourite) { return repo.save(favourite); }
```

Figure 1.44: Add Favourite Service

```
@PostMapping("/addFav")
public Favourite addFav(@RequestBody Favourite favourite) { return service.saveFav(favourite); }
```

Figure 1.45: Add Favourite Controller

```
@GetMapping("/fav")
public String addFav(@RequestParam("id") int id,@RequestParam("userId") int userId, @ModelAttribute("favourite") Favourite favourite, Model model)
{

    // get id for favourie (it will auto increment)
    int idf = favourite.getId();

    favourite = new Favourite(idf, id, userId); // get all the data to be inserted in fav table
    HttpEntity<Favourite> entity = new HttpEntity<Favourite>(favourite); //assign the data to the entity
    restTemplate.exchange( url: "http://localhost:8087/addFav", HttpMethod.POST, entity, Favourite.class).getBody(); //entity data send to fav-service to add in db

    return "favourite";// redirect to favourite page
}
```

Figure 1.46: Add Favourite Web

Figure 1.44 and Figure 1.45 is the code in fav-service. This are the code to insert data to fav table. Figure 1.46 is the code in web application. Parameter for movieId is send from the html as id and parameter for userId is send from html as userId. Int idf get id for favourite which is auto increment. All this data is put as favourite and assign to entity. The entity data is send to fav-service to be added to the database.
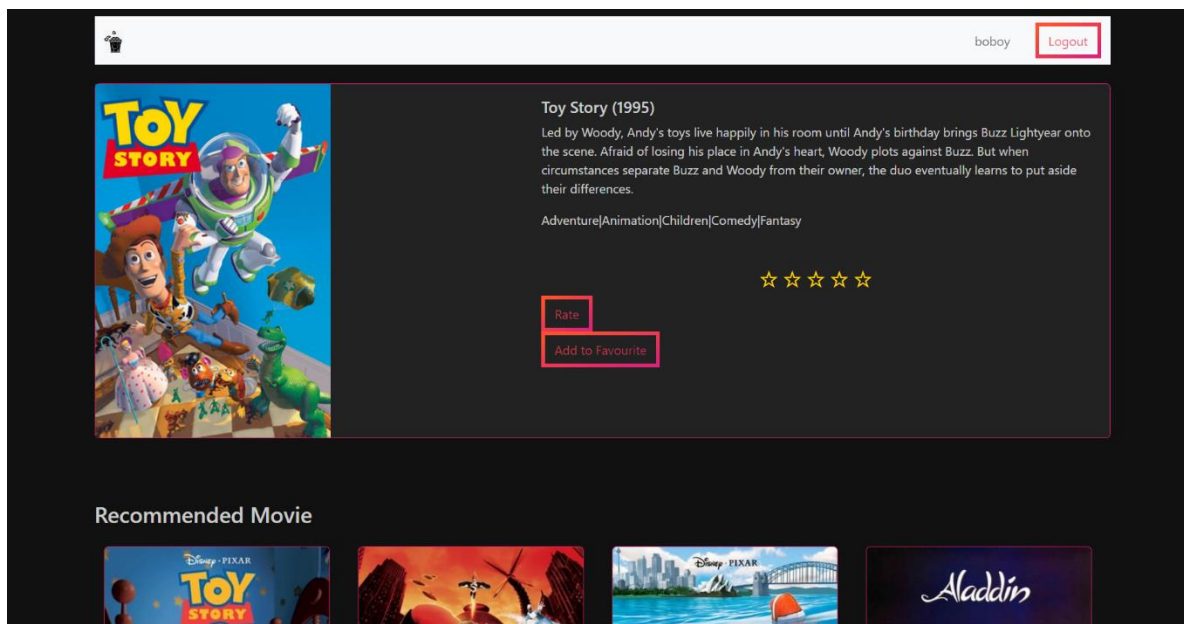
1.3.4 Give Rating

Figure 1.47: Give Rating

User are able to give rating to a movie by clicking on the star and then click the button rate. The star is actually a radio button that is in a form.

