

Full-Stack Web Application Development using Springboot, MySQL and JavaScript/jQuery frontend

SUBMISSION: 17/03/2025

A00325390: EOIN LAWLESS

Table of Contents

1	Introduction.....	2
	Overview	2
	Context & Rationale	2
2	System Design & User Stories	2
	User Stories	2
	Entity-Relationship Diagram (ERD)	7
3	Backend – REST API Development	8
	Security Implementation	8
	DTO's and how they are used	8
	Adherence to REST API Design Principles	9
	HATEOAS Implementation	10
	Error Handling & Validation	11
4	Database Schema.....	12
5	Frontend Development.....	13
	Technologies Used	13
	Key Features	15
	Graphical Statistics and Analytics	16
	Game Development	17
6	Testing & Error Handling.....	18
	Testing Approach	18
7	Conclusion & Future Work	20
	Key Learnings.....	20
	Limitations	21
	Possible Improvements	21
	Scalability Issues & Options.....	21
	Bibliography	22
	Link to Github: https://github.com/EoinieLawless/WebTechProject	23

1 Introduction

Overview

This project is a **full-stack web application** designed using:

- **Backend:** Spring Boot (REST API, HATEOAS, JWT Authentication)
- **Frontend:** Vue.js JavaScript, jQuery, Bootstrap, DataTables
- **Database:** MySQL (Spring Data JPA)
- **Security:** Role-based access control (RBAC) with JWT authentication
- **Best Practices:** REST API principles, error handling, validation

Context & Rationale

The purpose of my project (Lawless Gaming) is to build a secure, scalable, and user-friendly system that allows users/admins to register, submit scores, view leaderboards, and handle complaints. This aligns with industry standards in REST API development, frontend interactivity, and database design. (GitHub Repository, n.d.) (Spring Boot, n.d.) (Vue.js, n.d.)

2 System Design & User Stories

User Stories

All the Users stories were written into Jira, all were completed.

1. User Registration & Login

As a user, I want to register and log in so that I can access my personal game data.

Acceptance Criteria

- Given a new user provides a **unique username, email, and password**, when they submit the registration form, then the system should **create their account** and redirect them to the login page.

- Given a user tries to **register with an existing email**, when they submit the form, then the system should **reject the registration** and display an appropriate error message.
- Given a user enters an **invalid email format or weak password**, when they submit the form, then the system should display **validation errors**.

2. Submitting Game Scores

As a user, I want to submit my game scores so that I can compete on the leaderboard.

Acceptance Criteria

- Given I finish a game, when I submit my **score**, then it should be **stored in the database**.
- Given I have submitted a score, when I navigate to the **leaderboard**, I should be able to see my rank.
- Relevant **unit, integration tests** are completed.

3. Playing a Variety of Games

As a user, I want to play different types of games so that I can test my skill range.

Acceptance Criteria

- Given I am on the **Home Page**, I should be able to **view all available games**.
- Given I start a game, when I interact with it, then the **game should function properly**.
- Given I complete a game session, when the game ends, then I should **receive a score** based on my performance.

4. Viewing the Leaderboard

As a user, I want to see the leaderboard so that I can view the top players.

Acceptance Criteria

- Given I am on the website, when I navigate to the **leaderboard page**, then I should see a **ranked list of players** based on their scores.
- Given I am on the leaderboard, when I select a **specific game**, then I should see the **highest scores** for that game.
- Given I am viewing the leaderboard, when I select "**Most Active Players**," then I should see a ranking based on **total playtime or number of games played**.

5. Submitting User Complaints

As a user, I want to submit complaints so that I can report any issues or concerns to the admin.

Acceptance Criteria

- Given I am logged in as a user, when I navigate to the **complaints page** and submit a complaint, then my **name and email should be attached** automatically.
- Given I am an **admin**, when I navigate to the **complaints section**, then I should be able to **view all user complaints**.
- Given I submit a complaint, then I should **not be able to edit the sender details** (name and email).

6. Viewing Personal Game Stats

As a user, I want to view my **personal statistics** so that I can track my gaming habits.

Acceptance Criteria

- Given I am logged in, when I navigate to the **personal stats page**, then I should see **my most played game**.
- Given I am on the personal stats page, when I check my statistics, then I should see **at least two graphical charts**.
- Given I am on the personal stats page, when I check my ranking, then I should see my **current position on the global leaderboard**.

7. Admin Managing Users

As an admin, I want to **view and manage user accounts** so that I can control who has access to the platform.

Acceptance Criteria

- Given I am logged in as an **admin**, when I navigate to the **user management page**, then I should see a **list of all registered users**.
- Given I am on the user management page, when I **create a new user**, then they should be **assigned a role (User/Admin)**.
- Given a user exists in the system, when I **click delete** next to their name, then I should be prompted with a **confirmation modal** before proceeding.
- Given a user has the **"ADMIN" role**, when I attempt to delete them, then I should **receive an error message** preventing deletion.

