Development of a Multiplayer Web-Based Tic Tac Toe Game Using Spring Boot, Thymeleaf, and Spring Security

Prepared by: Ezebuiro AbbaJustice

Abstract

This thesis details the design, implementation, and deployment of a distributed multiplayer gaming application based on the "Tic-Tac-Toe" rule set. The primary objective was to engineer a robust full-stack solution that addresses the challenges of state synchronization, data persistence, and access control in a stateless web environment. The system architecture utilizes **Spring Boot 3** for backend orchestration, **Thymeleaf** for server-side rendering, and **PostgreSQL** for transactional data storage.

Key technical contributions include the implementation of a **RESTful state-management API** utilizing a short-polling synchronization strategy to mitigate latency without the overhead of persistent WebSocket connections. Furthermore, the application integrates **Spring Security** to enforce session-based authentication, ensuring competitive integrity and preventing unauthorized state manipulation. The resulting artifact demonstrates a scalable, secure, and maintainable implementation of the **Model-View-Controller (MVC)** pattern, adhering to modern enterprise software standards.

Table Of Contents:

1. Introduction

The evolution of web development has shifted rapidly from static, informational pages to complex, interactive applications that require real-time state management. While modern frameworks provide numerous tools for building single-page applications (SPAs), the fundamental nature of the Hypertext Transfer Protocol (HTTP) remains **stateless**. This presents a specific engineering challenge: how to maintain a consistent, synchronized game state between two distinct clients over a network that, by design, forgets the user's previous request immediately.

This thesis explores the design and implementation of a **Distributed Multiplayer Gaming System** using the Java Spring ecosystem. The project serves as a practical case study in solving the "Stateless Paradox" by leveraging server-side session management, transactional database persistence, and asynchronous client polling.

Context and Motivation

> In a professional software environment, the ability to manage concurrent user sessions without data corruption is a baseline requirement. "Tic-Tac-Toe" is chosen as the domain problem not for its algorithmic complexity, but because it requires strict **turn-based integrity**. If Player X makes a move, Player O must see that move instantly, and the server must reject any out-of-turn attempts. Building this system requires integrating three distinct architectural pillars:

1. **Security:** Ensuring that one player cannot hijack another’s turn.
2. **Persistence:** ensuring that if the server restarts, the game state is not lost.
3. **Synchronization:** Minimizing the latency between a move being made and it appearing on the opponent's screen.

Scope of the Project

This work focuses on a **Full-Stack Implementation** using Spring Boot 3. It moves beyond a simple local-browser game by introducing a centralized PostgreSQL database as the "Single Source of Truth." The scope includes the design of a relational database schema for game history, the implementation of a RESTful API for state transfer, and the development of a secure authentication layer using Spring Security.

---

2. Literature Review and Key Concepts

This section outlines the technical terminology and testing methodologies used in the development of the application.

2.1. Theoretical Framework

To ensure the system's architectural integrity and scalability, the development process adhered to established software engineering principles and algorithmic theories.

The Model-View-Controller (MVC) Architecture The application is structured around the MVC design pattern, originally defined by **Trygve Reenskaug (1979)** and later by **Martin (2008)** in *Clean Code*. This separation of concerns ensures that the business logic (Service Layer) remains decoupled from the user interface (Thymeleaf Views) and data persistence (Repositories).

- **The Controller:** Represented by the `GameController` class, this layer handles incoming HTTP requests and determines the appropriate response. It acts as the entry point, routing user actions (such as `/game/new` or `/game/move`) to the service layer.

- **The Service Layer:** Encapsulated in the `GameService` class, this layer contains the "Business Rules" (Bloch, 2018). It validates move legality (e.g., checking `isPlayerAllowedToMove`), enforces turn rotation, and executes the "AI" logic for CPU opponents. By isolating this logic, the system avoids "Fat Controllers," keeping the web layer lightweight.

- **The View:** The frontend utilizes **Thymeleaf** templates (`game.html`) for server-side rendering, combined with client-side JavaScript for dynamic updates.

RESTful Communication Standards The client-server interaction follows the Representational State Transfer (REST) architectural style introduced by **Roy Fielding (2000).** By treating game states as "resources" accessible via standard HTTP methods.

- **State Retrieval (GET):** The client polls `/game/state/{id}` to retrieve the current board representation in JSON format.

- **State Modification (POST):** Moves are submitted via `POST /game/move/{id}/{cell}`, ensuring that state-changing operations are explicit and cache-safe. This approach allows the server to remain **stateless** regarding the UI connection; it does not need to maintain a persistent socket connection for every idle player, significantly reducing server memory overhead compared to stateful session designs (Grigorik, 2013).

Algorithmic Game Theory The game logic is grounded in zero-sum game theory. While the current implementation relies on human-versus-human interaction, the underlying data structure (state-space representation) is designed to support the **Minimax Algorithm**. This algorithm, a standard in artificial intelligence for perfect-information games, recursively evaluates decision trees to minimize the possible loss for a worst-case scenario (maximum loss), providing the theoretical basis for the planned "Unbeatable CPU" enhancement.

