FINAL EXAM PROJECT: GUESS THAT MOVIE

Enterprise Java Development

By: Yin Tung Ng

Student ID: 991602581

PROG32758

Instructor: Dario Guiao

December 18, 2020

Contents

Introduction

This purpose of this project is to develop a **Spring Boot Java application called "Guess That Movie"**. The game application will allow players to guess the name of a random movie, selected from a list of G-rated Pixar and Disney animated films.

The **list of movies** contains: Finding Nemo, Monsters University, Toy Story, Wall-E, The Lion King, Aladdin, Cars, Mulan, Beauty and the Beast, The Jungle Book, Winnie the Pooh.

Rules: A movie is randomly selected from the database's list of movies. The player is invited to guess if a letter can be found in the movie name. If found, the player is offered the chance to guess the whole movie name. If the full name guess is correct, the player wins the game round. Alternatively, if the player fails 7 times to guess the random movie's name, the player loses the game round.

Players can enter lowercase letters (a-z), but cannot enter numbers. However, the game matching process is **case-insensitive**.

Players can register for a password-protected user account. The registered account can save the player's score and the number of games played. Regardless if the player is logged in, the game is played the same way. However, only registered players can see their running score and the scoreboard.

The application uses the following **dependencies**: Spring Boot, Spring Web, Thymeleaf tags and fragments, Spring Security, Lombok, Spring Data JDBC, and H2 Database. Basic web dependencies like HTML and CSS were used to output the game pages. The web pages are stylized with a beautiful CSS colour scheme of slate grey, green, and orange.

Spring Security allows for the registration and login of user players. The encrypted password credentials are stored in the database. Security settings control the display of content by using account roles and authentication (login) status. If a registered *non-admin* user tries to access an admin resource, they will be denied access.

The **H2 database** stores tables for the user, security role, scoreboard, and movie entities. Some of these data entities also have **corresponding Java Bean classes** for database access object interaction.

The admin account can play the game and access the admin pages. An **admin can perform CRUD operations** to the movie and scoreboard tables. They can add, modify, and delete movie names. Also, they can modify or delete scoreboard records.

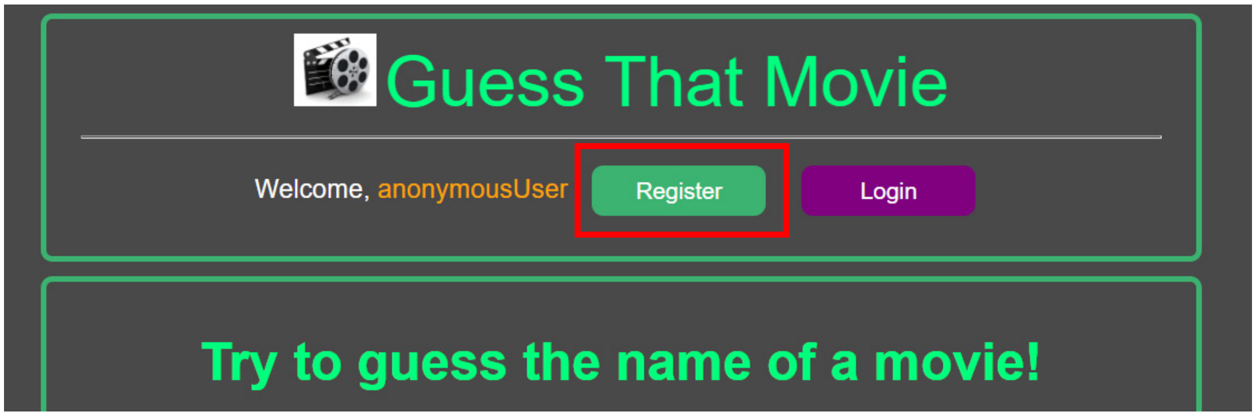
For enhancement, a **header menu bar (using Thymeleaf fragments)** appears at the top of all the gameplay pages. In the menu bar, the username, score, and buttons will vary depending on the login status: non-registered player, logged-in user, or admin user.

Navigation

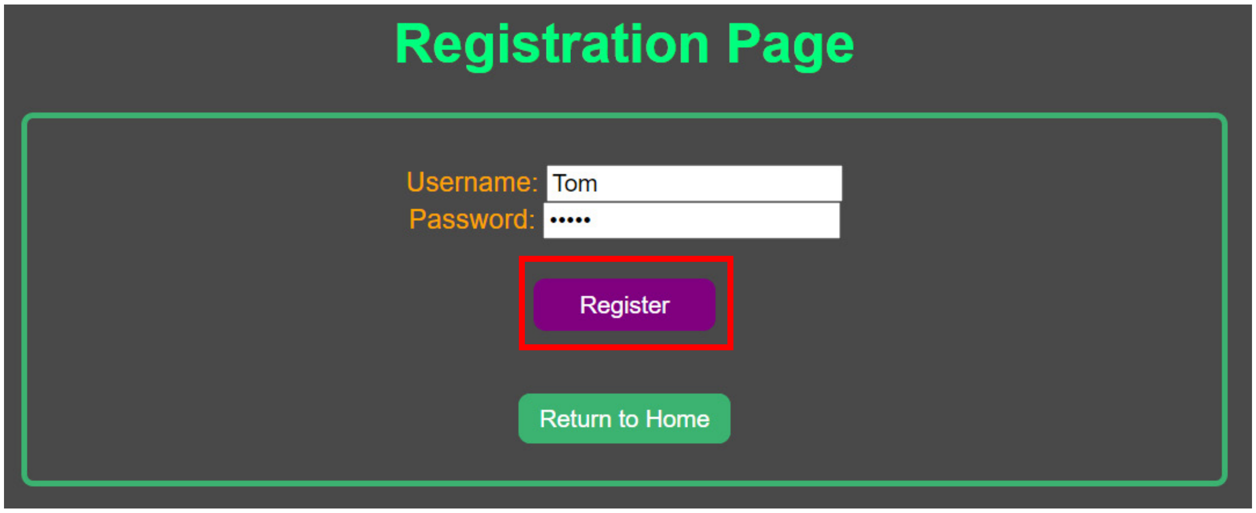
Screenshots will show the navigation through the application pages. A red box outline will indicate the button being pressed, to link to the next page.

Registration, Login, and Logout

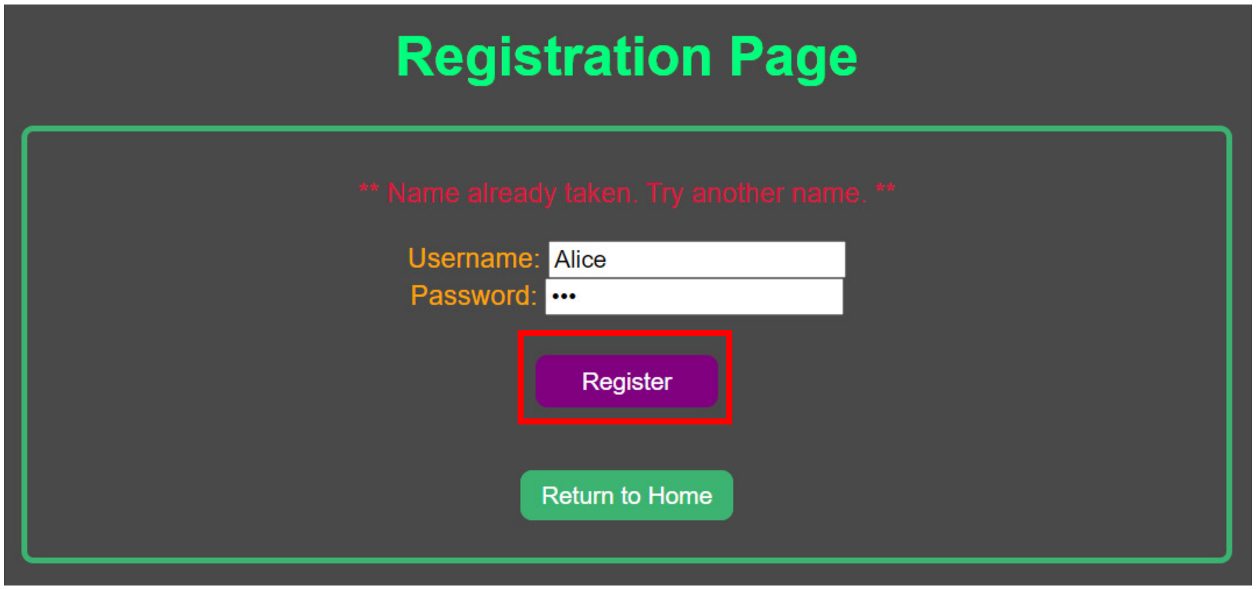
We can register for a new user account to demonstrate the navigation. A header menu bar shows the Register and Login buttons.



We can try to register with a username that is already taken. Usernames must be unique.



An error message is shown. Now, let us register with a new username "Alice" and a password.

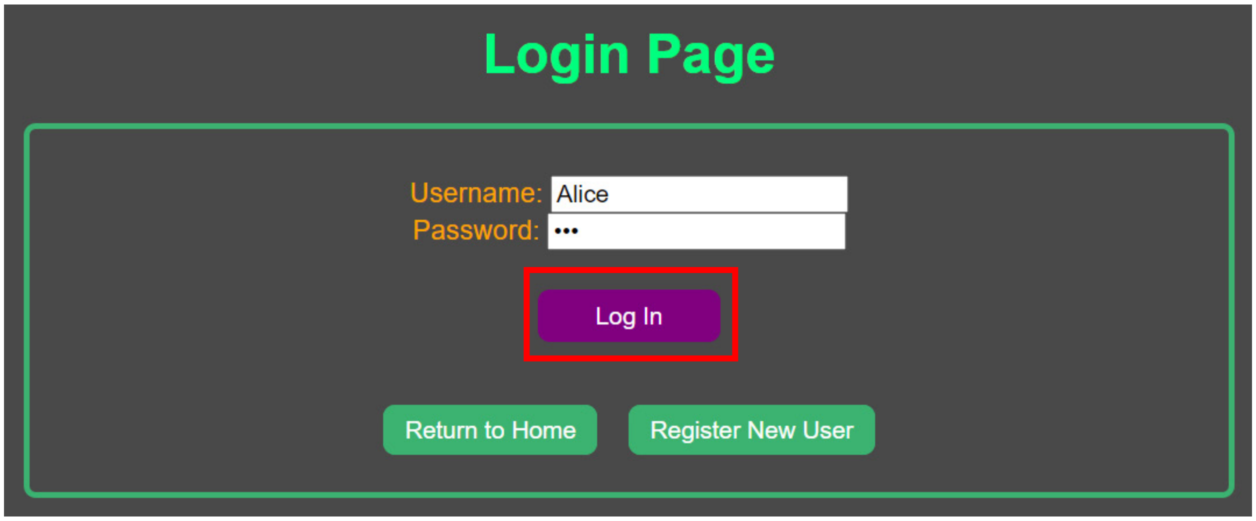


Registration Page

** Name already taken. Try another name. **

Username: Alice
Password: •••

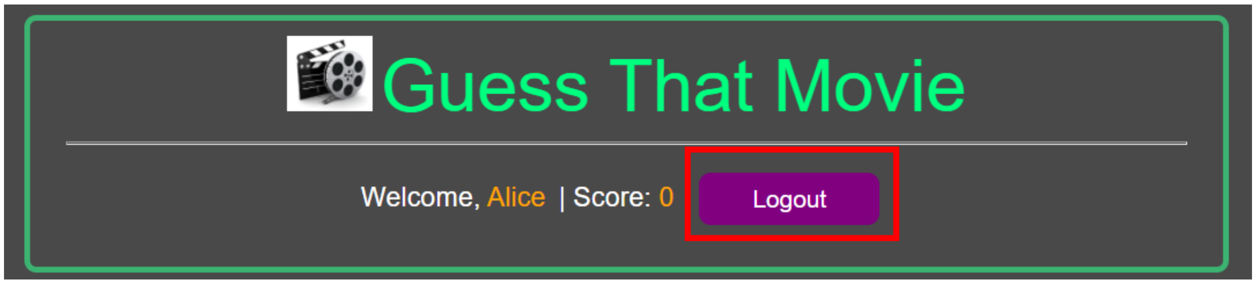
We now login with the newly registered user account.