8. Saving & Tracking Game Progress

As a user, I want to have my **game progress saved** so that I can track my performance.

Acceptance Criteria

- Given I am playing a game, when I **complete a session**, then my **game data should be saved**.
- Given the website has different **game categories (Precision, Puzzle, and Luck)**, when I play, then my **progress should be saved under the respective category**.
- Given I revisit the website **after logging in**, when I check my **game stats**, then I should see my **previous scores**.

9. Admin Importing Datasets

As an admin, I want to **import historical game data** so that I can maintain accurate records.

Acceptance Criteria

- Given I am logged in as an **admin**, when I navigate to the **import page**, then I should be able to **download all data files**.
- Given I am on the **import page**, when I start an import, if the system detects **existing data**, then it should **skip duplicates**.
- Given I complete a data import, when I navigate to the **leaderboard page**, then I should be able to see **historical game scores**.

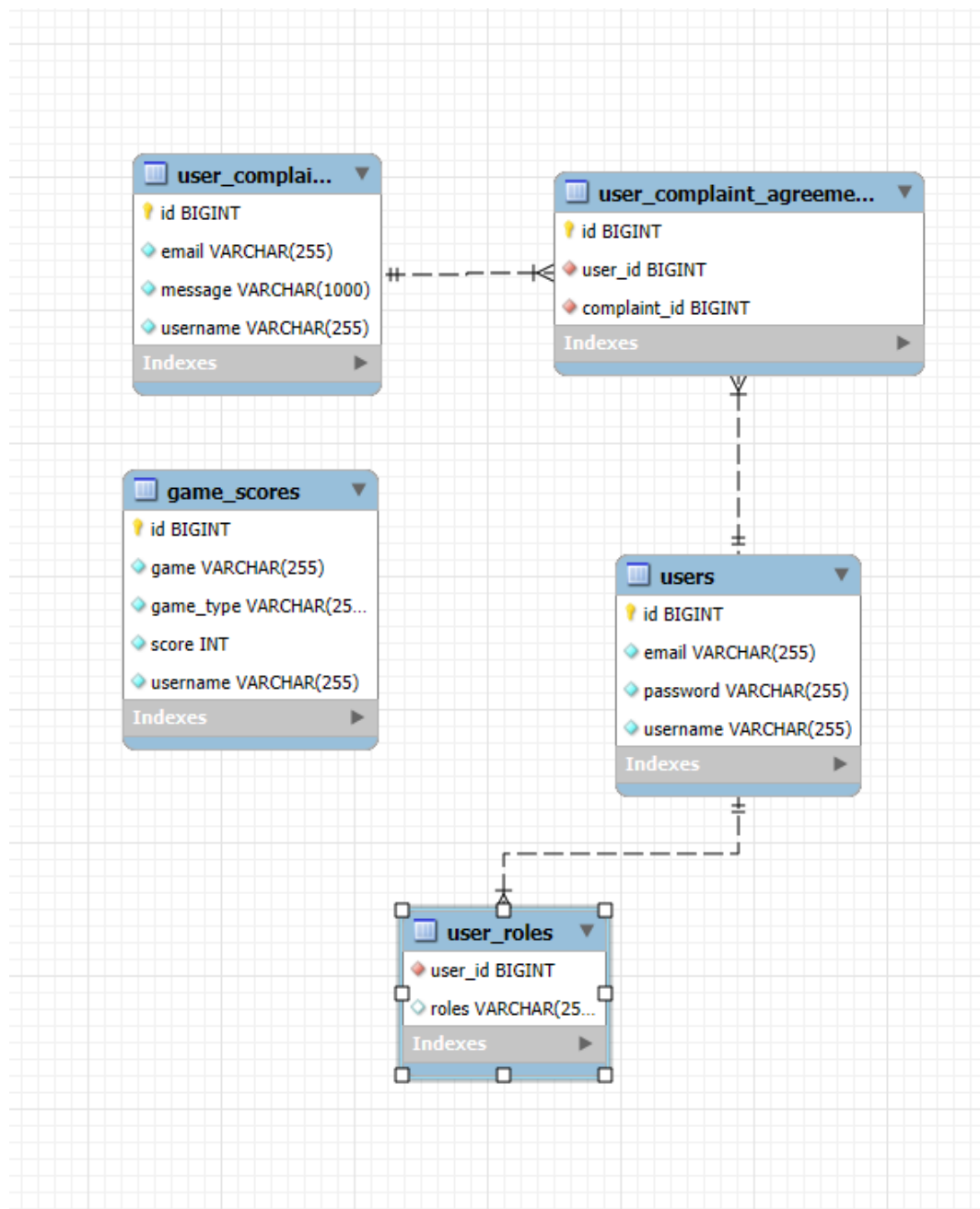
10. Complaint User Agreement

As a user, I want to submit a complaint so that I can report any issues or concerns to the admin.