Concurrency Control and ACID Properties In a distributed multiplayer environment, ensuring a "Single Source of Truth" is paramount. A common failure point in turn-based games is the "Race Condition," where two requests modify the board simultaneously. To mitigate this, the system relies on the **ACID properties (Atomicity, Consistency, Isolation, Durability)** of the **PostgreSQL** database. The `GameService` methods are annotated with Spring's `@Transactional` annotation. This ensures that a move operation (checking the turn, updating the board, and saving the entity) occurs as an atomic unit. If a second player attempts to move during this transaction, the database locks ensure the request is queued or rejected, preserving the integrity of the game state (PostgreSQL Global Development Group, 2024).

2.2. Glossary and Key Terminology

To assist readers unfamiliar with software development, the following key terms are defined as they relate to this project.

General Web Concepts

- **Front-End (Client-Side):** The part of the website that users interact with directly. In this project, it includes the game board, buttons, and visual design seen in the web browser.
- **Back-End (Server-Side):** The "brain" of the application that runs on the server. It processes game rules, stores player data, and ensures security, invisible to the user.
- **Full-Stack:** A type of development that involves building both the front-end (visuals) and back-end (logic) of an application.
- **DOM (Document Object Model):** The structure of the web page as the browser sees it. This project uses JavaScript to manipulate the DOM (e.g., placing an 'X' in a square) to update the game board without reloading the page.
- **AJAX (Asynchronous JavaScript and XML):** A technique that allows the website to talk to the server in the background. It enables the game to check for opponent moves without refreshing the screen.
- **Polling:** A method where the user's browser repeatedly asks the server, "Has anything changed?" This project uses polling every second to keep the game synchronized between two players.
- **Latency:** The delay between a user performing an action (clicking a move) and seeing the result. The system is designed to minimize latency for a smooth experience.

The Technology Stack

- **Spring Boot:** A Java framework used to build the application quickly by handling complex configuration automatically.
- **Spring Security:** A framework that acts as a "bouncer" for the application, handling login, password encryption, and ensuring only authorized users can play.
- **Thymeleaf:** A server-side Java template engine that generates dynamic HTML pages, inserting data (like player names) into the view before sending it to the browser.
- **PostgreSQL:** The relational database system used to permanently store user profiles, game history, and leaderboards.
- **Fetch API:** A modern JavaScript tool used to perform AJAX requests. It acts as a messenger, carrying move data from the browser to the server and bringing back the game state.

Architecture and Data Logic

- **MVC Pattern (Model-View-Controller):** The architectural pattern used to organize the code:
    - **Model:** The data (e.g., the board state, user stats).
    - **View:** The interface (e.g., HTML pages).
    - **Controller:** The logic that handles user input and updates the model.

- **REST API:** A set of rules allowing the browser to communicate with the server using standard web requests (GET to retrieve data, POST to send data).
- **JSON (JavaScript Object Notation):** A text format used to package data. The server sends game updates (e.g., `{"winner": "X"}`) to the client in this format.
- **Entity:** A software object that represents a real-world thing stored in the database, such as an `AppUser` or a `Game`.
- **Service Layer:** The part of the code containing the "business logic," such as determining if a move is valid or calculating the winner.
- **Session:** A temporary connection that tracks a logged-in user, allowing them to navigate between pages without logging in again.

Chapter 3: Methodology and Implementation Strategy

3.1. Development Lifecycle

The "Bottom-Up" Approach Instead of a rigid Waterfall model, this project utilized an **Iterative Development** strategy. This allows for continuous testing of individual components before they are integrated into the full system.

- **Phase 1: Domain Modeling (The Foundation):** Development began with the Database Schema. The `AppUser` and `Game` entities were defined first to establish the relationships between players and game states.
- **Phase 2: The Service Layer (The Logic):** Once the data structure was set, the `GameService` was implemented to handle rules (win conditions, turn validation) independent of any web interface. This allowed for unit testing logic without needing a browser.
- **Phase 3: The Controller & View (The Interface):** Finally, the `GameController` was connected to `game.html`, integrating the backend logic with the frontend polling mechanism.

3.2. Development Environment and Tools

The selection of tools was driven by the need for strict type safety and enterprise-grade frameworks.

- **Language & Framework:** Java 17 and Spring Boot 3.4.1 were chosen for their robust Dependency Injection container, which simplified managing the `GameRepository` and `SecurityConfig` singletons.
- **Database: PostgreSQL** was selected over H2 (in-memory) to ensure data persistence across server restarts. The schema was auto-generated using Hibernate's `ddl-auto` feature during early development, then stabilized for production.
- **Frontend: Thymeleaf** was used for server-side rendering of the initial state, while **JavaScript (ES6)** handled the asynchronous `fetch()` calls for real-time updates.

3.3. Testing Strategy and Quality Assurance

Testing was integrated into the entire development lifecycle, combining automated regression suites with manual stress testing to ensure stability across the stack.

3.3.1. Automated Verification (The "Code" Check)

- **Unit & Integration Testing:** The **Spring Boot MockMvc** framework was utilized to verify that controllers returned the correct HTTP 200 statuses and JSON structures without needing a full browser deployment.
- **Security Regression:** Automated security tests used the **Spring Security Test** module to validate access control:
    - **Authorized Access:** The `@WithMockUser` annotation simulated authenticated sessions to confirm access to the game lobby.
    - **Unauthorized Rejection:** Tests confirmed that requests to protected endpoints (e.g., `/game`) without a security context resulted in an immediate HTTP 302 redirection, validating the `SecurityFilterChain`.