Login Page

Username: Alice
Password: •••

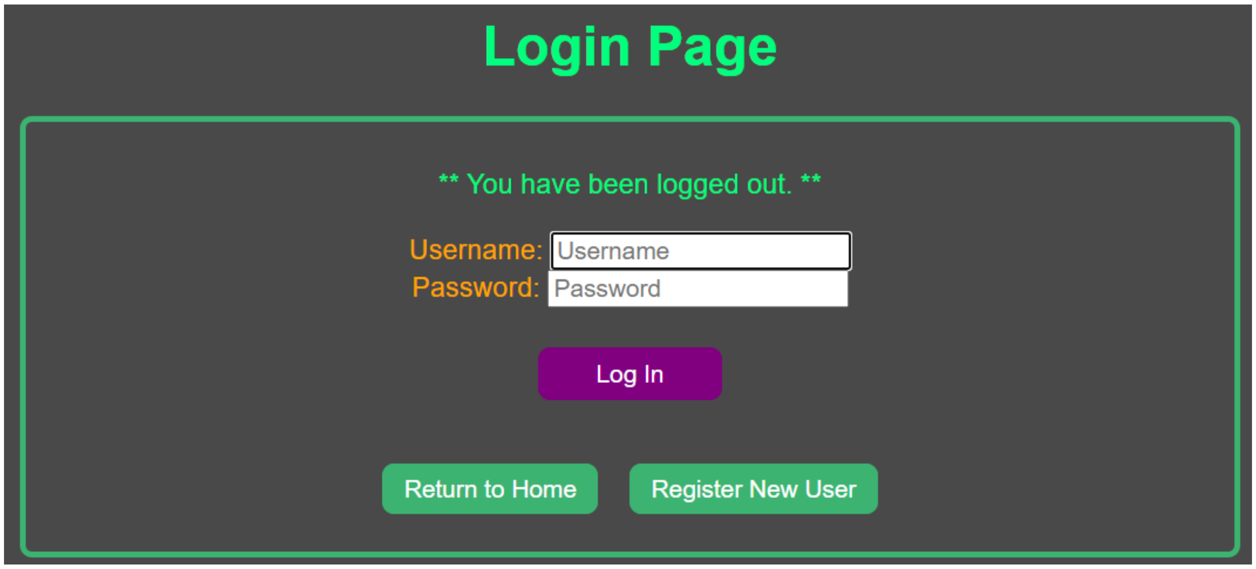
The new user is now logged in. The menu bar also changes. It now shows the username, score, and logout button. The user can now logout.



Guess That Movie

Welcome, Alice | Score: 0

The logout redirects to the login page.



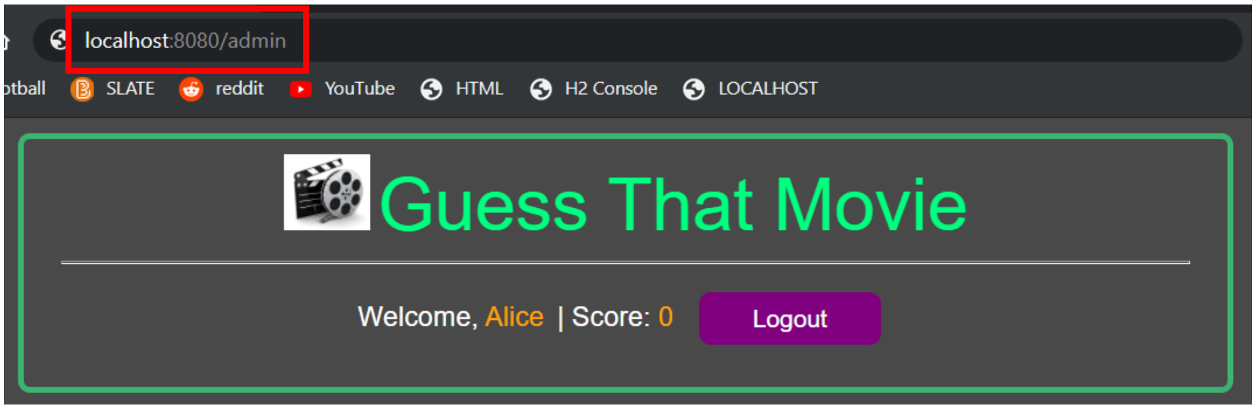
Login Page

** You have been logged out. **

Username: Username
Password: Password

Access Denied

The user does not have a button linking the Admin page. But they can try to access it by URL. They will be rejected when they try to go to "/admin".

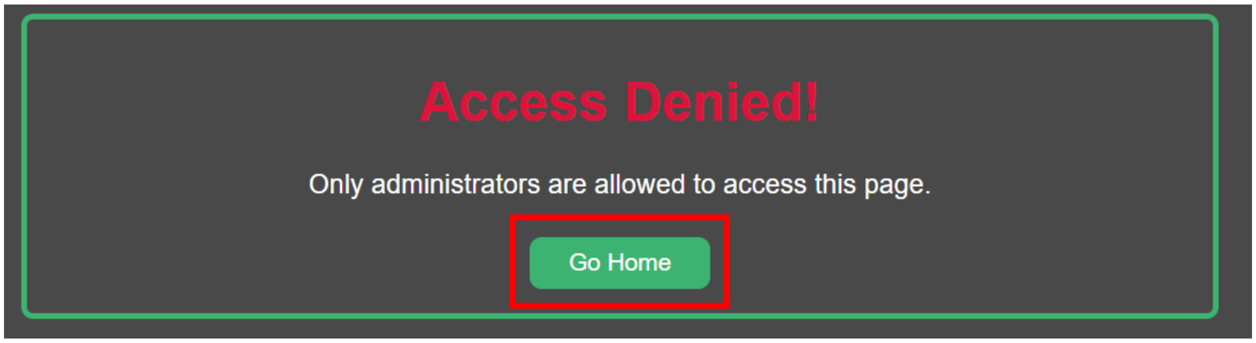


localhost:8080/admin

Guess That Movie

Welcome, Alice | Score: 0

The Access Denied page tells the user that they cannot access the admin page.

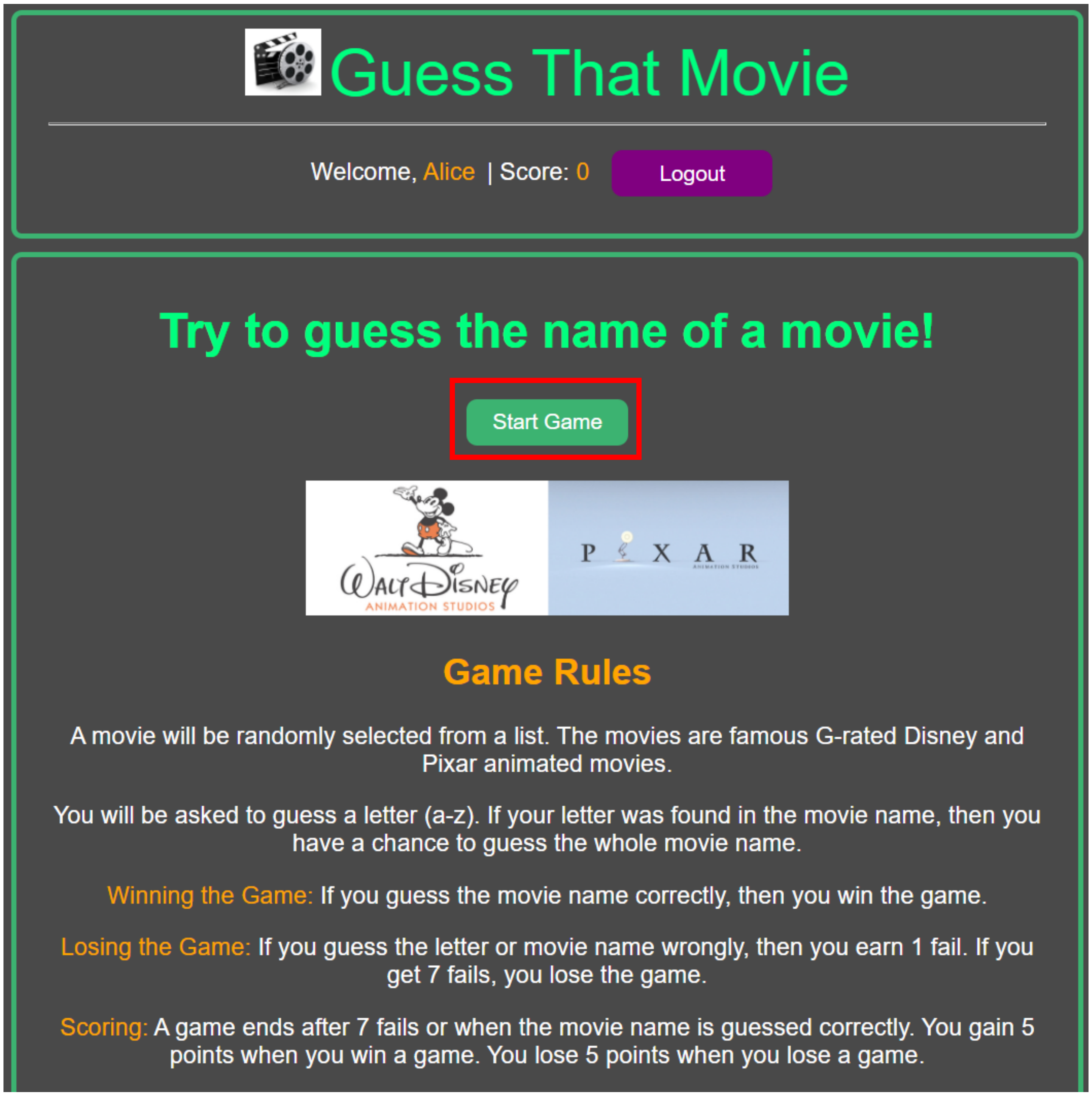


Access Denied!

Only administrators are allowed to access this page.

Gameplay

The game begins at the page index.html. The game rules are shown below the Start Game button. The game will be played with a logged-in user, so the scoreboard can be shown.



Try to guess the name of a movie!

Game Rules

A movie will be randomly selected from a list. The movies are famous G-rated Disney and Pixar animated movies.