Acceptance Criteria:

- Given I am logged in, when I navigate to the complaint form and fill in the complaint message, then the system should automatically attach my name to the complaint I agree with.
- Given I am logged in, when I navigate to the complaint form, I can see the names attached to the complaint, then I know who agrees with the complaint.

Entity-Relationship Diagram (ERD)



(Diagram showcasing relationships between User, GameScore, Leaderboard, and UserComplaint)

Backend – REST API Development

Security Implementation

- **JWT-based authentication** for secure access.

When a user logs in (handled in my `AuthController.java`), I verify their username and password using Spring Security's `AuthenticationManager`. Once verified, I use my `JwtUtil.java` class to generate a JWT that includes the user's name and roles, and this token is set to expire in one hour. Every time the user makes a request, the `JwtRequestFilter.java` intercepts it, extracts the token from the header, validates it, and then sets up the security context. This way, I ensure that each request comes from an authenticated user.

- **Role-based access control (RBAC)** to ensure proper user privileges.

In my `SecurityConfig.java`, I clearly define which endpoints require specific roles. For instance, endpoints under `/components/admin/**` are restricted to users with the ADMIN role, while `/components/user/**` are for regular users. I also use method-level security annotations (like `@PreAuthorize` in `AdminController.java`) to enforce these rules further. Additionally, during user registration in `AuthController.java` and `UserService.java`, new users are automatically assigned the USER role, while an initial admin account is created in `DataInitializer.java`.

DTO's and how they are used

I use DTOs to keep the internal data model separate from what the API exposes. For example, I created a `GameScoreDTO`

```

1 package com.tus.GamingSite.gameScore.dto;
2
3 import lombok.Getter;
4
5
6 @Getter
7 @Setter
8 public class GameScoreDTO extends RepresentationModel<GameScoreDTO> {
9     private String username;
10    private String game;
11    private int score;
12    private String gameType;
13
14    public GameScoreDTO() {}
15
16    public GameScoreDTO(String username, String game, int score, String gameType) {
17        this.username = username;
18        this.game = game;
19        this.score = score;
20        this.gameType = gameType;
21    }
22 }
23
24

```

(see GameScoreDTO.java) that only contains the username, game, score, and game type.

This way, my GameScore entity (see GameScore.java) remains hidden, and any changes in the database don't directly affect the API.

I perform the conversion in the GameScoreService (see GameScoreService.java), which gives me flexibility and easier maintenance.

Adherence to REST API Design Principles

I've designed the API to follow RESTful principles at a Level 3 maturity according to the Richardson Model. Here's how I do it:

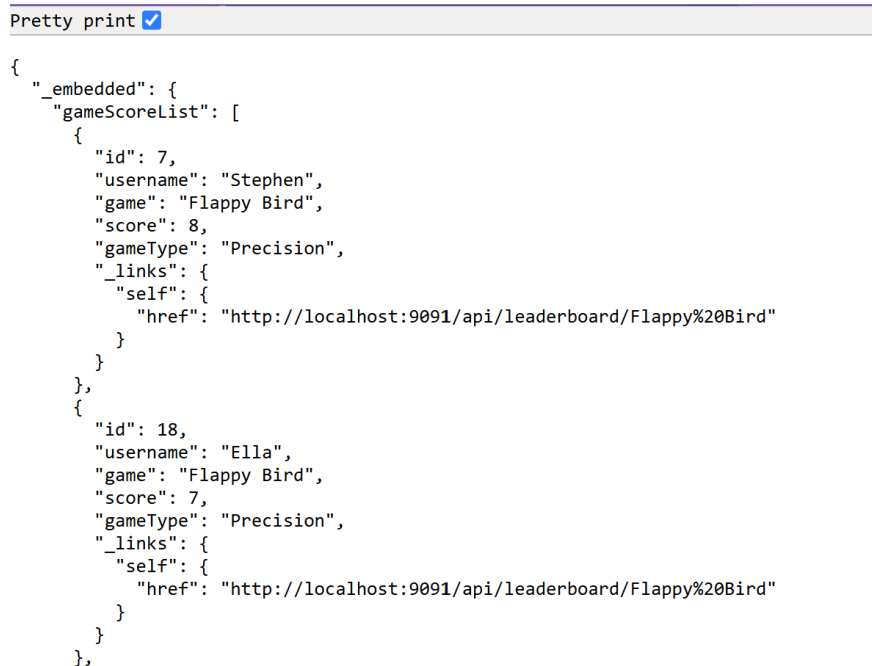
- **Clear Resource Identification:**
Every resource has a unique URI, like /api/games or /api/auth/login.
- **Use of HTTP Methods:**
I use standard HTTP methods (GET, POST, DELETE, etc.) to operate on resources. For example, saving a game score is done with a POST request in my GameScoreController.java.
- **Hypermedia as the Engine of Application State (HATEOAS):**
I wrap my responses in DTOs that extend Spring's RepresentationModel, allowing me to include hypermedia links. This means clients can discover related actions or resources dynamically.