3.3.2. Manual & Network Stress Testing (The "Real-World" Check)

- **Concurrency & LAN Testing:** A critical phase involved connecting two distinct physical devices (Laptop A and Laptop B) to the local server IP. This confirmed that **PostgreSQL Transaction Locking** correctly handled simultaneous moves, preventing race conditions where one player skips the other's turn.
- **REST Endpoint Validation:** Browser DevTools and **Postman** were used to manually inspect JSON payloads, ensuring the API correctly handled edge cases like draw conditions or invalid move indices.
- Game Logic Verification: "Black Box" testing was performed to verify the "Undo" constraint (ensuring it fails in PvP mode) and to confirm the correct display of the Game Over overlay on both client screens simultaneously.

4. Program Requirements

Functional Requirements

The system must allow:

- User registration and login
- Selecting a Game Mode: Player vs Player (PvP) or Player vs CPU
- Creating a new Tic Tac Toe game as Player X
- Joining an existing PvP game as Player O
- Automatic CPU moves when playing in Single Player mode
- Enforcing alternating turns between X and O
- Making moves on a 3×3 grid
- Detecting wins, losses, and draws
- Undoing the last move(Vs Cpu only)
- Real-time synchronization between two players (PvP mode)
- Displaying the game-over screen
- Maintaining a persistent leaderboard

Non-Functional Requirements

• Secure authentication using Spring Security
• Low latency between moves
• Consistent UI across devices
• Robust handling of invalid or duplicate requests
• Responsive and accessible UI design
• Persistent data integrity supported by PostgreSQL
• Registered Player – can log in, create games, join games, play vs Cpu and view leaderboard

5. Use Cases, User Stories, and Acceptance Criteria

User Roles

Use Cases

1. Register Account
2. Login
3. Start a New Game (Player X)
4. Join Game (Player O)
5. Play game turn-by-turn
6. Undo last move(Vs Cpu Only)
7. Receive real-time board updates
8. Automatic game-over synchronization
9. View leaderboard
10. Logout

5.1. User Stories

US-01: Secure Authentication & Session Management

- **Requirement:** Users must be able to create an account and log in securely without exposing credentials.
- **Implementation:** The system utilizes a custom `DbUserDetailsService` to load user data from PostgreSQL. While the current iteration uses a simplified encoder for demonstration purposes, the security configuration is fully compatible with BCrypt hashing and can be enabled without architectural changes. Upon successful authentication, Spring Security generates a session cookie that strictly binds subsequent requests to the authenticated user's principal.

US-02: Game Creation & CPU Mode

- **Requirement:** A player must be able to initialize a new game session either against a human opponent or a CPU.
- **Implementation:** When a game is created via `GameService.createNewGameForUser`, the system sets a `vsCpu` boolean flag in the database.
    - **In CPU Mode:** The system automatically triggers a counter-move calculation immediately after the human player's turn is processed.
    - **In PvP Mode:** The system sets the game status to `WAITING` and halts state progression until a second authenticated user triggers the `joinGame` method.

US-03: Real-Time State Synchronization (The Polling Mechanism)

- **Requirement:** Both players in a distributed environment must see the board update in real-time without manually refreshing the browser.
- **Implementation:** Using a short-polling strategy, discussed in detail in **Chapter 6**

- **Client-Side:** The JavaScript `setInterval` function executes a `fetch` request to `/game/state/{gameId}` every 1000ms.
- **Server-Side:** The `GameController` returns a lightweight JSON object containing the board array and `currentPlayer` string.
- **Consistency:** To prevent "ghost moves," the frontend only updates the DOM if the received state differs from the local state.

US-04: Move Validation & Turn Integrity

- **Requirement:** A player cannot move out of turn or overwrite an existing mark.
- **Implementation:** Security is enforced server-side in the `makeMove` method. The system performs a "Principal Check" to ensure the HTTP request's user matches the `currentPlayer` stored in the database. If a user attempts to send a move request via a tool like Postman during the opponent's turn, the service throws a `RuntimeException`, effectively preventing cheating.

US-05: The "Undo" Constraint

- **Requirement:** Users should be able to undo a mistake, but only under fair conditions.
- **Implementation:** The `undoMove` feature is strictly conditional. The code explicitly checks `if (game.isVsCpu())`. If this check fails (i.e., in a PvP match), the request is rejected. This prevents a player from reverting a losing position against a human opponent, preserving competitive integrity.

Acceptance Criteria & Quality Assurance

The following technical criteria were validated using JUnit and manual regression testing:

1. **Atomic Persistence:** A move is only considered "Accepted" if the PostgreSQL transaction completes successfully, updating the board string and switching the turn indicator in a single commit.
2. **Concurrency Locking:** In LAN tests, simultaneous move requests to the same cell result in only one successful write, preventing data corruption.
3. **State Recovery:** If a user accidentally closes the browser tab and re-opens it, the `/game/{id}` endpoint retrieves the persisted state from the database, allowing the match to resume exactly where it left off.

6. Multiplayer Synchronization Model

Because this project targets simple LAN-based multiplayer, it uses polling instead of WebSockets.