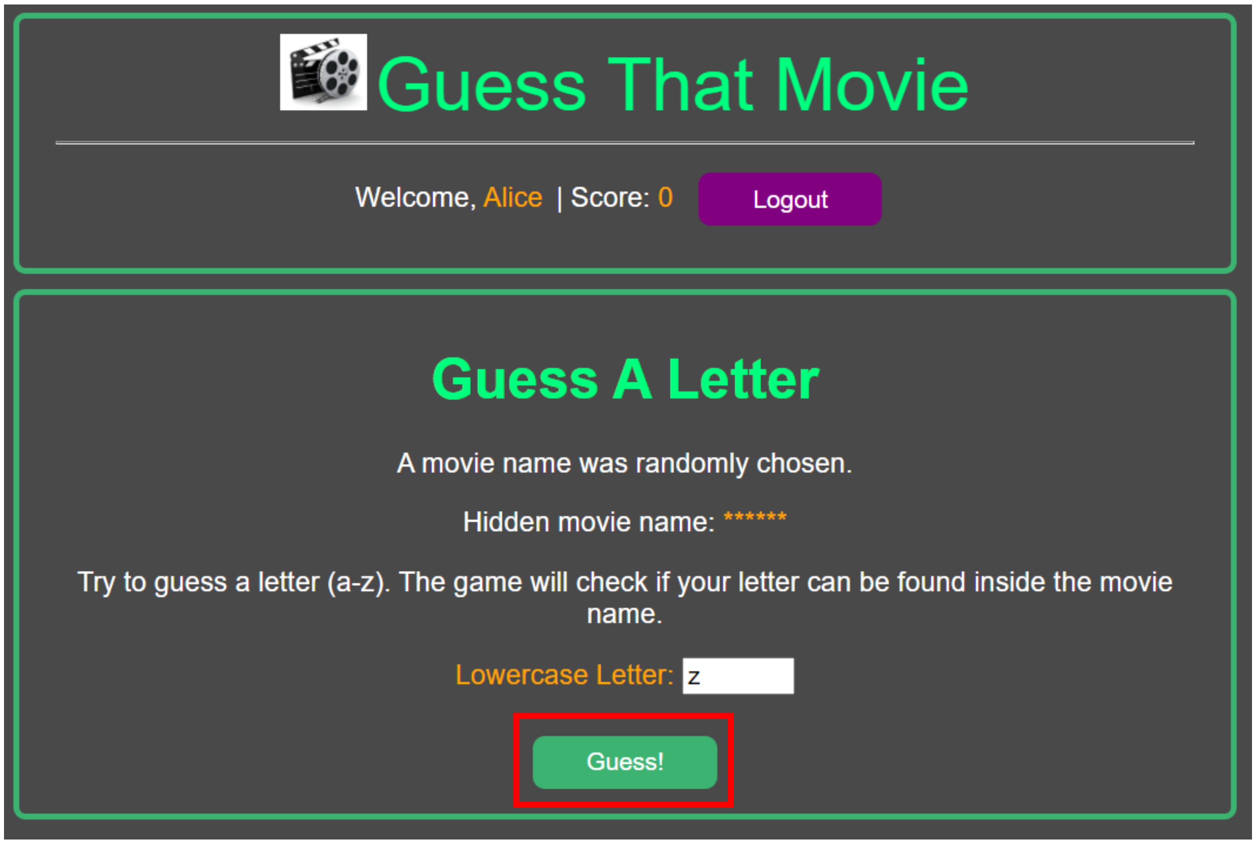
You will be asked to guess a letter (a-z). If your letter was found in the movie name, then you have a chance to guess the whole movie name.

Winning the Game: If you guess the movie name correctly, then you win the game.

Losing the Game: If you guess the letter or movie name wrongly, then you earn 1 fail. If you get 7 fails, you lose the game.

Scoring: A game ends after 7 fails or when the movie name is guessed correctly. You gain 5 points when you win a game. You lose 5 points when you lose a game.

The user can try to guess a letter.


Guess That Movie

Welcome, Alice | Score: 0

Guess A Letter

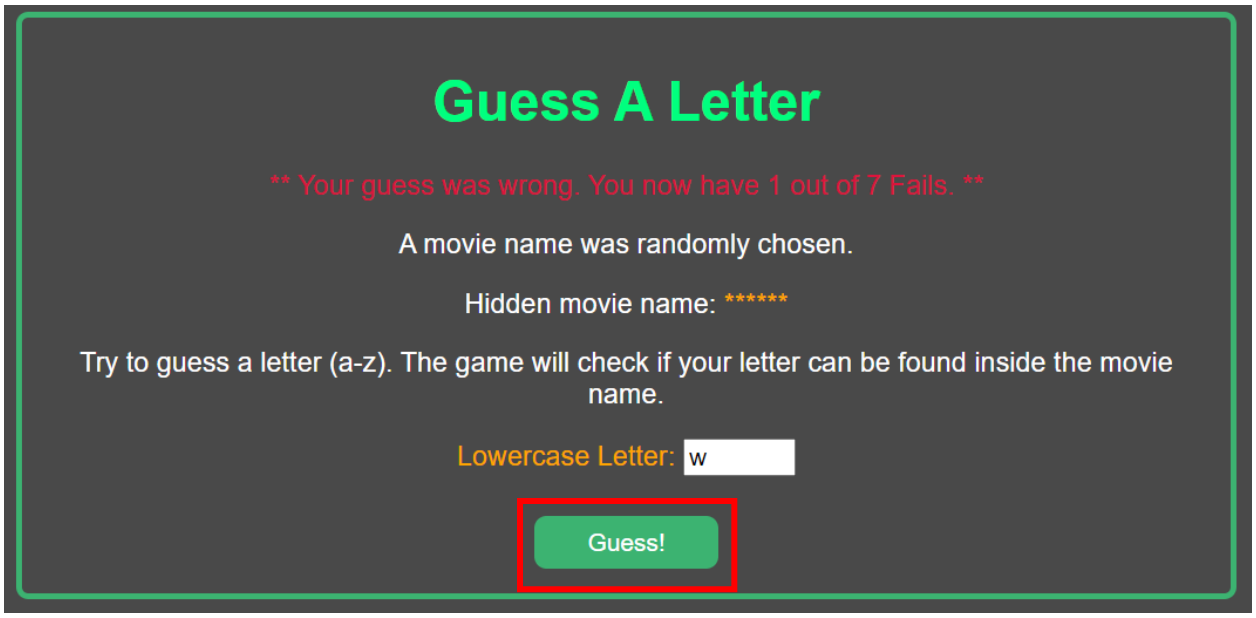
A movie name was randomly chosen.

Hidden movie name: ******

Try to guess a letter (a-z). The game will check if your letter can be found inside the movie name.

Lowercase Letter: z

The guess was incorrect. Let us try again.


Guess A Letter

** Your guess was wrong. You now have 1 out of 7 Fails. **

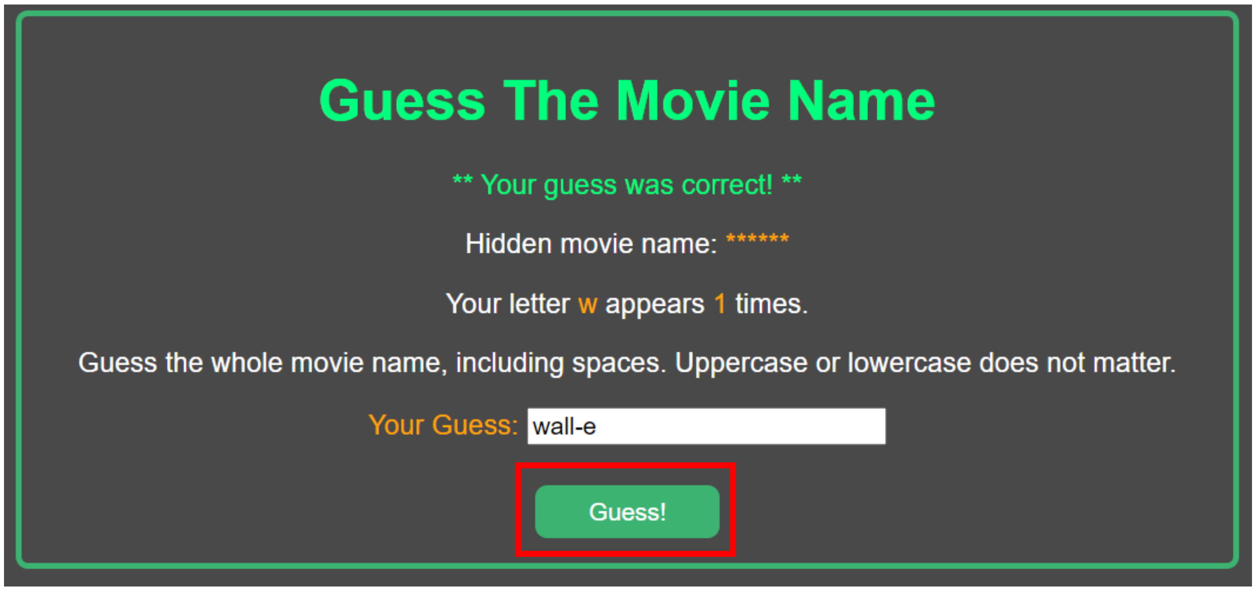
A movie name was randomly chosen.

Hidden movie name: ******

Try to guess a letter (a-z). The game will check if your letter can be found inside the movie name.

Lowercase Letter: w

The letter guess was found. Let us guess the movie name.



Guess The Movie Name

** Your guess was correct! **

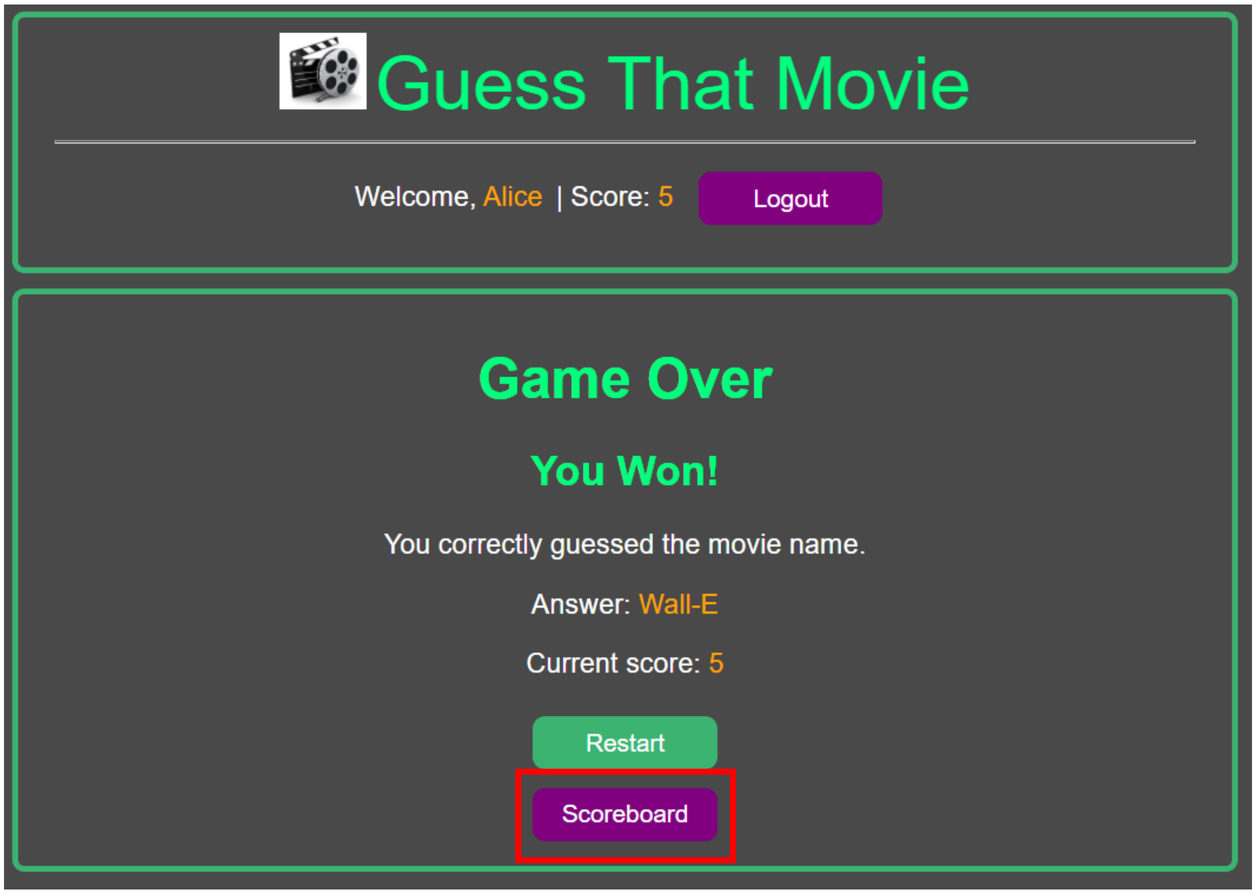
Hidden movie name: ******

Your letter w appears 1 times.

Guess the whole movie name, including spaces. Uppercase or lowercase does not matter.

Your Guess: wall-e

The guess was correct, and the user won the game. We can go to see the scoreboard.