In my `GameScoreController`, after saving a score, I add links (like "game-scores" and "user-scores") to the response, which is a key characteristic of Level 3 REST APIs.

HATEOAS Implementation

The backend, built with **Spring Boot** and **Spring HATEOAS**, attaches **hypermedia links** to API responses. This allows the frontend to dynamically discover actions without needing hardcoded URLs.

Example from the Leaderboard API



```

Pretty print ☒
{
  "_embedded": {
    "gameScoreList": [
      {
        "id": 7,
        "username": "Stephen",
        "game": "Flappy Bird",
        "score": 8,
        "gameType": "Precision",
        "_links": {
          "self": {
            "href": "http://localhost:9091/api/leaderboard/Flappy%20Bird"
          }
        }
      },
      {
        "id": 18,
        "username": "Ella",
        "game": "Flappy Bird",
        "score": 7,
        "gameType": "Precision",
        "_links": {
          "self": {
            "href": "http://localhost:9091/api/leaderboard/Flappy%20Bird"
          }
        }
      }
    ]
  }
}
```

The image above shows an API response for the **leaderboard of "Flappy Bird"**, returning scores from different users. Each game score entry contains:

- **Username and Score** – Basic game data.
- **Game Type** – The type of skill tested (e.g., "Precision").
- **_links Property** – A "self" links that points to the specific leaderboard entry.

This follows **HATEOAS principles**, where the API itself provides the relevant actions (e.g., retrieving specific leaderboard data) without requiring the frontend to construct URLs manually.

Explanation

1. HATEOAS on the Backend:

As outlined in my project report, my backend (using Spring Boot and Spring HATEOAS) attaches hypermedia links to responses. For example, when a complaint is submitted or fetched, its DTO (which extends `RepresentationModel`) contains a `_links` property with links like "self" and "delete-complaint" (see `UserComplaintController.java` and `UserComplaintDTO.java`).

2. Fetching Complaints:

The `fetchComplaints()` method retrieves all complaints from the backend. The response is expected to have an `_embedded` property that contains the list of complaints (each with HATEOAS links). This means my frontend does not need to know about specific endpoints for actions like deletion—the backend provides them.

3. Using Hypermedia Links for Deletion:

In the `confirmDelete()` method, instead of hardcoding the URL for deletion, the code locates the corresponding complaint and retrieves its deletion URL from the `_links` object (specifically, the "delete-complaint" link). This follows the HATEOAS principle by allowing the API to guide the client in what actions are available.

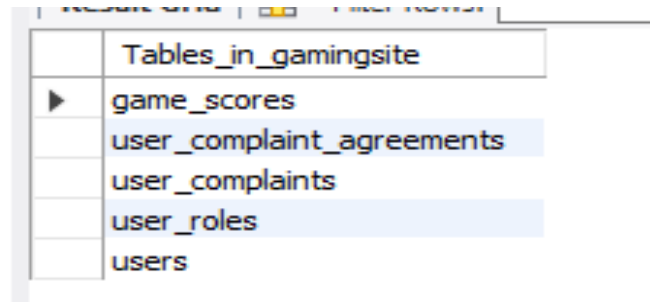
4. Benefits:

- **Discoverability:** The frontend can dynamically discover available actions based on the hypermedia links provided by the backend.
- **Decoupling:** Changes to endpoint URLs on the backend won't break the frontend as long as the hypermedia links remain updated.
- **Maintainability:** The client code is more resilient to backend changes, aligning with the design approach described in my report.

Error Handling & Validation

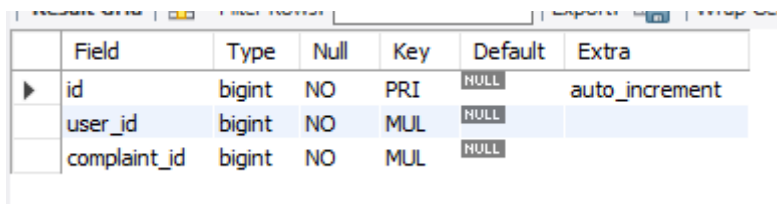
- Custom **Global Exception Handler** (`GlobalExceptionHandler.java`) to return standardized API errors.
 - Example error response:
-

4 Database Schema



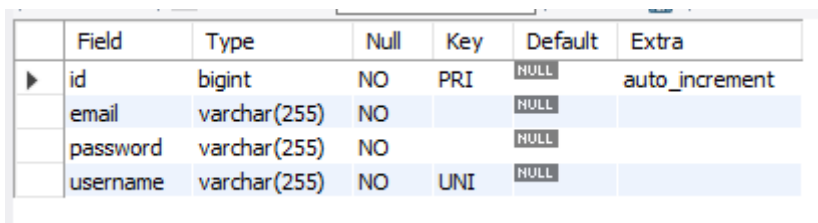
Tables_in_gamingsite	
▶	game_scores
	user_complaint_agreements
	user_complaints
	user_roles
	users