```
public Rating saveRating(Rating rating) { return repo.save(rating); }
```
Figure 1.48: Rating Service

```
@PostMapping("/addrating")
public Rating addRating(@RequestBody Rating rating) { return service.saveRating(rating); }
```
Figure 1.49: Rating Controller

```
String timestamp = rating.getTimestamp(); //get timsestamp data
int movieId = rating.getMovieId(); //get movieId data
int userId = rating.getUserId(); //get userid data
double movierating = rating.getMovierating(); //get movie rating
int id = rating.getId(); //get id (auto increment)
rating = new Rating(id,userId,movieId,movierating,timestamp); // assign the data to rating
HttpEntity<Rating> entity = new HttpEntity<Rating>(rating); //assign rating to entity
restTemplate.exchange( url: "http://localhost:8084/addrating", HttpMethod.POST, entity, Rating.class).getBody(); //entity data to be added to db by rating-service
```
Figure 1.50: Rating Web

Figure 1.48 and Figure 1.49 are the code in rating-service to add data to rating table. Figure 1.50 is the data in web application. The data for each attribute needed to add to the table is fetch from html and this data is then assign to rating that is then assign to entity. Entity send the data to rating-service to be added to the database.

Module

2.1   Login
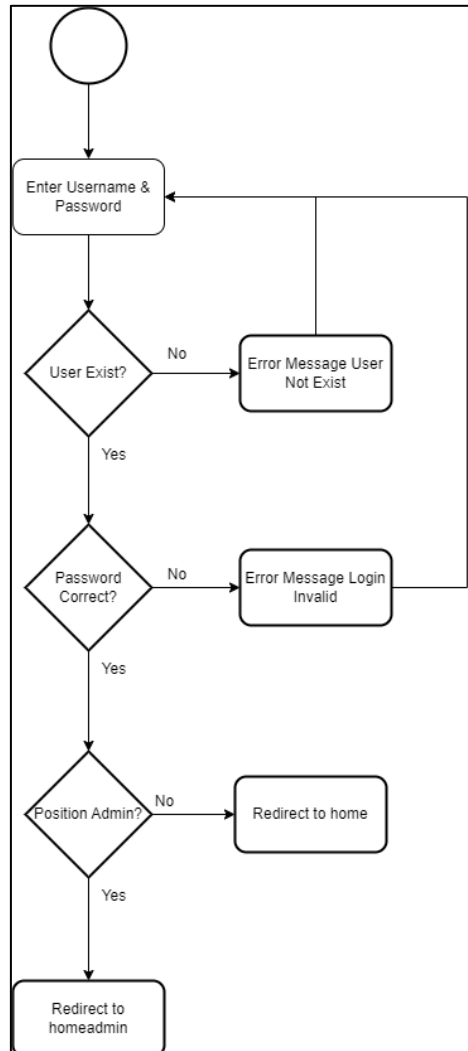## 2.1.1   Flowchart

Figure 2.1: Login Flowchart