Guess That Movie

Welcome, Alice | Score: 5

Game Over

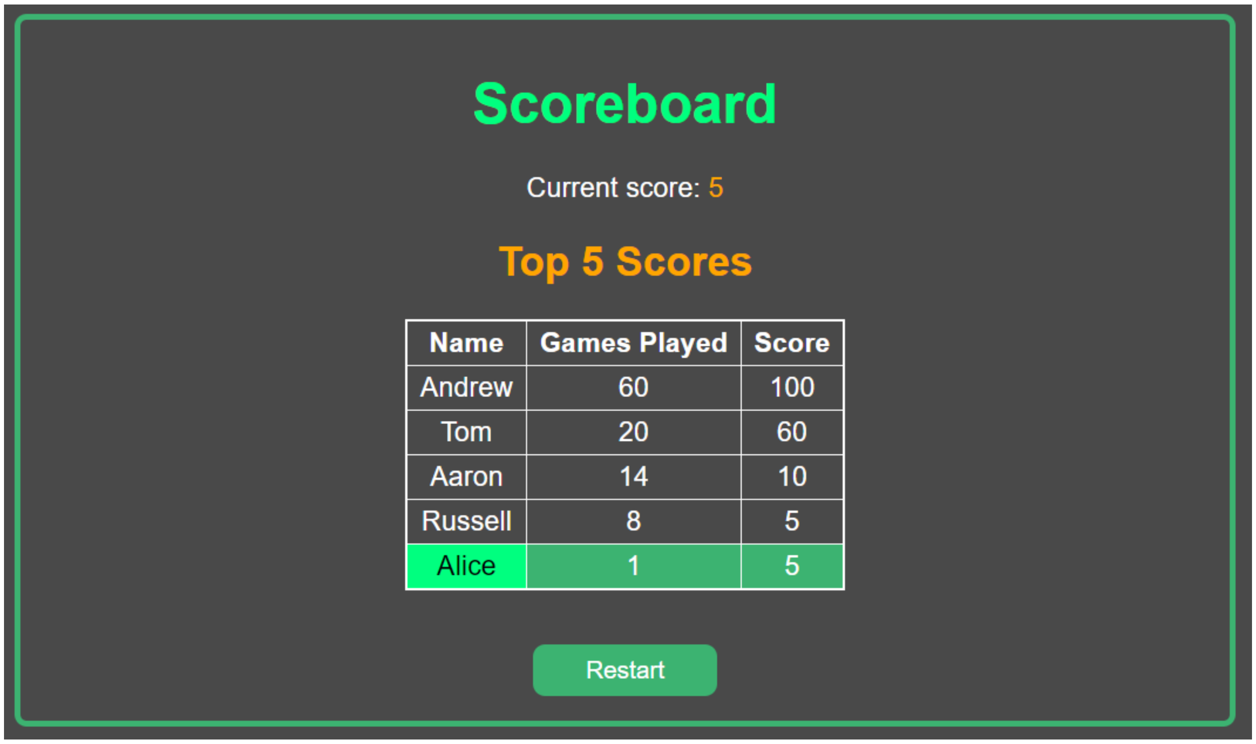
You Won!

You correctly guessed the movie name.

Answer: Wall-E

Current score: 5

We can now see the scoreboard and the player “Alice” is ranked in the top 5 scores.



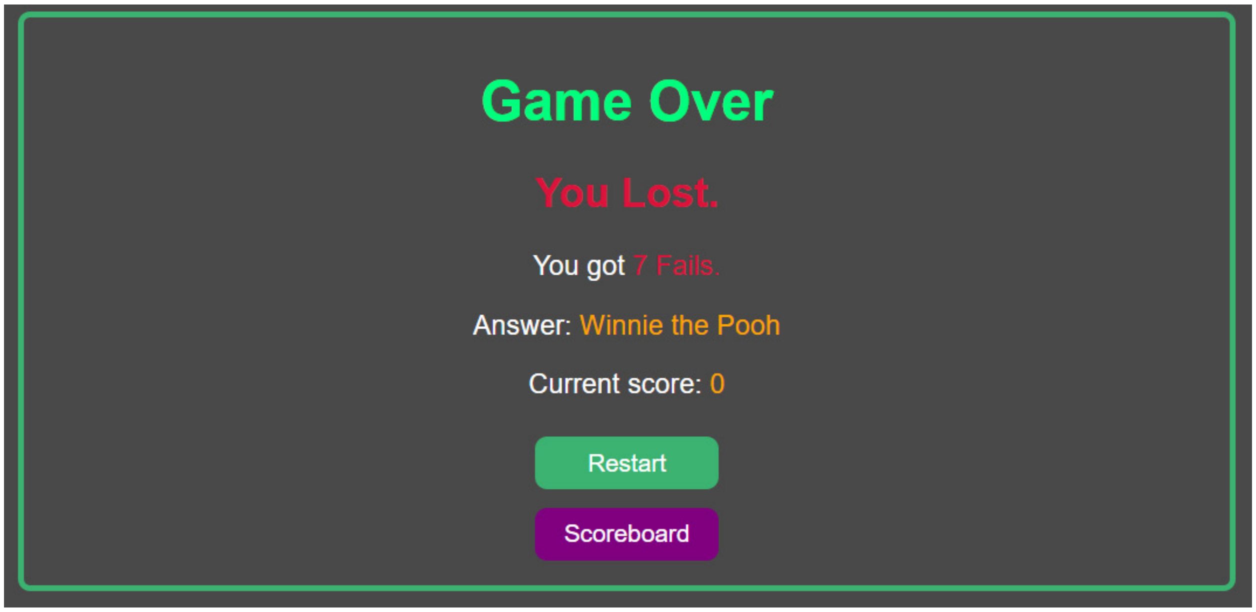
Scoreboard

Current score: 5

Top 5 Scores

Name	Games Played	Score
Andrew	60	100
Tom	20	60
Aaron	14	10
Russell	8	5
Alice	1	5

If the user gained 7 fails, then the user loses the game. The game will show the movie name answer.



Game Over

You Lost.

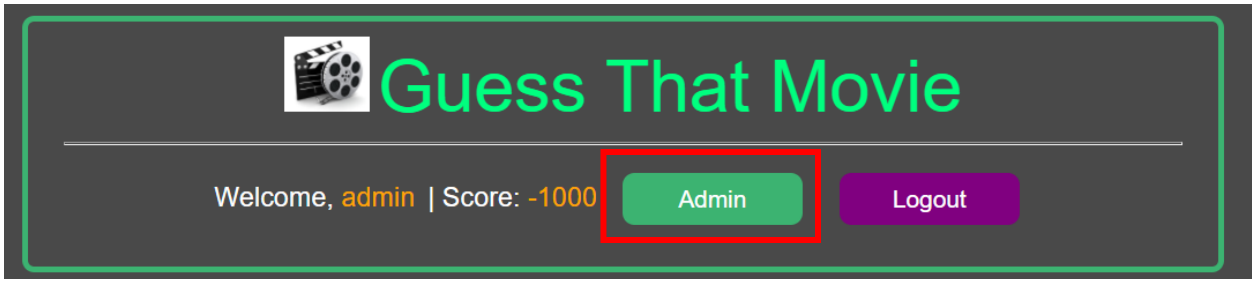
You got 7 Fails.

Answer: Winnie the Pooh

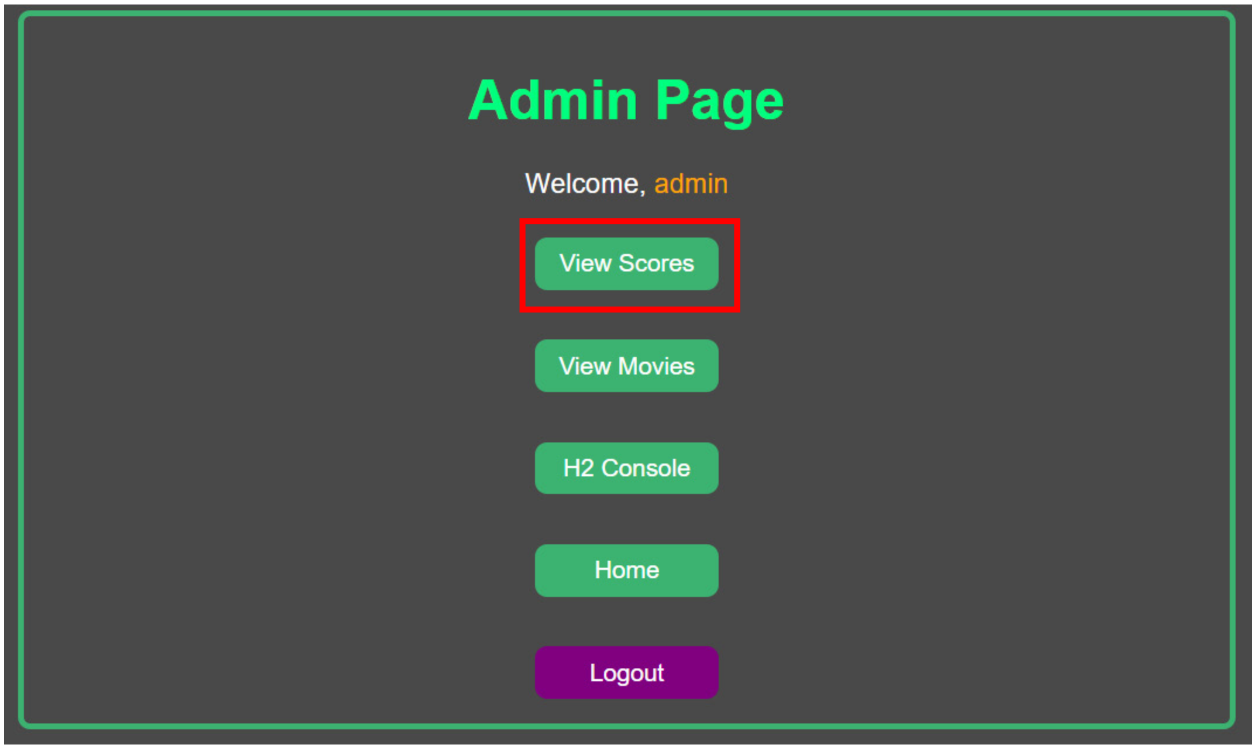
Current score: 0

Admin Features

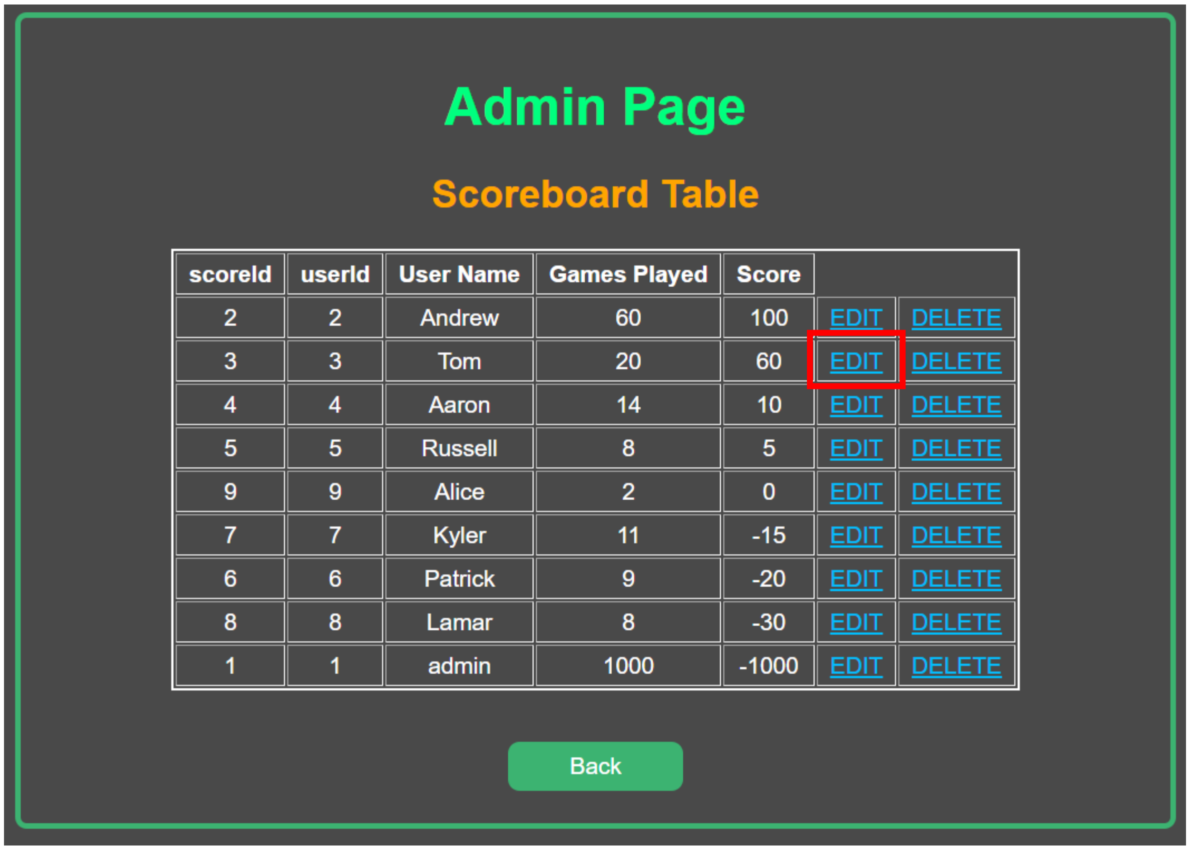
The admin account can access the admin pages and play the game. The Admin button links to the admin pages.