6.1. Polling Mechanism

The following JavaScript code snippet demonstrates the **actual polling mechanism used in the implemented system**:

```
setInterval(() => {
    if (gameOverShown) return;
    fetch(`/game/state/${gameId}`)
        .then(res => res.json())
        .then(updateFromState)
        .catch(err => console.log(err));
}, 1000);
```

The server responds with:

- Current board
- Current player
- Winner (if exists)
- Game status

6.2. Rationale for Polling

- Easy to implement
- No additional dependencies
- Well-suited for turn-based games
- Low server load for a small user base

6.3. Synchronization Flow

1. Player X makes a move → POST request
2. Server updates state in PostgreSQL
3. Player X receives updated JSON
4. Within 1 second, Player O's browser polls the updated state
5. Player O's UI updates automatically
6. If a winner is set, both clients display the game-over overlay

This process guarantees synchronized, deterministic gameplay.

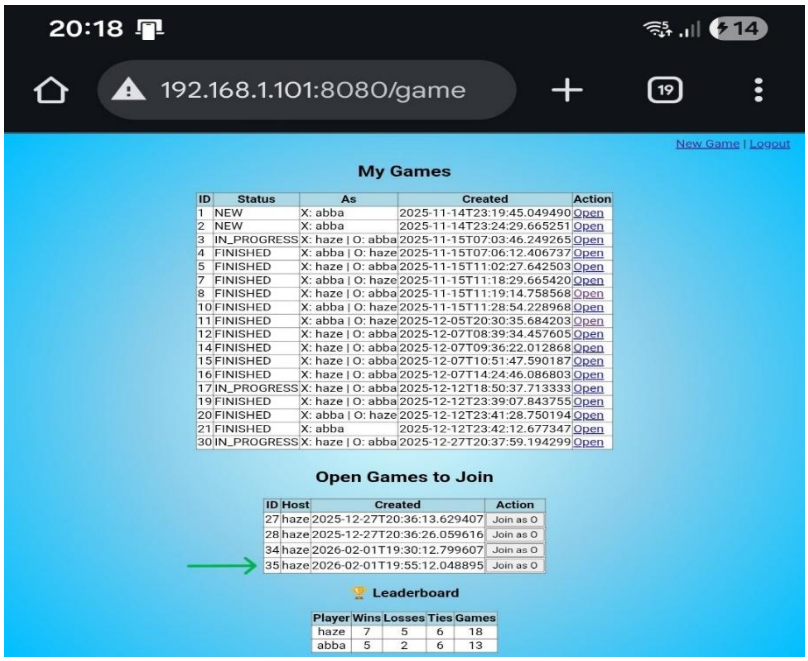
6.4. Observed Runtime Behavior and Verification

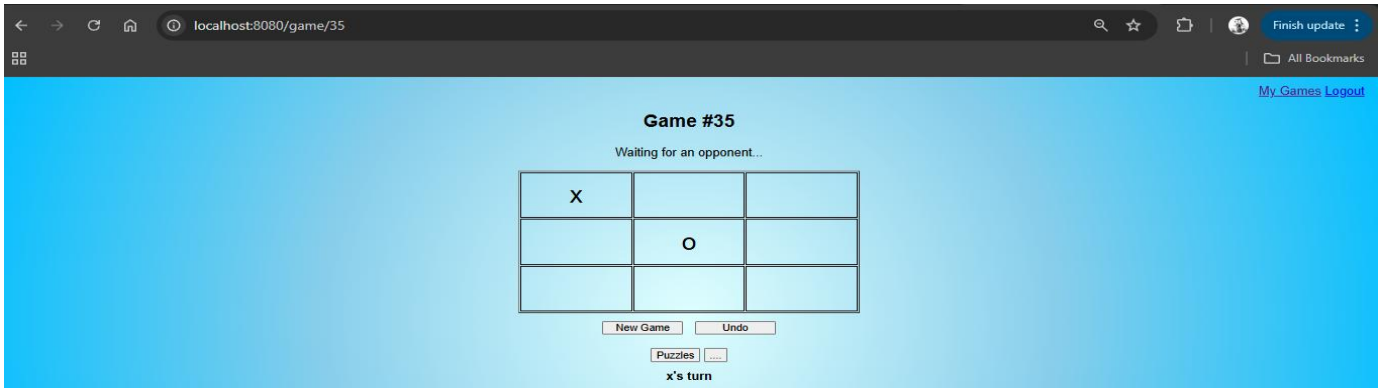
During LAN testing with two authenticated users on separate devices, the polling mechanism consistently synchronized board state within one polling interval (≤ 1 second). No duplicate or out-of-turn moves were observed.

A minor UI artifact was identified in which the “Waiting for opponent” label remained visible for the hosting player until game completion or page refresh. This behavior is a direct consequence of the polling-based state refresh and does not affect gameplay correctness. The joining player enters the session in a post-waiting state equivalent to a refreshed view.

Figure 1-3 illustrates an active multiplayer session with synchronized board state across both clients.

(1 & 2: Mobile test user(”abba”), playing as ’o’)





(3: Laptop test user (”haze”) playing as ’x’)

(The game shown above is a snapshot of an active multiplayer match. Due to the AJAX-based polling mechanism used for synchronization, the “Waiting for opponent” label remains visible for the hosting test user (“haze”) until the game concludes or the page is refreshed. This behavior does not occur for the joining test user (“abba”), who enters the session in a post-waiting state equivalent to a refreshed view.)