(Tables in Database)



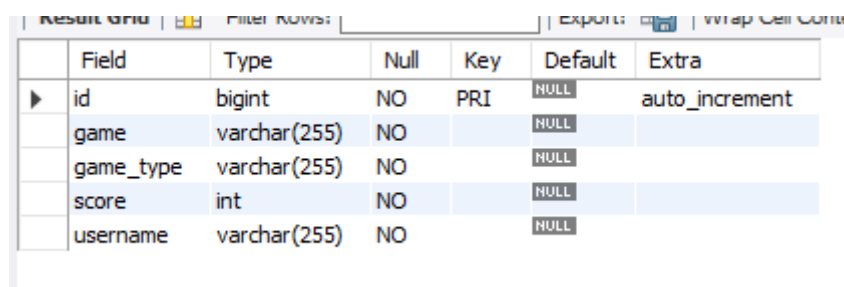
Field	Type	Null	Key	Default	Extra
▶ id	bigint	NO	PRI	NULL	auto_increment
user_id	bigint	NO	MUL	NULL	
complaint_id	bigint	NO	MUL	NULL	

(Description of user_complaint_agreements)



Field	Type	Null	Key	Default	Extra
▶ id	bigint	NO	PRI	NULL	auto_increment
email	varchar(255)	NO		NULL	
password	varchar(255)	NO		NULL	
username	varchar(255)	NO	UNI	NULL	

(Description of users)



Field	Type	Null	Key	Default	Extra
▶ id	bigint	NO	PRI	NULL	auto_increment
game	varchar(255)	NO		NULL	
game_type	varchar(255)	NO		NULL	
score	int	NO		NULL	
username	varchar(255)	NO		NULL	

(Description of GameScore)

One to many: User → Complaints, User → Scores:

Users serve as a central entity in the system. Each user can be associated with multiple other records (complaints, scores, roles).

UserComplaints represent issues submitted by users. The relationship is **one-to-many**, meaning a single user can submit many complaints, but each complaint belongs to exactly one user.

Many to Many: Users ↔ ComplaintAgreements:

UserComplaintAgreements capture which users agree with a given complaint. This creates a **many-to-many** relationship: many users can agree to the same complaint, and each user can agree to multiple complaints.

The rest follow as:

GameScores track each user's scores for different games. A user can have multiple scores, reflecting a **one-to-many** relationship between User and GameScores.

UserRoles) often implement a many-to-many relationship with Users—one user can hold multiple roles, and a single role can be assigned to many users, but they aren't an entity, they are enums, so I don't count them

5 Frontend Development

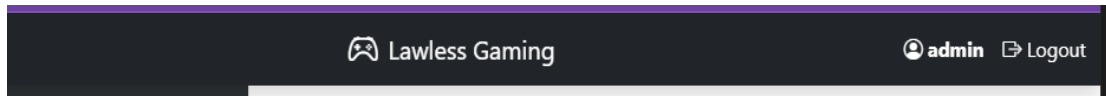
In my project, the frontend is built as a Single Page Application (SPA) that leverages several technologies to provide a smooth, responsive, and interactive user experience:

Technologies Used

- **Vue.js:**
I use Vue.js as the primary framework to build the SPA. It manages the dynamic component rendering and state management, allowing transitions between different views like login, registration, and game interfaces.

- **Bootstrap:**

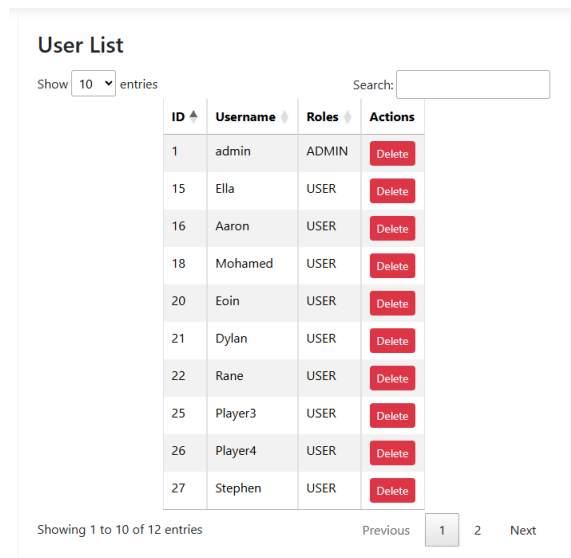
Bootstrap is used to create a responsive UI that adapts well across different devices and screen sizes.



(Example of Bootstrap Navbar)

- **DataTables:**

DataTables are integrated to display interactive tables for users, game scores, and complaints, making it easy for users to navigate and filter large data sets.