This is the main admin page. It has links to the scoreboard table, movies table, and H2 console.



The admin can see the scoreboard records, sorted by descending score. Let us edit a score record.

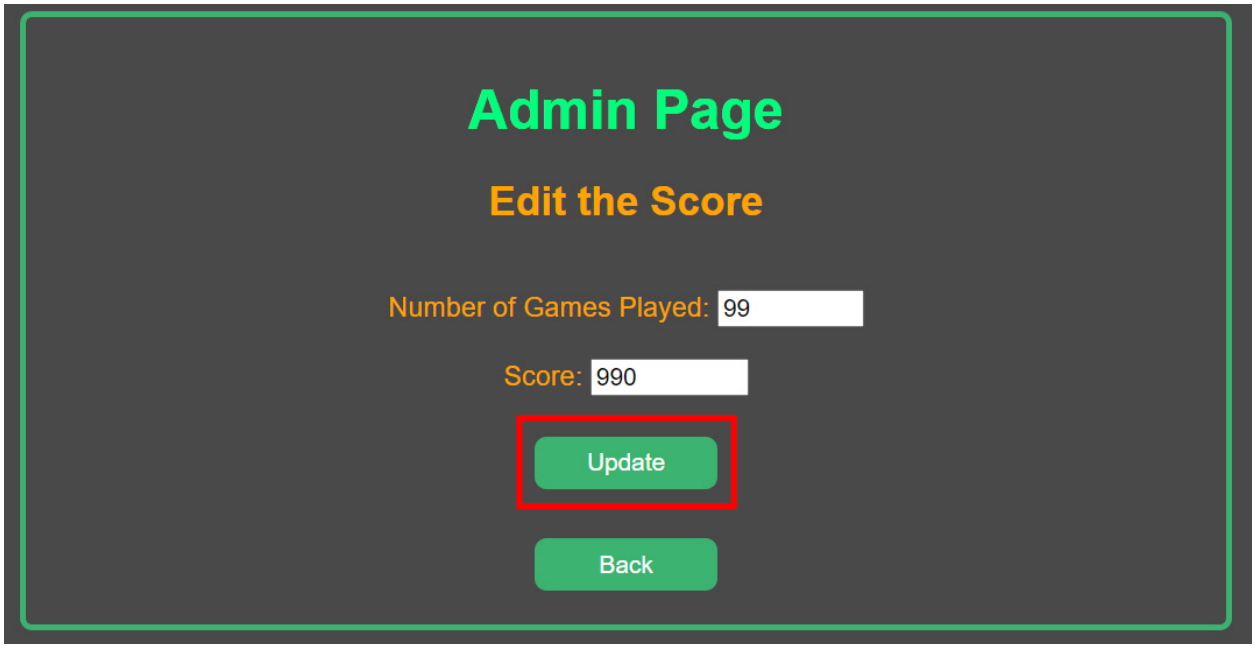


Admin Page

Scoreboard Table

scoreId	userId	User Name	Games Played	Score		
2	2	Andrew	60	100	EDIT	DELETE
3	3	Tom	20	60	EDIT	DELETE
4	4	Aaron	14	10	EDIT	DELETE
5	5	Russell	8	5	EDIT	DELETE
9	9	Alice	2	0	EDIT	DELETE
7	7	Kyler	11	-15	EDIT	DELETE
6	6	Patrick	9	-20	EDIT	DELETE
8	8	Lamar	8	-30	EDIT	DELETE
1	1	admin	1000	-1000	EDIT	DELETE

We can now update the number of games played and the score.



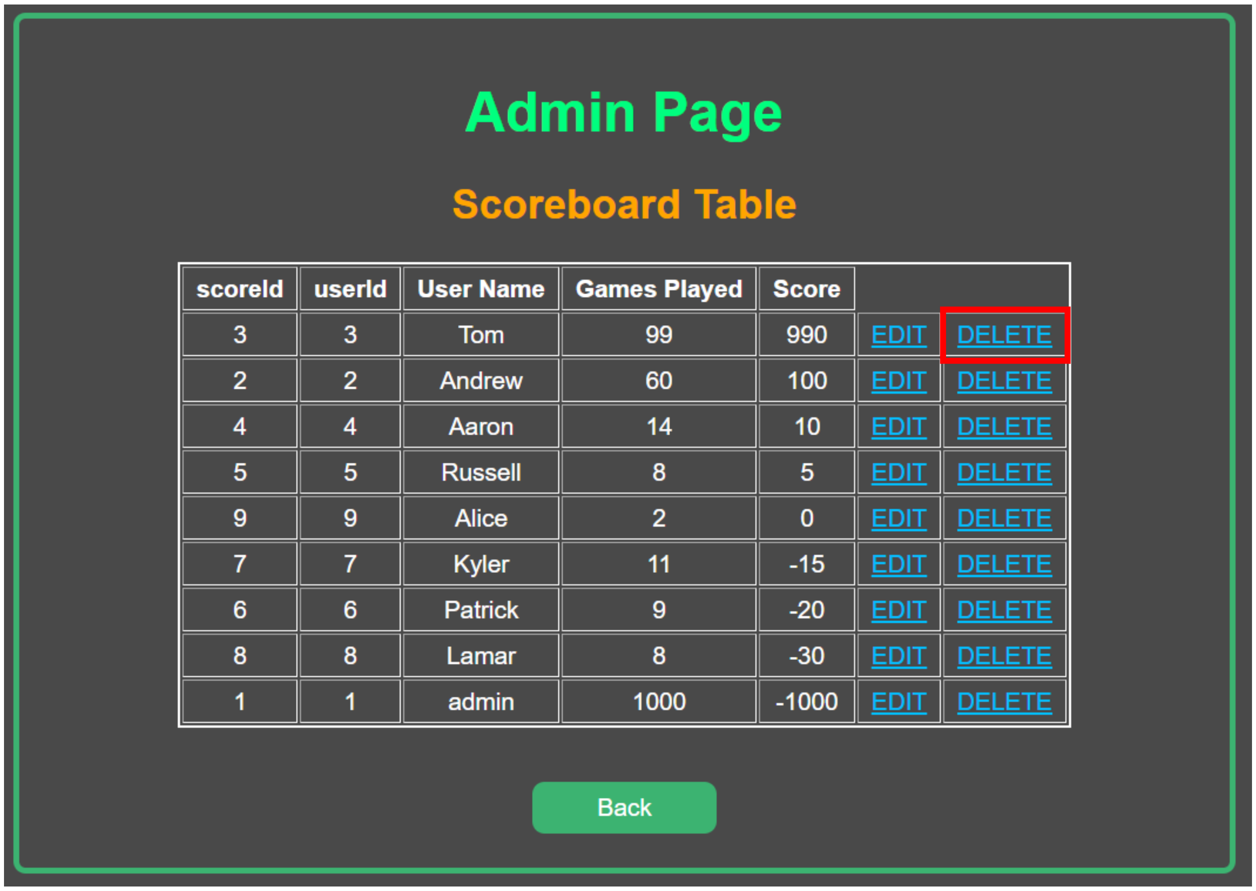
Admin Page

Edit the Score

Number of Games Played: 99

Score: 990

We can see that Tom’s record is updated, and moves to the top. Now let us delete Tom’s record.

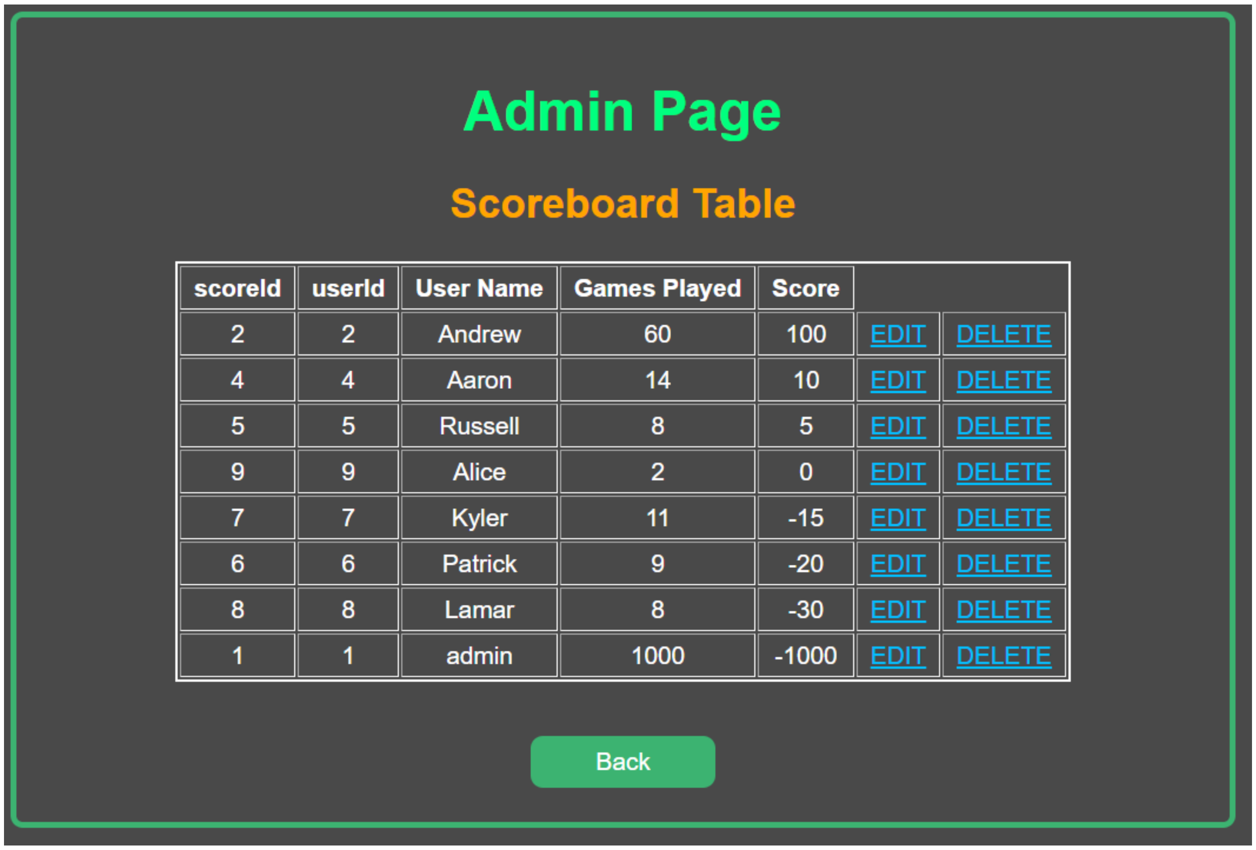


Admin Page

Scoreboard Table

scoreId	userId	User Name	Games Played	Score		
3	3	Tom	99	990	EDIT	DELETE
2	2	Andrew	60	100	EDIT	DELETE
4	4	Aaron	14	10	EDIT	DELETE
5	5	Russell	8	5	EDIT	DELETE
9	9	Alice	2	0	EDIT	DELETE
7	7	Kyler	11	-15	EDIT	DELETE
6	6	Patrick	9	-20	EDIT	DELETE
8	8	Lamar	8	-30	EDIT	DELETE
1	1	admin	1000	-1000	EDIT	DELETE

Tom's record is now deleted.