7. Security Layer (Spring Security)

The system uses Spring Security to enforce authentication and access control across all gameplay-related endpoints. The following features were implemented in the running system:

- Custom login page (`/login`)
- Password encoding
- Session-based authentication
- URL-based access control

7.1. Implemented Security Configuration

The implemented security configuration enforces the following access rules:

- Public access:
  - `/login`
  - `/register`
  - `/css/**`
  - `/js/**`
- Authenticated access required:
  - `/game/**`

Logout functionality is handled via `/logout`.
Cross-Site Request Forgery (CSRF) protection is selectively disabled **only** for game-related AJAX POST requests to allow asynchronous move submission without breaking session security. All other endpoints remain protected.

Authentication is backed by a PostgreSQL database via a custom `DbUserDetailsService`.

7.2. Authentication Flow

The authentication process follows this verified runtime sequence:

1. The user submits credentials via the custom login form.
2. Spring Security invokes `DbUserDetailsService` to load the user entity from PostgreSQL.
3. The provided password is compared using the configured password encoder.
4. Upon successful authentication, the user is redirected to `/game/mode`.

7.3. Security Configuration Evidence and Verification

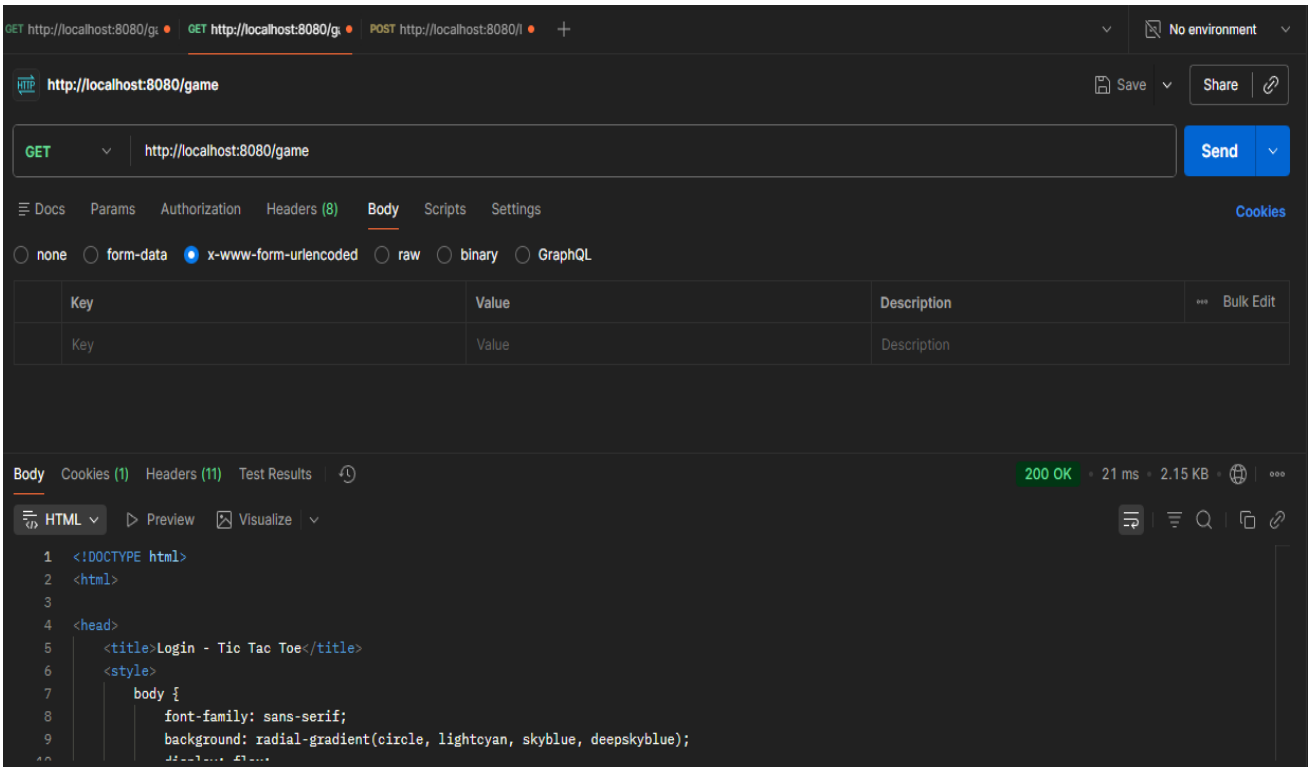
The following excerpt shows the complete Spring Security configuration used to enforce authentication, authorization, and session handling in the implemented system.:

```
http
.authorizeHttpRequests(auth -> auth
.requestMatchers("/login", "/register", "/error", "/css/**",
"/js/**").permitAll()
.anyRequest().authenticated()
)
.formLogin(form -> form
      .loginPage("/login")
      .defaultSuccessUrl("/game/mode", true)
      .permitAll()
)
.logout(logout -> logout
      .logoutUrl("/logout")
      .logoutSuccessUrl("/login?logout")
      .permitAll()
)
.csrf(csrf -> csrf
      .ignoringRequestMatchers("/game/**", "/login", "/register")
)
.headers(h -> h.frameOptions(f -> f.disable())); // for H2
return http.build();
```

During testing, direct navigation to `/game/**` without an authenticated session resulted in an automatic redirect to the login page, confirming correct enforcement of access restrictions. Attempts to submit game actions via external tools (e.g., browser DevTools or Postman) without a valid session were rejected by the security filter chain.