A screenshot of a 'User List' DataTable. It features a search bar at the top right and a 'Show 10 entries' dropdown at the top left. The table has four columns: ID, Username, Roles, and Actions. The Actions column contains red 'Delete' buttons for each row. The footer shows 'Showing 1 to 10 of 12 entries' and pagination controls for 'Previous', '1', '2', and 'Next'.

ID	Username	Roles	Actions
1	admin	ADMIN	Delete
15	Ella	USER	Delete
16	Aaron	USER	Delete
18	Mohamed	USER	Delete
20	Eoin	USER	Delete
21	Dylan	USER	Delete
22	Rane	USER	Delete
25	Player3	USER	Delete
26	Player4	USER	Delete
27	Stephen	USER	Delete

(jQuery DataTable)

- **jQuery AJAX:**

jQuery AJAX is used for API calls, ensuring that data is fetched and submitted without requiring full page reloads.

- **HTML Canvas:**

HTML Canvas is employed to render dynamic graphics and animations for the interactive games, providing a fast and good gaming experience.

Key Features

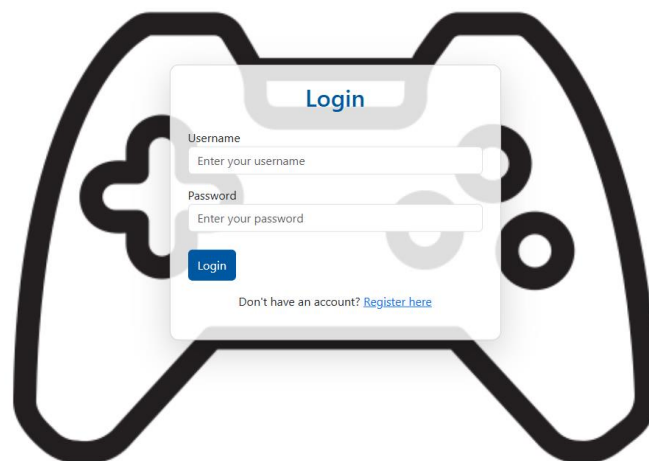
Dynamic Radar Charts:

In the stats module, two radar charts are displayed:

1. **Personal Performance Metrics** – Tracks user skills such as Precision, Puzzle Solving, and Luck.
2. **Global Performance Comparison** – Compares personal stats against global averages, providing insight into relative performance.

- **User Registration & Login with JWT Authentication:**

The login and registration processes are handled by Vue.js components (e.g., *login.js* and *register.js*). When users log in, their credentials are verified and a JWT is stored in local storage. This token is then used for all subsequent API calls, ensuring secure access



(Login with JWT Authentication)

- **Dynamic DataTables:**

I employ DataTables to display lists of users, game scores, and complaints. These tables update dynamically based on API responses via jQuery AJAX, providing real-time interactivity.

- **Real-Time Leaderboard Updates:**

The SPA architecture enables real-time updates of the leaderboard. As new game scores are submitted, the leaderboard refreshes instantly without reloading the entire page.




Leaderboard

Top Players

Most Active Players

Select a Game: Flappy Bird

Top Players for Flappy Bird

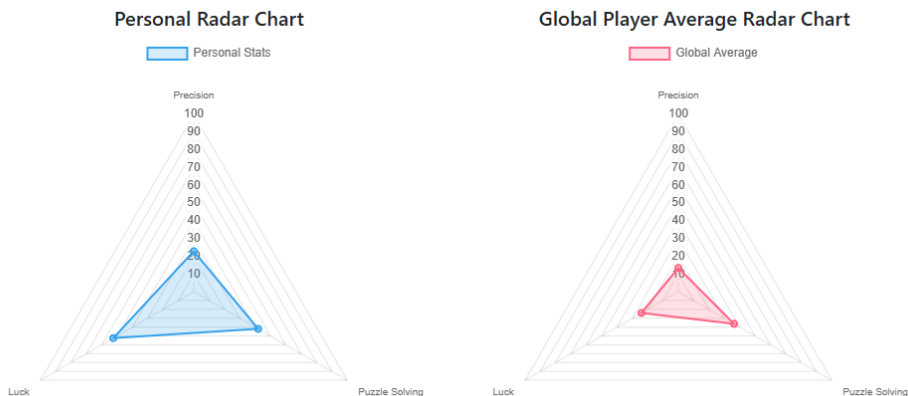
Rank	Username	Score
	player2	93
	player4	86
	player1	34
#4	player3	23
#5	Stephen	8
#6	Ella	7
#7	admin	6
#8	Rane	4
#9	Santa	2

(Example of the leaderboard)

Graphical Statistics and Analytics

To provide users with visual insights into their performance, I integrated interactive graphs for statistics: (Chart.js (2025), n.d.)