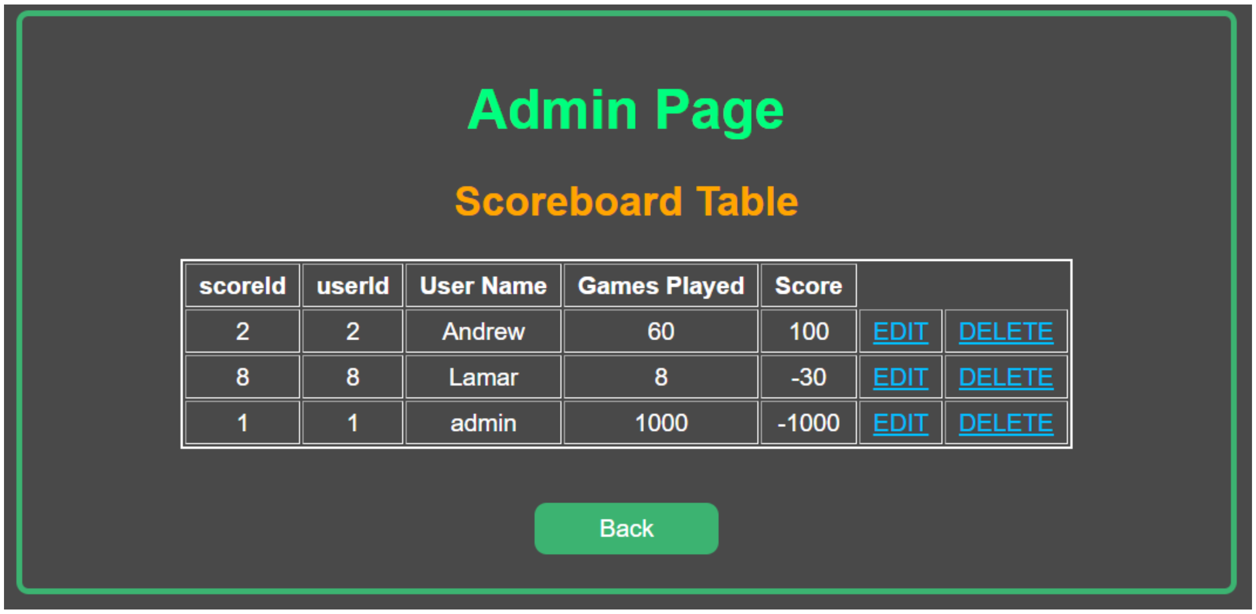


Admin Page

Scoreboard Table

scoreId	userId	User Name	Games Played	Score		
2	2	Andrew	60	100	EDIT	DELETE
4	4	Aaron	14	10	EDIT	DELETE
5	5	Russell	8	5	EDIT	DELETE
9	9	Alice	2	0	EDIT	DELETE
7	7	Kyler	11	-15	EDIT	DELETE
6	6	Patrick	9	-20	EDIT	DELETE
8	8	Lamar	8	-30	EDIT	DELETE
1	1	admin	1000	-1000	EDIT	DELETE

We can delete a lot of score records. The table shrinks.

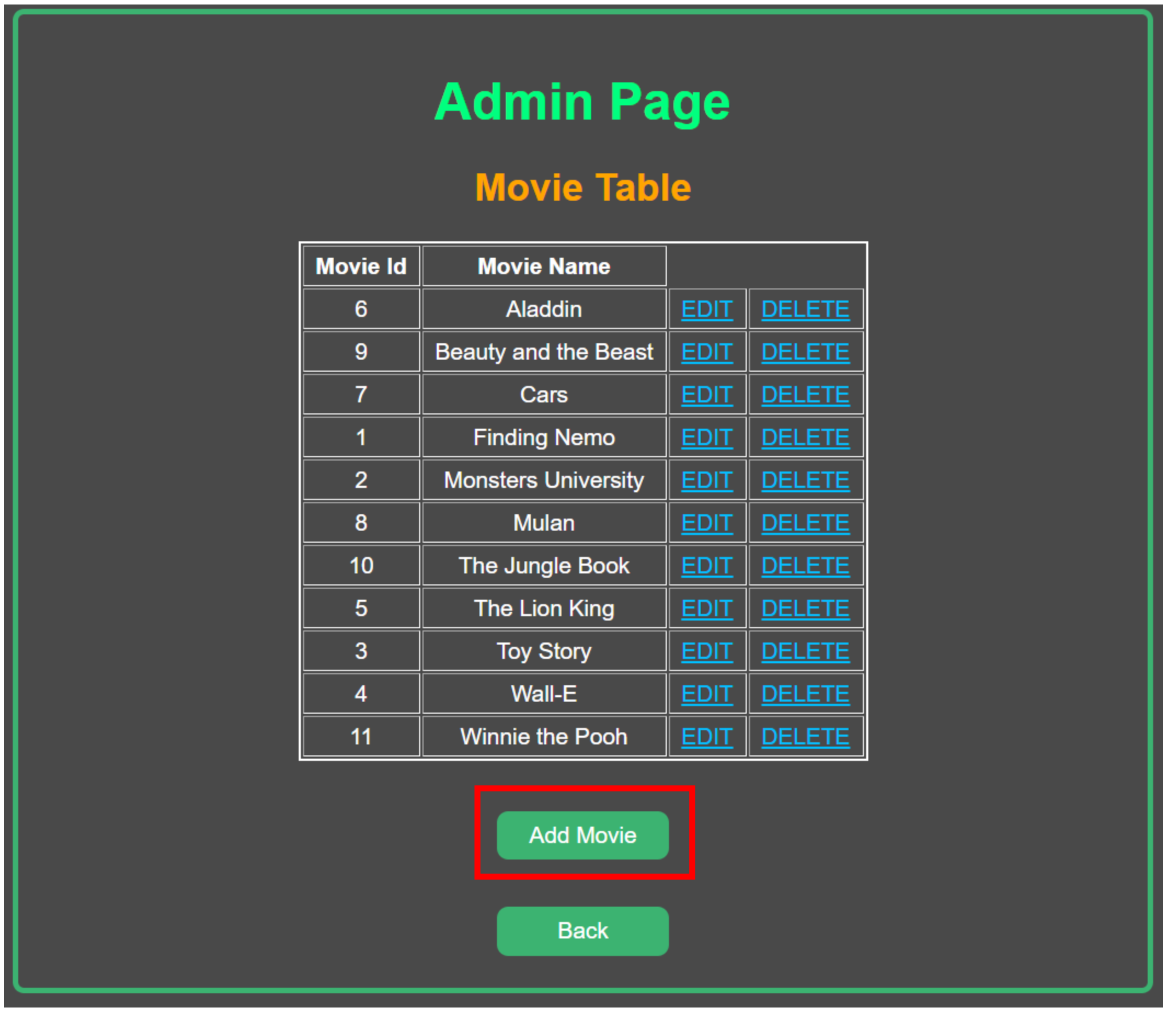


Admin Page

Scoreboard Table

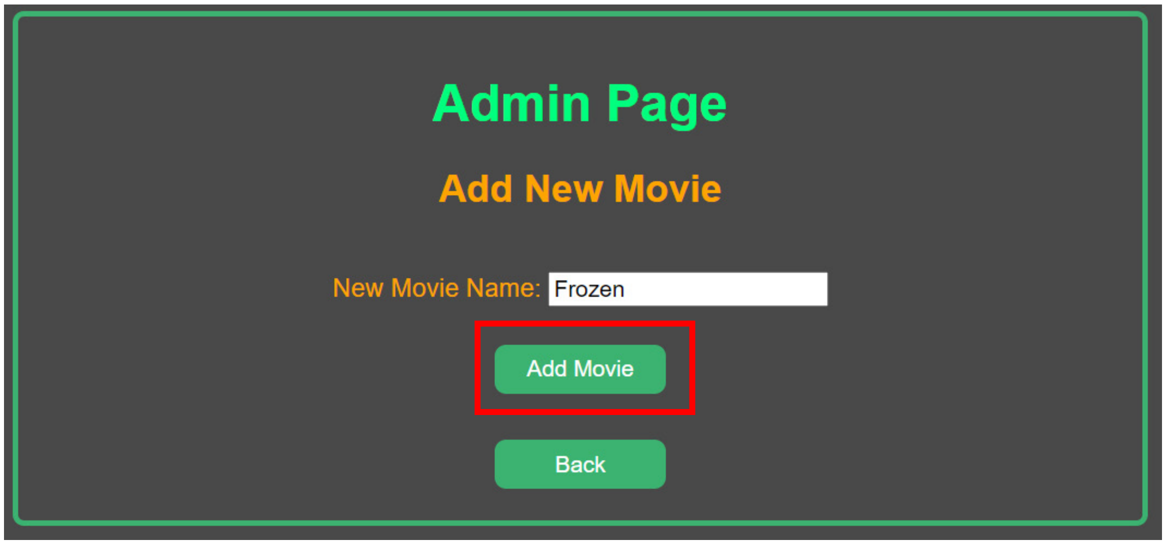
scoreId	userId	User Name	Games Played	Score		
2	2	Andrew	60	100	EDIT	DELETE
8	8	Lamar	8	-30	EDIT	DELETE
1	1	admin	1000	-1000	EDIT	DELETE

On the Movie Table page, we can see the movie list, sorted alphabetically. Let us add a movie.



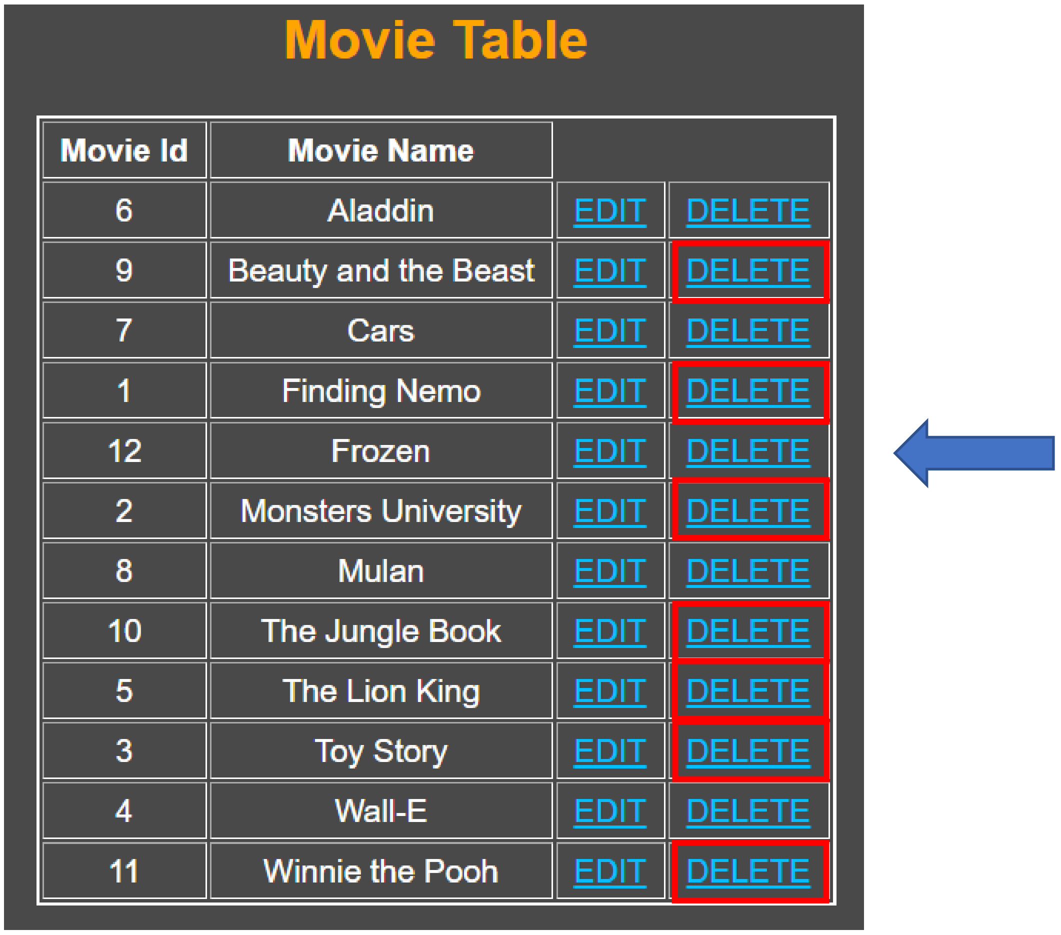
Movie Id	Movie Name		
6	Aladdin	EDIT	DELETE
9	Beauty and the Beast	EDIT	DELETE
7	Cars	EDIT	DELETE
1	Finding Nemo	EDIT	DELETE
2	Monsters University	EDIT	DELETE
8	Mulan	EDIT	DELETE
10	The Jungle Book	EDIT	DELETE
5	The Lion King	EDIT	DELETE
3	Toy Story	EDIT	DELETE
4	Wall-E	EDIT	DELETE
11	Winnie the Pooh	EDIT	DELETE

Let us add the movie Frozen.