Figure 2.1 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 index page back. If user exist but the password is wrong, error message login invalid will appear and the page will be redirect to the login page. If the password is also correct then the position of the user is check. If the user is an Admin, the page id directed to home for admin page. If the user is not an admin, the page will be directed to the home page.
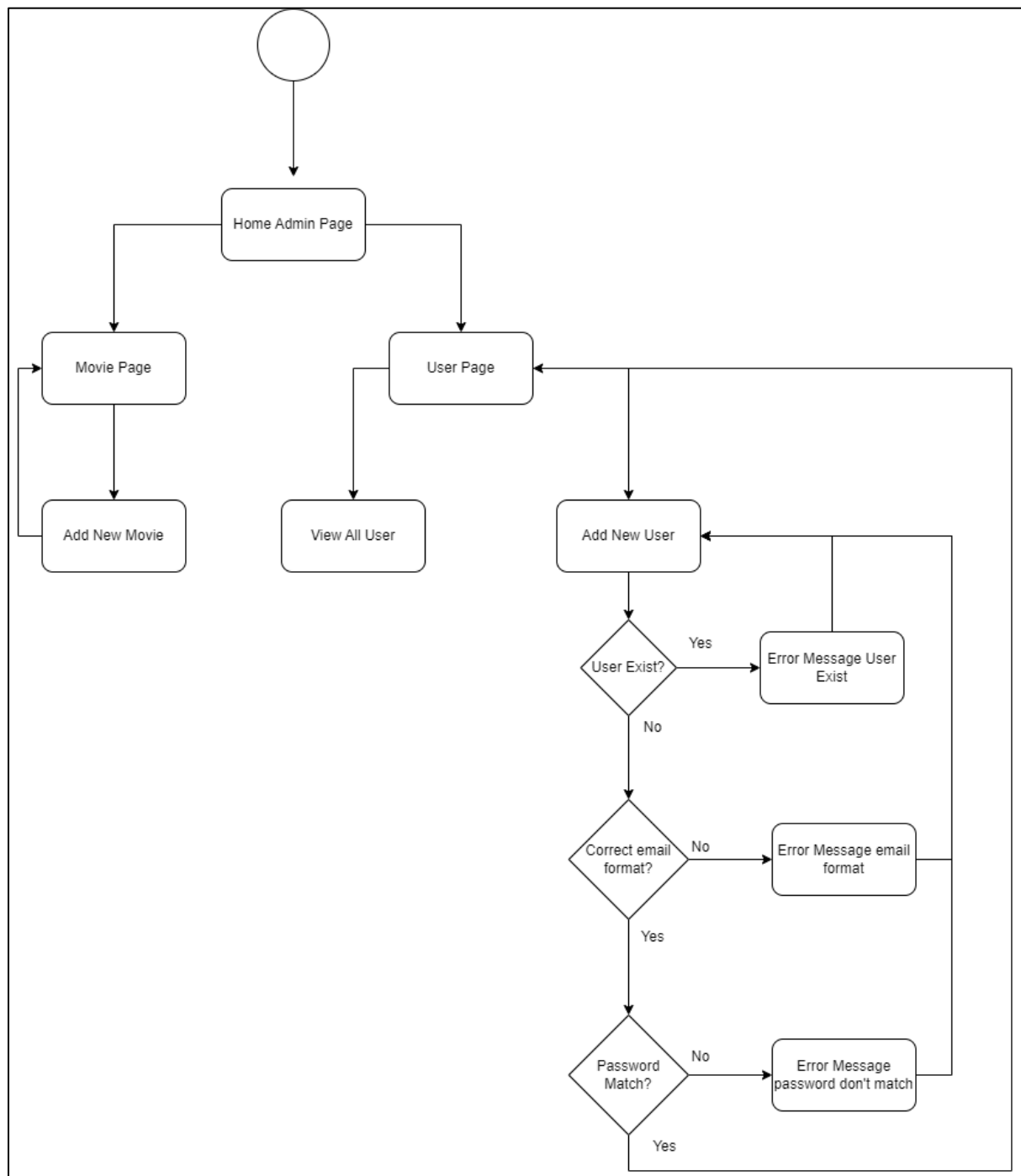
2.2 Admin
2.2.1 Flowchart

Figure 2.2: Admin Flowchart

Figure 2.2 is the flowchart for admin page. Admin can go to movie page or user page. In movie page, admin can add new movie and after adding, the page will be redirect back to movie page. In user page, admin can view all user or add new user. When adding new user, if the username inserted exist, the error message User exist will appear and the page is redirect back. If the

email format is wrong, error message email format will appear, If the password enter does not match, error password don't match will appear. If are ok, the data is inserted in database and redirect to user page.
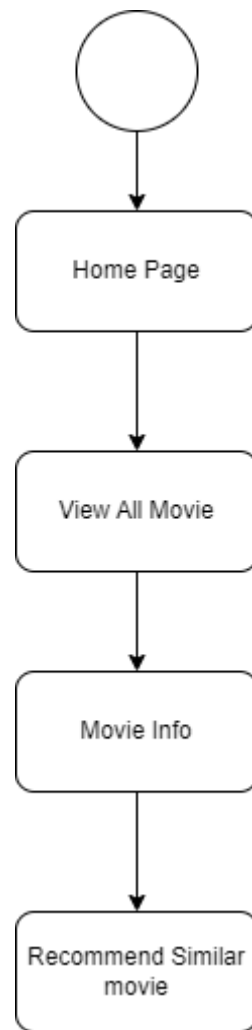
2.3   User
2.3.1   Flowchart

Figure 2.3: User Flowchart

Figure 2.3 is the flowchart for user. After login, user is direct to home page that view all movie. When clicking on a chosen movie, the page is direct to page movie info that contains information on the movie and the movie similar to the selected movie.