- Radar Charts:**
In the stats module, I use Chart.js to render radar charts. Two radar charts are displayed—one showing personal performance metrics (such as Precision, Puzzle Solving, and Luck) and the other comparing these metrics against global averages. This graphical representation helps users quickly understand their performance.



(Example of the two radar charts)

- **Dynamic Data Updates:**

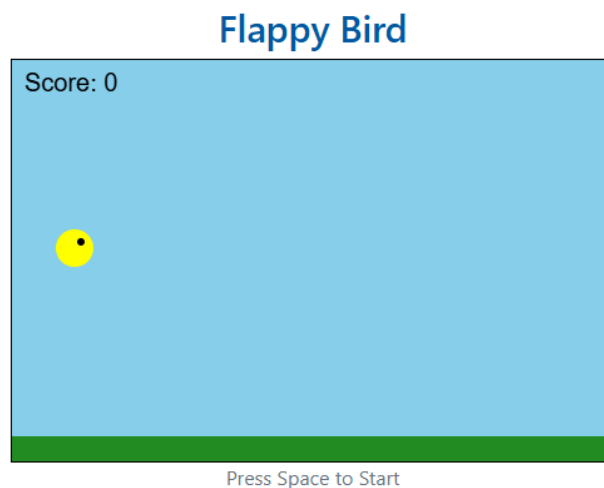
The charts update dynamically based on data fetched from the backend, ensuring that the statistics always reflect the latest performance metrics. This adds an extra layer of interactivity and provides users with actionable insights into their gameplay.

Game Development

For the interactive games, I built a few mini games (with help from ChatGPT and online versions (OpenAI, n.d.)) that combine gameplay with data management (GeeksforGeeks, n.d.) (GeeksforGeeks, n.d.) (Ferreira, n.d.):

- **Dynamic Graphics Rendering with HTML Canvas:**

The games use HTML Canvas to render dynamic visuals and animations, ensuring good quality, real-time gameplay.



(Example of HTML Canvas)

- **Asynchronous Score Saving:**

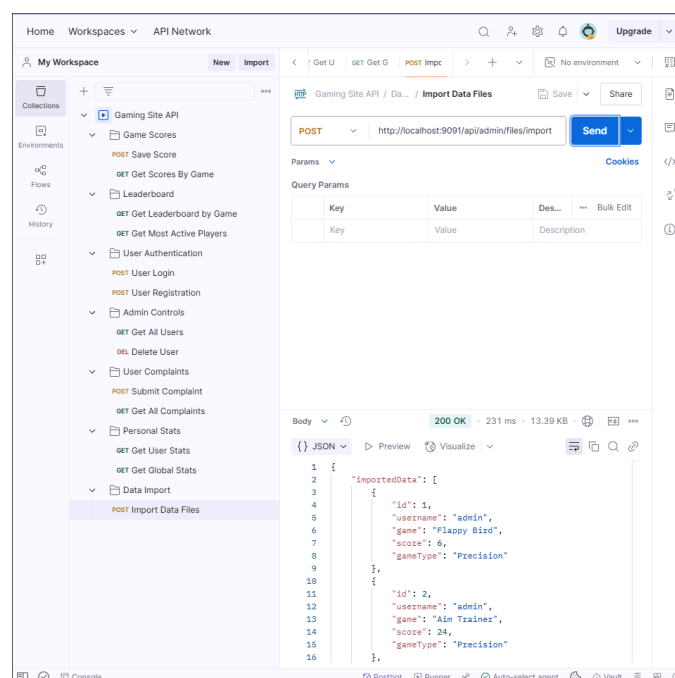
Each game includes asynchronous functions (using `async/await`) to save game scores. For example, in the Aim Trainer game, when the game ends, an `async` function posts the score to the backend without interrupting the gameplay. This non-blocking approach ensures that the user interface remains responsive even during score submission.

6 Testing & Error Handling

Testing Approach

The project employs a structured testing approach, focusing on backend validation, API security, and error handling.

- **Unit Tests:**
 - JUnit is used to test individual service methods and repository interactions.
 - MockMvc is utilized for controller testing without requiring a full application context.
- **Integration Tests:**
 - Postman API testing validates API responses against expected behaviors.
 - Endpoints requiring JWT authentication must include a valid token before accessing secured routes.
- **Postman API Testing:**
 - A Postman collection was created to test API functionality across user authentication, leaderboard management, and score tracking (Postman, n.d.).
 - Some endpoints require a valid JWT token, which can cause authentication failures if not included



(Example of postman Testing)

Validation & Error Messages

To ensure **data integrity** and **prevent security issues**, a **ValidationService** is implemented in the backend.