New Movie Name: Frozen

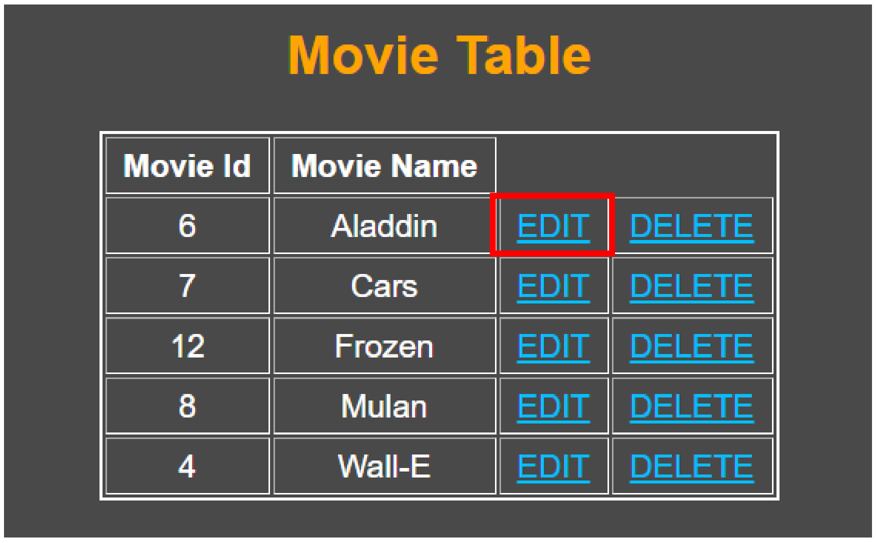
The new movie Frozen is now added to the list. Now, let us delete movies with long names.



Movie Table

Movie Id	Movie Name		
6	Aladdin	EDIT	DELETE
9	Beauty and the Beast	EDIT	DELETE
7	Cars	EDIT	DELETE
1	Finding Nemo	EDIT	DELETE
12	Frozen	EDIT	DELETE
2	Monsters University	EDIT	DELETE
8	Mulan	EDIT	DELETE
10	The Jungle Book	EDIT	DELETE
5	The Lion King	EDIT	DELETE
3	Toy Story	EDIT	DELETE
4	Wall-E	EDIT	DELETE
11	Winnie the Pooh	EDIT	DELETE

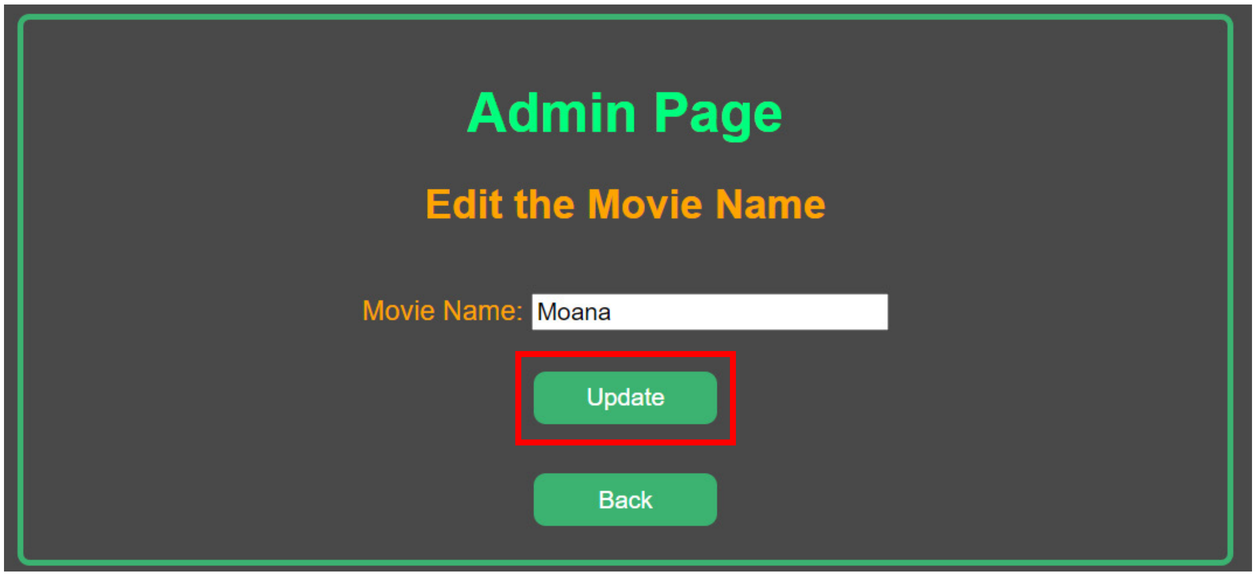
Now the movie table is smaller. We can edit the name of a movie.



Movie Table

Movie Id	Movie Name		
6	Aladdin	EDIT	DELETE
7	Cars	EDIT	DELETE
12	Frozen	EDIT	DELETE
8	Mulan	EDIT	DELETE
4	Wall-E	EDIT	DELETE

We can change the movie name to “Moana”.

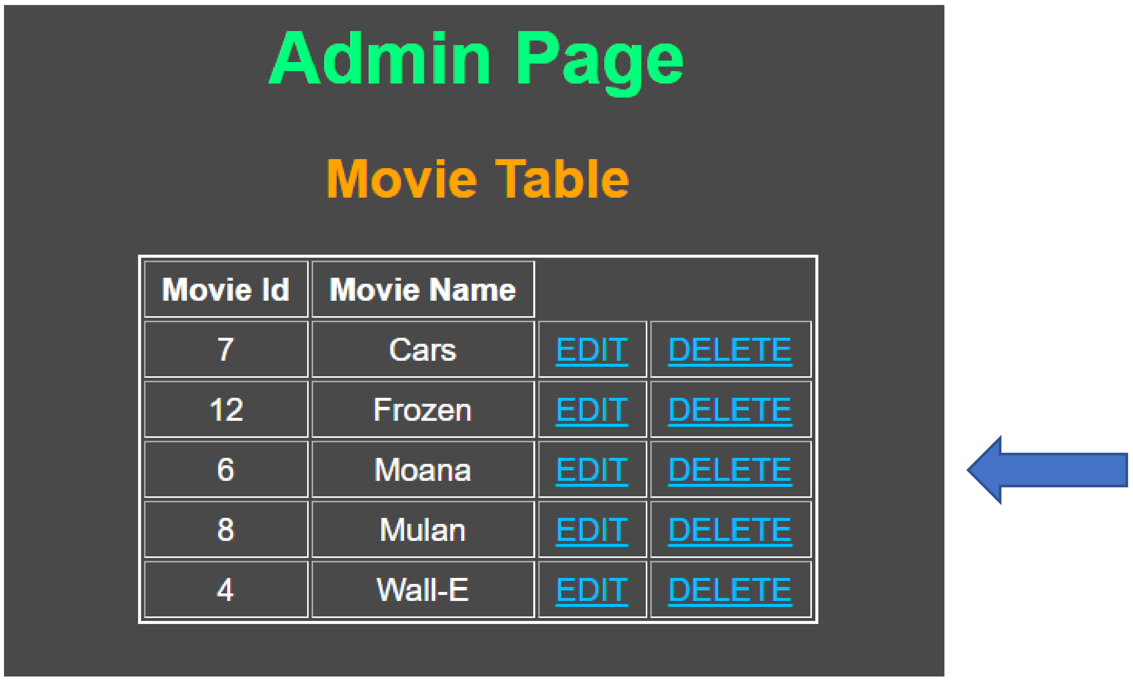


Admin Page

Edit the Movie Name

Movie Name: Moana

The movie name “Aladdin” is now updated to “Moana”.



Admin Page

Movie Table

Movie Id	Movie Name		
7	Cars	EDIT	DELETE
12	Frozen	EDIT	DELETE
6	Moana	EDIT	DELETE
8	Mulan	EDIT	DELETE
4	Wall-E	EDIT	DELETE

Database Design

The database has five tables:

1. **sec_user:**	stores the userId, username, and encrypted password.
2. **sec_role:**	stores the authentication roles of “ROLE_USER” and “ROLE_ADMIN”.
3. **user_role:**	an intersection table between sec_user and sec_role.
4. **scoreboard:**	stores the userId, number of games played, and the user’s score.
5. **movie:**	a basic table that only stores movie names.

Foreign keys were used in the design. For the scoreboard table, there is a foreign key called userId which references the primary key in the sec_user table. This associates users with their score. Also, we can join the two tables **to get the username for a scoreboard record**. I wanted to show the username with its score when displaying the in-game scoreboard.

Foreign keys were also used in the **user_role intersection table**. Since a user can have a one-to-many relationship with roles, the intersection table connects the sec_user and sec_role tables. The user_role table has **one concatenated primary key, which is made up of two foreign keys**: userId from the sec_user table and roleId from the sec_role table.

All the users were stored in the sec_user table. I did not separate the regular users and admin accounts into separate tables. **We can already distinguish the account types by roles** (with the user_role intersection table). So there was no critical need to store admin accounts separately. Plus, I wanted to store the user credentials in a simple and centralized table for easier maintenance and management.

SQL Injection Prevention

We can prevent SQL injection attacks by using **prepared queries and named parameters** in our database access methods.

If the input values were directly inserted the SQL queries by string concatenation, then the application is vulnerable to SQL injection attacks. The application is tricked into processing malicious SQL queries. Attackers can input into forms some SQL code that reveal sensitive data, such as passwords or emails. Or a table can be ordered to delete all its records.

Prepared SQL queries using named parameters prevents these attacks. My program **indirectly loads the input values into named parameters**. The named parameter *values* are stored in a Map container. The SQL query string also has references (ex. “:userId”) which are the Map *keys* of the key-value pair.

In fact, almost all my database access methods use named parameters with the SQL queries. This design choice offers a lot of flexibility, since it is easy to load multiple values into the MapSqlParameterSource container called parameters. We then feed the SQL query and the parameters into the NamedParameterJdbcTemplate called jdbc to query the database. Thus, my application gains greater flexibility and security.