2.4   Db2

2.4.1   Entity Relational Diagram (ERD)
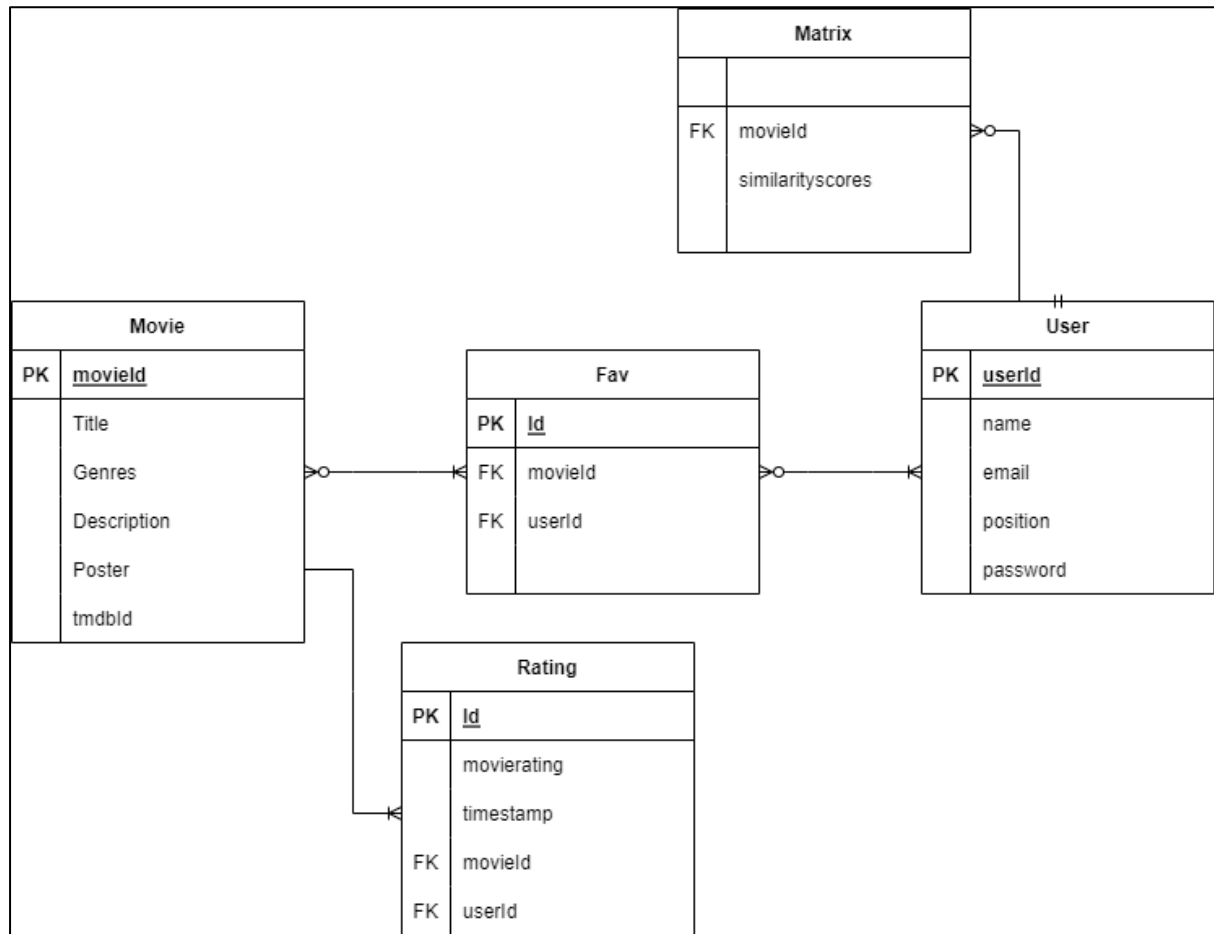

Figure 2.4: ERD

Figure 2.4 is the ERD of this project in the db2 database.

2.4.2   Data Dictionary

2.4.2.1 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	
tmdbId	Int	4	Not null

2.4.2.2 User

Field Name	Data Type	Field Length	Constraint
userId	Int	4	Primary Key
Name	Varchar	255	
Email	Varchar	255	
Position	Varchar	255	
password	Varchar	255	

2.4.2.3 Rating

Field Name	Data Type	Field Length	Constraint
Id	Int	4	Primary Key
Movierating	Double	8	
Email	Varchar	255	
Timestamp	Varchar	255	
movieId	int	4	Foreign Key
userId	int	4	Foreign Key

2.4.2.4 Fav

Field Name	Data Type	Field Length	Constraint
Id	int	4	Primary Key
movieId	int	4	Foreign Key
userId	int	4	Foreign Key

2.5 Microservice

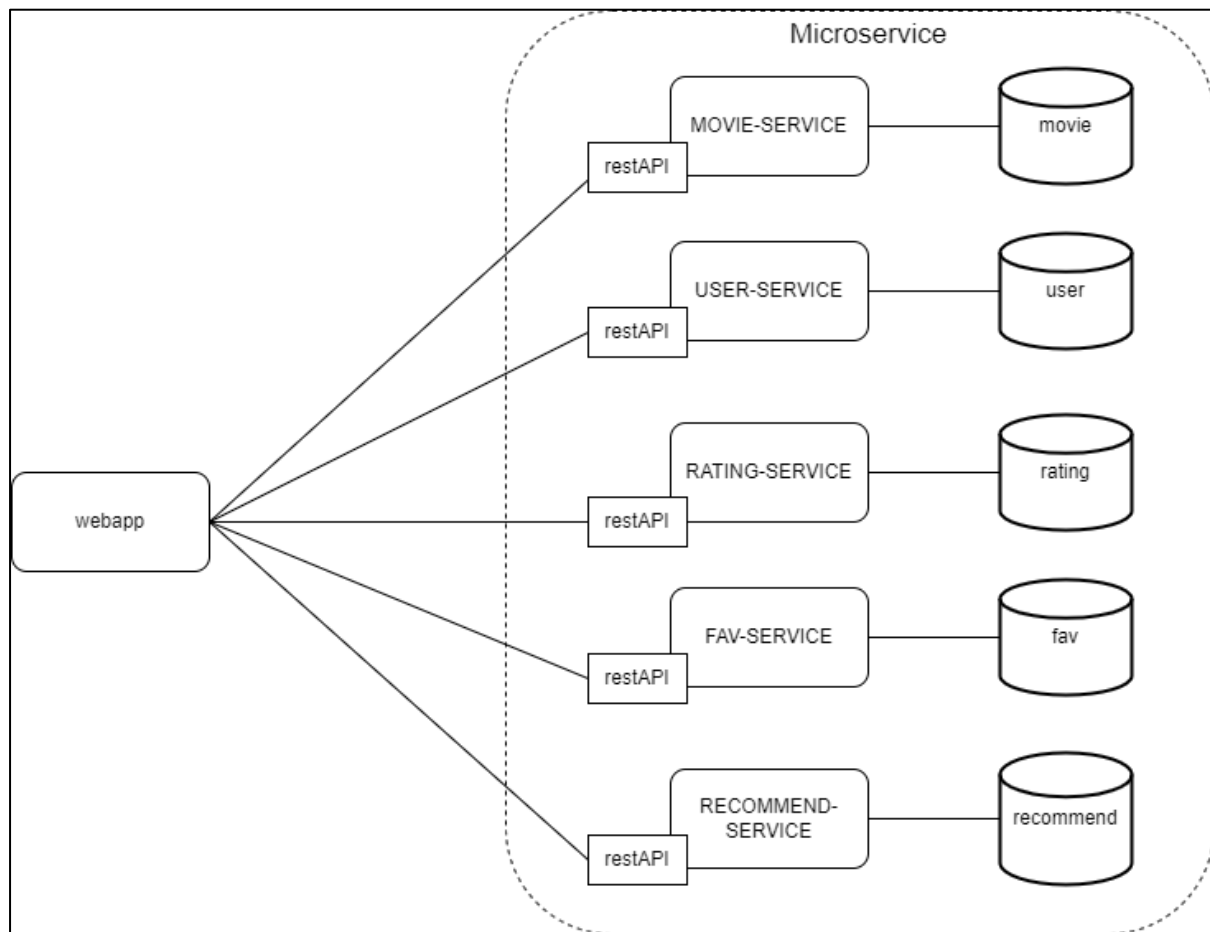### 2.5.1 Microservice Architecture

Figure 2.5: Microservice Architecture

Microservices architecture is a concept of building an application based on breaking it into multiple modules. Each module has its own specific responsibilities but communicates with others to form a unified system. Each microservice exposes a REST API that conforms to the constraints of REST architectural style. A single microservice has its own set of responsibilities. MOVIE-SERVICE are responsible for action related to movie table, USER-SERVICE are responsible for action related to user table, RATING-SERVICE are responsible for action related to rating table, FAV-SERVICE are responsible for action related to fav table and RECOMMEND-SERVICE are responsible fore action related to recommend table.

Pearson's Correlations

Pearson's correlation is a statistical method used to measure the strength and direction of the linear relationship between two continuous variables. It assigns a value between -1 and 1 where 0 is no correlation, 1 is total positive correlation and -1 is total negative correlation.

For this project, Pearson's Correlations is use to measure the similarity scores between the movies based on the ratings given by the user. A positive correlation for this project is when the ratings for two movies are similar and the negative correlation is when the rating has low similarity in ratings. The closer the score is to 1, the more similar the ratings of the two movies.

The following are the steps Pearson's Correlations are use is this project.

1. Dataset of movie ratings are obtained. This data is stored in a matrix where the row is the is the movie and the column is the user.

2. Pearson's Correlation is use to calculate the similarity of each pair of movies by comparing the ratings of the user for the two movies. Below are the formula of Pearson's Correlation:

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

Where,

r = Pearson Correlation Coefficient

x_i = x variable samples y_i = y variable sample

$\bar{x}$ = mean of values in x variable $\bar{y}$ =mean of values in y variable

3. The result of the calculation is stored in a table or matrix.