Key Validation Features:

1. User Existence Check

- Ensures a user with a given username does not already exist before allowing registration.
- Implemented in ValidationService.java:

```
public boolean userExists(String username) {  
    return userRepository.existsByUsername(username);  
}
```

2. Automatic Role Assignment

- If a user registers without specifying a role, they are automatically assigned the "USER" role.

```
public void ensureUserHasRole(User user) {  
    if (user.getRoles() == null || user.getRoles().isEmpty()) {  
        user.setRoles(Set.of(Role.USER)); // Default role is USER  
    }  
}
```

3. Common API Error Messages & Handling

- A global exception handler ensures proper error messages are returned for invalid operations.
- Example of user authentication error handling:

```
@PostMapping("/login")  
public ResponseEntity<?> createAuthenticationToken(@RequestBody AuthRequest  
    authRequest) {  
    try {
```

```
        authenticationManager.authenticate(  
            new UsernamePasswordAuthenticationToken(authRequest.username,  
            authRequest.password)  
        );  
    } catch (BadCredentialsException e) {  
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Incorrect username  
        or password");  
    }  
}
```

- **Example Error Messages:**

- "Invalid credentials" → When login details are incorrect.
- "User not found" → When trying to retrieve a non-existent user.
- "Unauthorized access" → When an API call is made without a valid JWT token.
- "Bad request" → When missing or invalid data is submitted in API requests.

📌 Conclusion & Future Work

Key Learnings

Developing this project provided several important insights:

- **Using Vue.js for a Single Page Application (SPA)** made the user experience smooth by allowing quick navigation without full-page reloads.
- **JWT Authentication** helped improve security by making sure only logged-in users could access certain features.
- **Spring Boot and MySQL** worked well together to create a reliable backend, making it easier to store and retrieve game data efficiently.
- **Chart.js for data visualization** made it easier for users to understand their performance with clear and interactive graphs.

Limitations

- **Testing Games is Difficult:**
 - Automated testing tools like Selenium don't work well for interactive games because they require real user input.
 - Testing had to be done manually, which takes more time and effort.
- **SPA Limitations:**
 - Since everything runs on the browser, the initial page load can be slow if a lot of data needs to be fetched.
 - Search Engine Optimization (SEO) is harder with SPAs because search engines struggle to index dynamic content.

Possible Improvements

- **More Detailed Stats:**
 - Adding deeper analysis, such as tracking accuracy over time or heatmaps for player actions, would give users better insights.
- **Multiplayer Mode:**
 - Introducing a two-player mode could make the games more engaging, but it would require real-time data synchronization and more complex backend logic.

Scalability Issues & Options

- **Database Scaling:**
 - Right now, MySQL is running locally, which limits how many users the system can handle. Moving to a cloud-based database like AWS RDS or Firebase would make it more scalable.
- **Backend Performance:**
 - Instead of constantly requesting updates from the server, using WebSockets could make real-time updates more efficient, especially for leaderboards.
- **Load Balancing:**
 - If the user base grows, running the backend on multiple servers with load balancing would help manage traffic and prevent slowdowns.

Bibliography

Auth0. (n.d.). Retrieved from JWT Authentication Guide: <https://auth0.com/learn/json-web-tokens>

Bootstrap (2025). (n.d.). Retrieved from Bootstrap Documentation: <https://getbootstrap.com>.

Chart.js (2025). (n.d.). Retrieved from Chart.js Documentation - Radar Graphs: <https://www.chartjs.org/docs/latest/charts/radar.html>.

DataTables (2025). (n.d.). Retrieved from DataTables - Interactive Tables: <https://datatables.net>.

Ferreira, M. (n.d.). *JavaScript Memory Game*. Retrieved from github: <https://marina-ferreira.github.io/tutorials/js/memory-game>

GeeksforGeeks. (n.d.). Retrieved from Flappy Bird Game in JavaScript: <https://www.geeksforgeeks.org/flappy-bird-game-in-javascript>

GeeksforGeeks. (n.d.). Retrieved from Typing Speed Test Game in JavaScript: <https://www.geeksforgeeks.org/design-a-typing-speed-test-game-using-javascript>

GitHub Repository. (n.d.). Retrieved from Lawless Gaming: WebTech Project: <https://github.com/EoinieLawless/WebTechProject>

jQuery (2025). (n.d.). Retrieved from jQuery API Documentation: <https://api.jquery.com>

MySQL. (n.d.). Retrieved from MySQL Workbench Documentation: <https://dev.mysql.com/doc/workbench/en>

ONAP. (n.d.). Retrieved from RESTful API Design Specification: <https://wiki.onap.org/display/DW/RESTful+API+Design+Specification>

OpenAI. (n.d.). Retrieved from ChatGPT.

Postman. (n.d.). Retrieved from Postman API Testing Tool: <https://www.postman.com>

Spring Boot. (n.d.). Retrieved from Spring Boot Framework Documentation: <https://spring.io/projects/spring-boot>

Vue.js. (n.d.). Retrieved from Vue.js Official Documentation: <https://vuejs.org>

Link to Github:

<https://github.com/EoinieLawless/WebTechProject>