Code Design

Beans

I created three beans: Movie, ScoreRow, and User. These classes model data entities for several database tables – the movie, scoreboard, and sec_user tables. **The beans act as a data container for the respective SQL table records.**

As such, the Java beans contain data fields that match up with the table fields of their counterparts. When I use database access methods to get table records, I store the records (or rows) into the appropriate bean. Then I can use the list of such beans (ex. ArrayList<ScoreRow>) to populate the scoreboard. Or I can randomly select a movie name from the list of Movie beans (ex. ArrayList<Movie>).

SOLID: Single Responsibility Principle

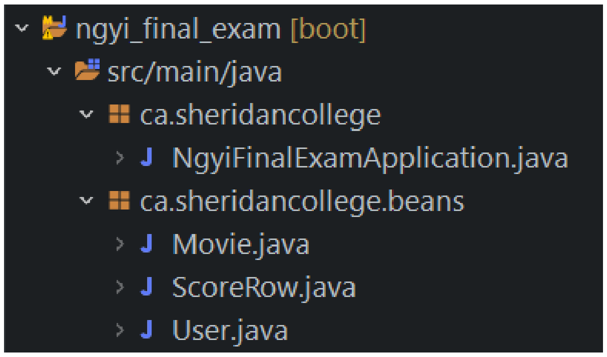
In SOLID, the single responsibility principle (SRP) says that a class should have "one and only one reason to change". It calls for high cohesion within a class, where all the attributes and methods are highly related in purpose and responsibility.

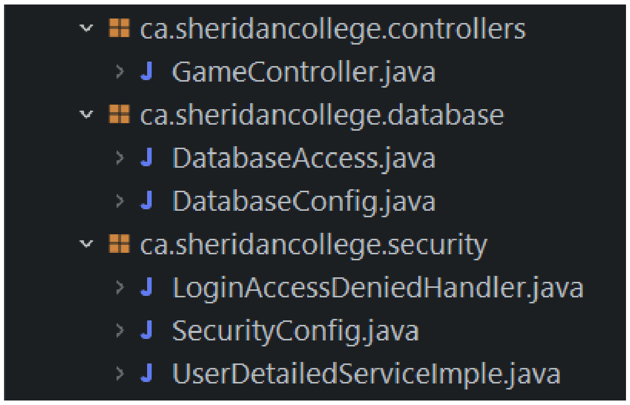
I believe that I have followed the SRP by diligently organizing my classes, packages, and files.

My classes are strictly divided by function and responsibility. I did not combine code altogether into large "God classes". For example, the Movie bean class only contains data fields and methods that relate to a movie or its movie name string. Next, the SecurityConfig class is solely responsible for user authentication and URL access control. The DatabaseAccess class is solely responsible for querying the database. Finally, the GameController class directs the program flow and processes the game logic.

On a higher level, I have **grouped related classes into packages**. For example, the ca.sheridancollege.beans package holds all my beans. Also, the ca.sheridancollege.database package contains the DatabaseAccess and DatabaseConfig classes.

<u>Package organization:</u>


```
ngyi_final_exam [boot]
  src/main/java
    ca.sheridancollege
      NgyiFinalExamApplication.java
    ca.sheridancollege.beans
      Movie.java
      ScoreRow.java
      User.java
```


```
ca.sheridancollege.controllers
  GameController.java
ca.sheridancollege.database
  DatabaseAccess.java
  DatabaseConfig.java
ca.sheridancollege.security
  LoginAccessDeniedHandler.java
  SecurityConfig.java
  UserDetailedServiceImple.java
```

Sessions

Sessions are great for **storing data across multiple HTTP requests** (that is, moving through multiple URLs and pages). Sessions allow the game program and the player user to carry along and update game data.

Using dependency injection, I **injected an HttpSession session object** into the GameController methods. This allowed me to get data from the session, such as the number of fails. I can also update or set values into the session object, such as increasing the score after a player wins.

I used sessions to track game data: the number of fails that a player has, the randomly chosen Movie object, the player's score, and the hidden movie name string.

- **Fails:** used to **track the number of fails** that a player has. When the player loses a guess, the program directs the player back to the "Guess A Letter" stage. This navigation flow looping can occur several times (up to 7 fails for a loss). Since the player can loop through multiple HTTP requests, I stored the fails value in the session to track it when the player moves through the game.
- **Movie object:** the target movie name was randomly chosen from the start. I needed to **save the movie name answer through multiple pages**, so I stored the Movie bean object in the session.
- **Score:** for logged-in users, the **score is always shown in the menu bar** Thymeleaf fragment. I needed to store the score across all the game pages, since the menu bar shows up everywhere in the game.
- **Hidden movie string: helps the player to guess the movie.** The spacing of the words is shown, while the letters are scrambled. I showed the hidden string in two different pages, so I stored it in the session. **Example: "The Lion King" = "***_****_****"**

Spring Security

With Database

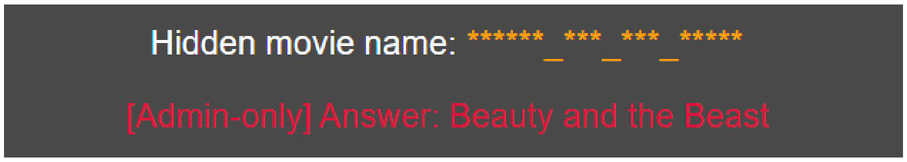
The database stores user credentials in the sec_user table. The passwords are stored as salted encrypted hashes, instead of storing passwords as plaintext. This enhances the security level, since only the scrambled hashes are exposed if the database is breached.

Roles (ROLE_USER and ROLE_ADMIN) are assigned to users with the user_role intersection table.

With Thymeleaf

Thymeleaf security tags were used extensively to customize content by authentication status and role.

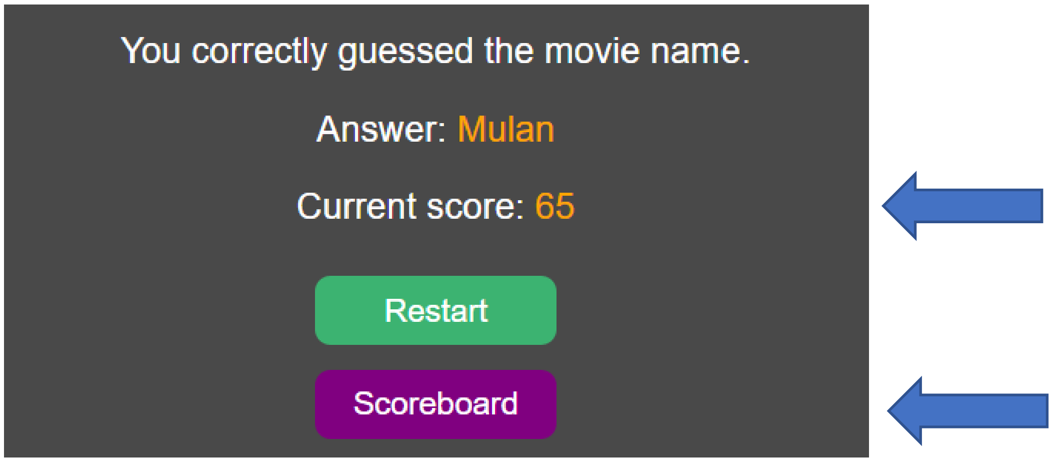
For the guessLetter.html and guessMovie.html pages, security tags display the movie name answer to admins only (by role). This is used for game testing and debugging.



Hidden movie name: ******_***_***_*****

[Admin-only] Answer: Beauty and the Beast

At the end of a game, the player's score and the Scoreboard button are only displayed for authenticated (logged-in) users. Non-registered players cannot see their score or the button.



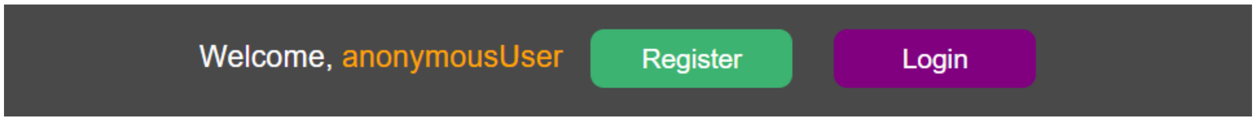
You correctly guessed the movie name.

Answer: Mulan

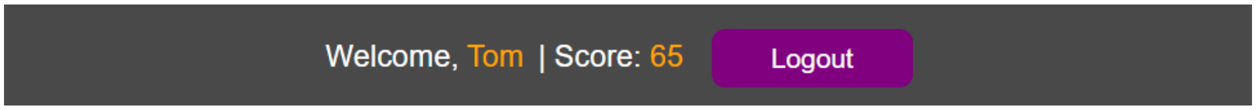
Current score: 65

In the menu bar Thymeleaf fragment, security tags are used to customize the username, score, and login buttons. **The menu bar changes depending on the authentication and role.**

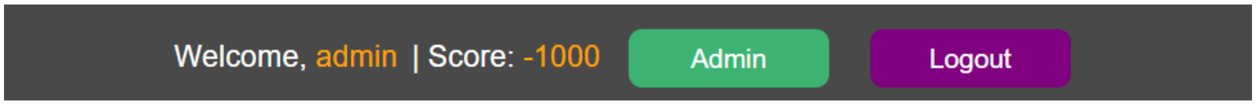
1. Anonymous non-registered players can see the **Register and Login buttons**.



2. Registered players can see the **username, score, and the Logout button**.

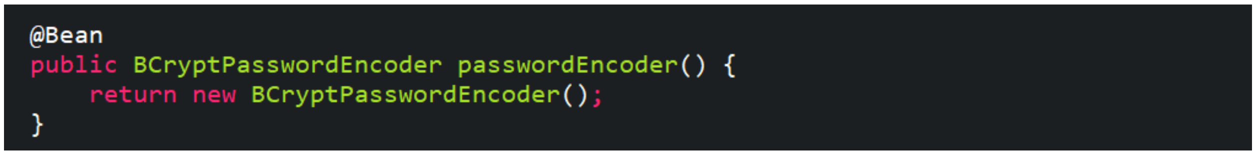


3. Admins can see the **username, score, Admin button, and Logout button**.



With Java

User credentials are handled using a BCryptPasswordEncoder object. Passwords are encrypted and decrypted using the BCrypt hash function. During registration, the password encoder salts and hashes the password. This enhances the security level since identical plaintext password can result in different password hashes. This protects passwords from cracking by rainbow tables and dictionary attacks.



```
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

In the SecurityConfig class, the configure() method is used to control access to resources. AntMatchers are used to identify URLs, to permit or restrict access. Static and gameplay resources are permitted for “public” access by any user. The in-game top 5 scoreboard is restricted to registered users with the role (ROLE_USER). Admin pages and the H2 console are restricted to admins with the role (ROLE_ADMIN).

The H2 console was enabled, for the convenience of the admin. However, I do understand that the CSRF protections should not be disabled in production code.

The **login, registration, and logout requests** are also handled by the SecurityConfig class.

User credential and role details are retrieved from the database using the UserDetailedServiceImple object.

The admin-related resources (/admin/** and /h2-console/**) are restricted to admin users with the (ROLE_ADMIN) role. **Should regular users attempt to access the restricted resources, the program will deny access** with the LoginAccessDeniedHandler object.

CRUD Operations

The admin can manage the database with CRUD operations. All SQL queries were handled by the DatabaseAccess class. Please refer to the section Navigation -> Admin Features.

Create

The admin can **add a new movie to the movie table**.

A new user is created through the account registration process (via the Register button).

Read

The admin can **view the scoreboard records**, which also includes the usernames. Also, the **list of movies can be viewed**, in alphabetical order.

Admins can view the user table via the H2 console. I am simulating a segregated user repository by separating the sec_user table from the admin pages (/admin/**).

Update

The admin can **edit a movie's name** in the movie table.

In the scoreboard table, the admin can **edit the number of games played and the score** for a player record.

Delete

The admin can **delete a movie record or a scoreboard table record**.

Deleting a scoreboard record does not delete the username or user record in the sec_user table.

For referential integrity purposes, I did not allow for deleting user rows from the sec_user table. This is because the sec_user.userId primary key is used as a foreign key in the scoreboard table.