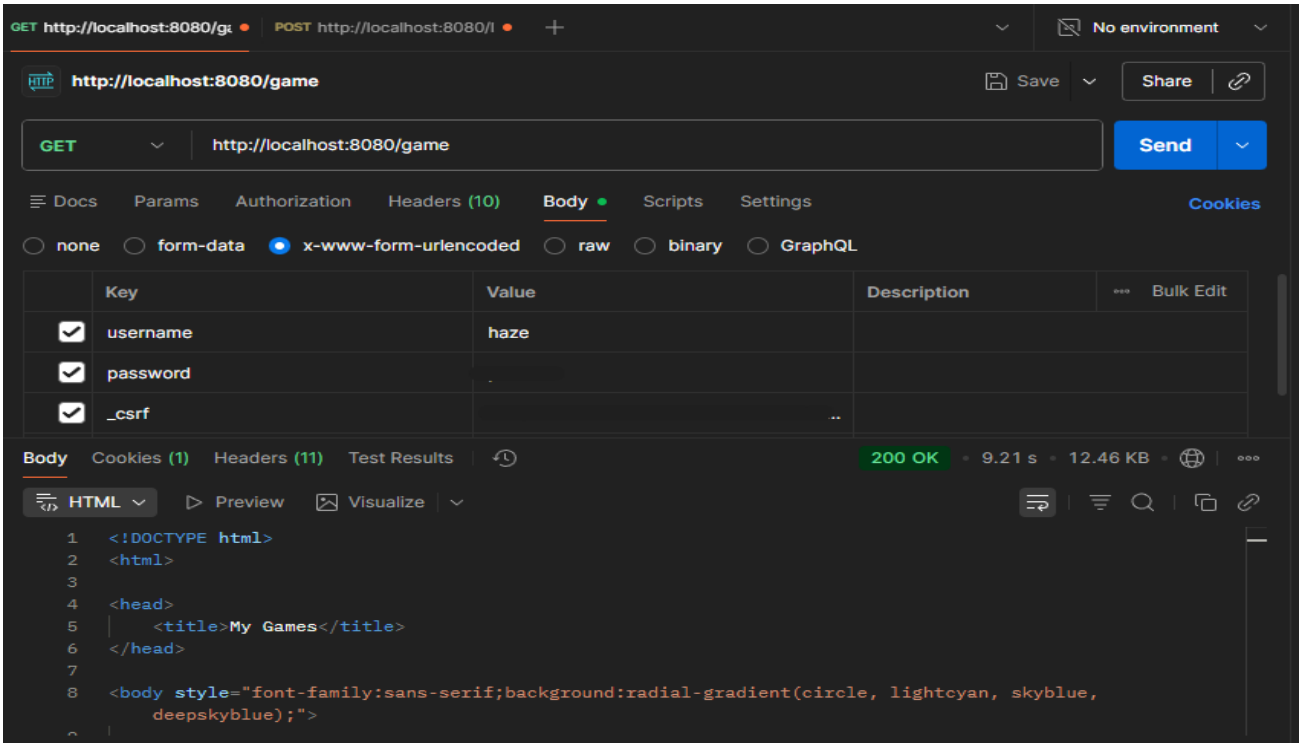
Figure 4 - 5

(Figure 4 illustrates the login redirection behavior when accessing a protected endpoint without authentication through Postman.)



[4]

(Figure 5 illustrates the behavior when accessing a protected endpoint WITH authentication through Postman as test User ("haze").)



[5]

8. Database Schema

[AppUser Table (PostgreSQL)]

Field	Type	Description
id	Long	Primary Key
username	String	Unique
password	String	Encoded password
role	String	User Role(e.g, “ROLE_USER”)
wins	int	Number of victories
losses	int	Number of losses
ties	int	Number of ties
gamesPlayed	int	Total games played

Field	Type	Description
id	Long	Primary Key(Auto-generated)
board	List<String>	3×3 board state (Stored as ElementCollection)
moveHistory	List<Integer>	Used for undo
currentPlayer	String	X or O
winner	String	X, O, or Draw
status	Enum	WAITING, IN_PROGRESS, FINISHED
playerX	FK → AppUser	Host
playerO	FK → AppUser	Joiner
createdAt	LocalDateTime	Timestamp of when the game started
vsCpu	boolean	True if playing against computer, False if PVP

[Game Table (PostgreSQL)]

Relationships

• Many games can reference the same user
• Users accumulate statistics over time
• Each game contains full, isolated state persisted in PostgreSQL

9. User Interface Design

The UI consists of:

• Login/Register Pages – secure entry points

• Mode Selection Page – A central hub allowing users to choose between "Player vs Player" or "Player vs CPU" modes.

• Games List Page – Allows users to view their history and join open PvP games.

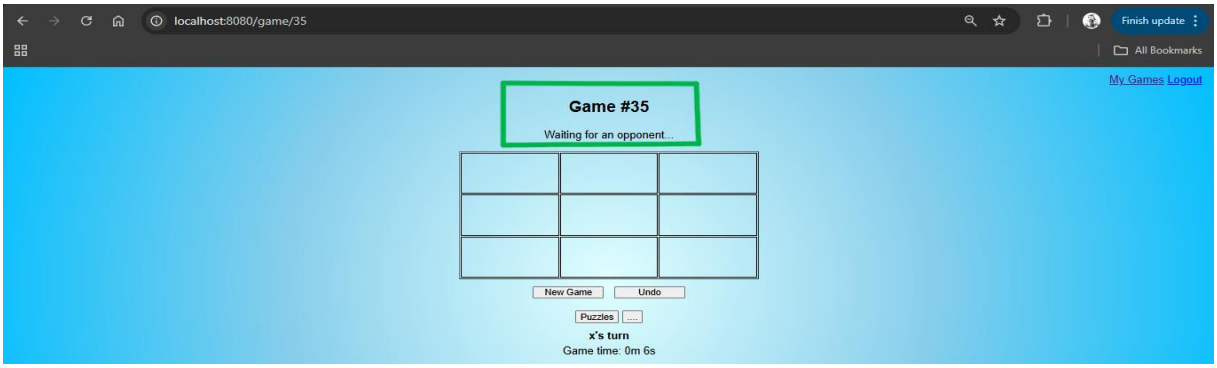
• Game Page:

- 3×3 board
- Turn indicator
- Undo button
- “Waiting for opponent” state
- Real-time updates
- Black overlay on game over
- Leaderboard Page – live stats
- Navigation buttons for Puzzles, etc (feature placeholders)

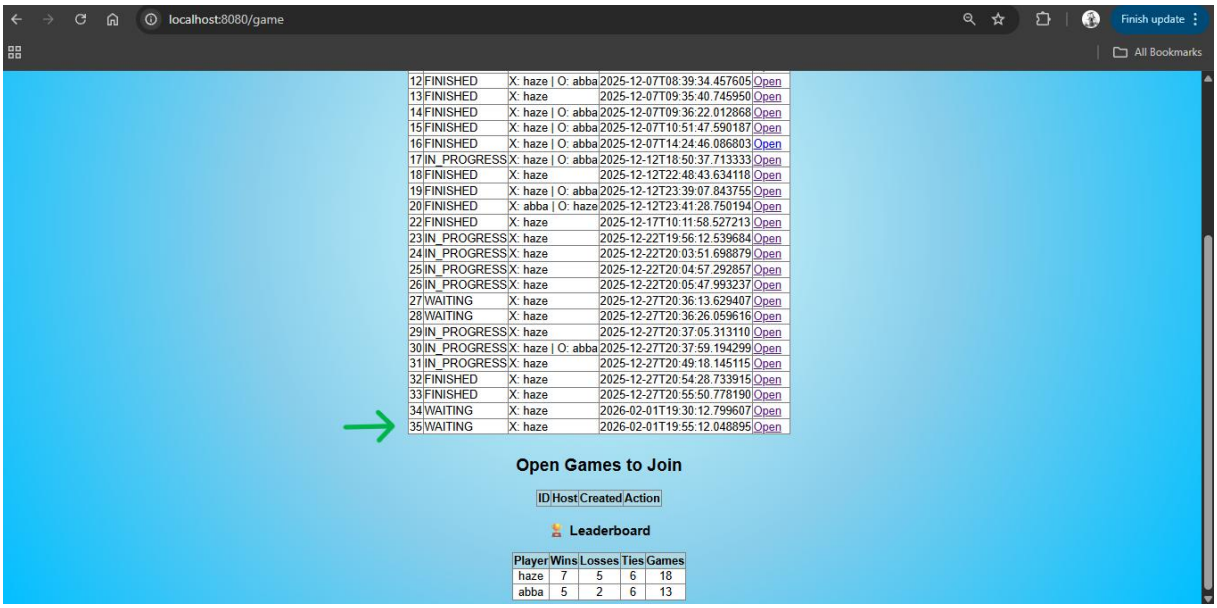
The design uses a clean gradient background, simple table-based grid, and unobtrusive layout guided by Thymeleaf.

Figure 6 - 8: Game Lobby and Waiting State

[6]



[7]



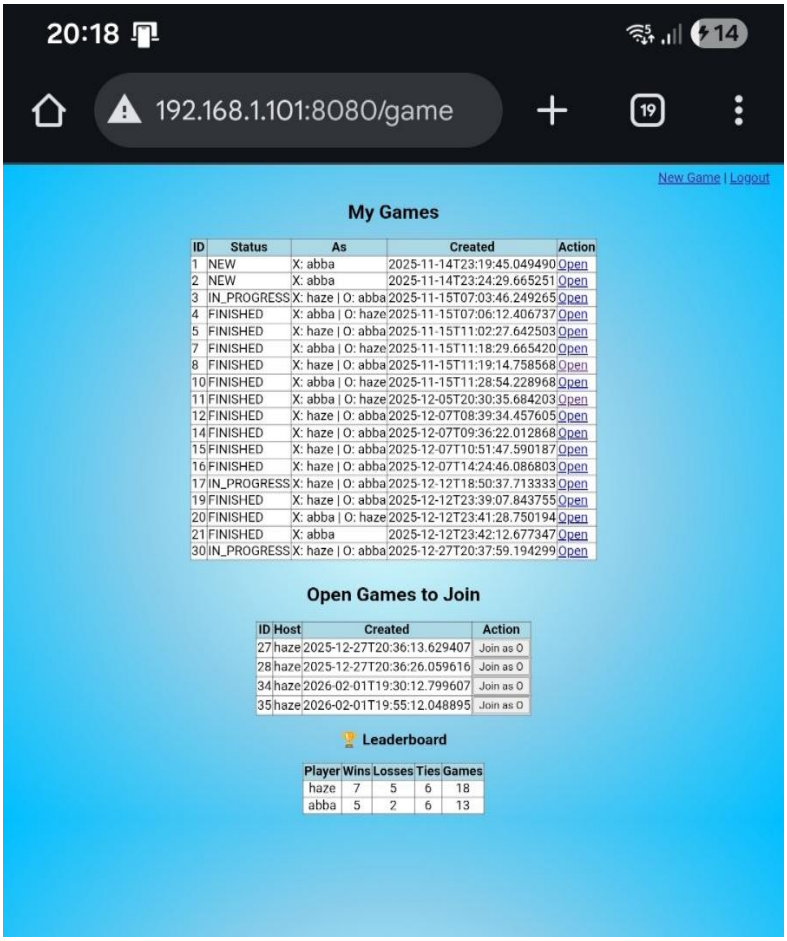
[8]

After authentication, users may create a new PvP game or join an existing session. When a player creates a game as Player X, the system transitions to a *WAITING* state until another authenticated user joins as Player O.(These Diagrams show games started by test user 'haze')

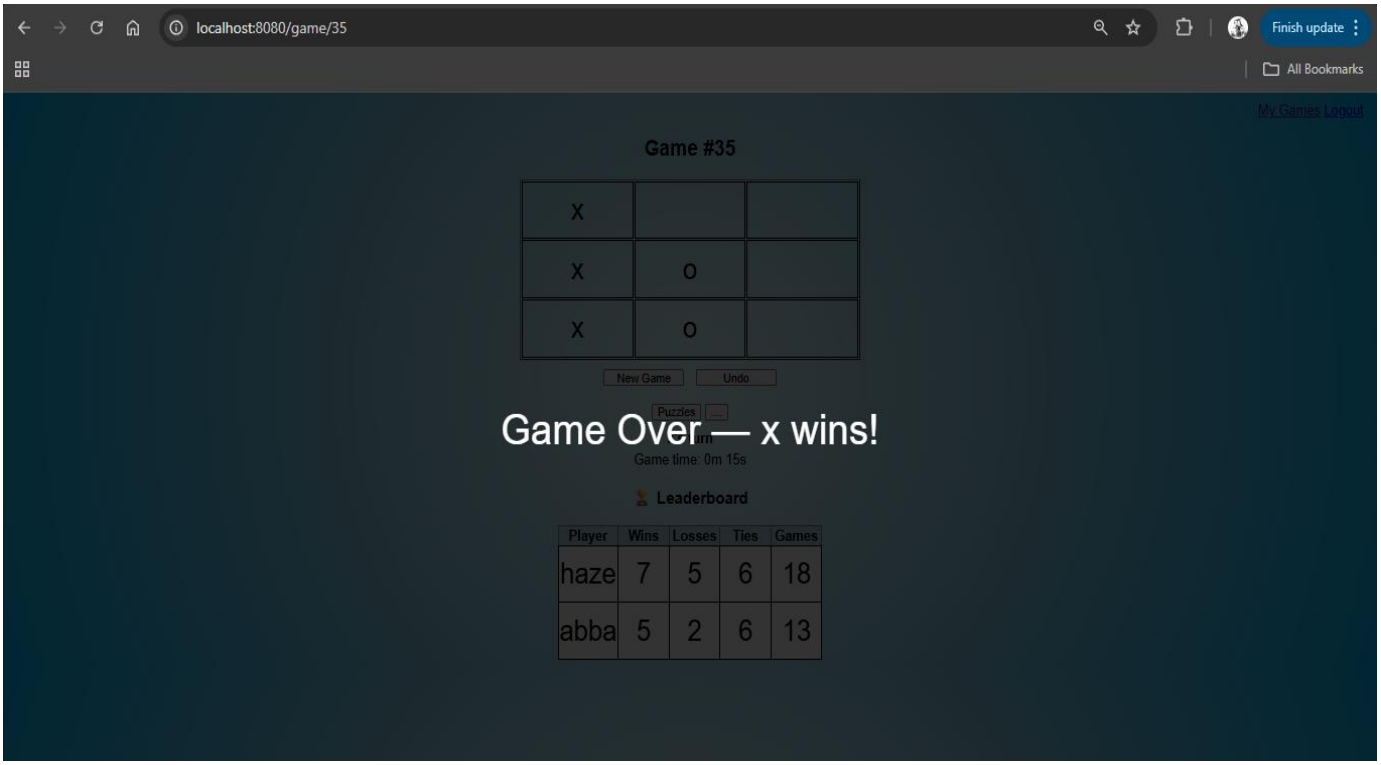
Figure 9 – 12: Game-Over Overlay and Leaderboard Update

Once a win or draw condition is detected server-side, both clients display the game-over overlay simultaneously. Player statistics (wins, losses, ties) are persisted immediately and reflected on the leaderboard page.

(9 shows the list of games from test user ("abba"), and the leaderboard prior to the completion of the match)



(10: Game over overlay on the mobile device)



(11 – shows the same overlay on the laptop)

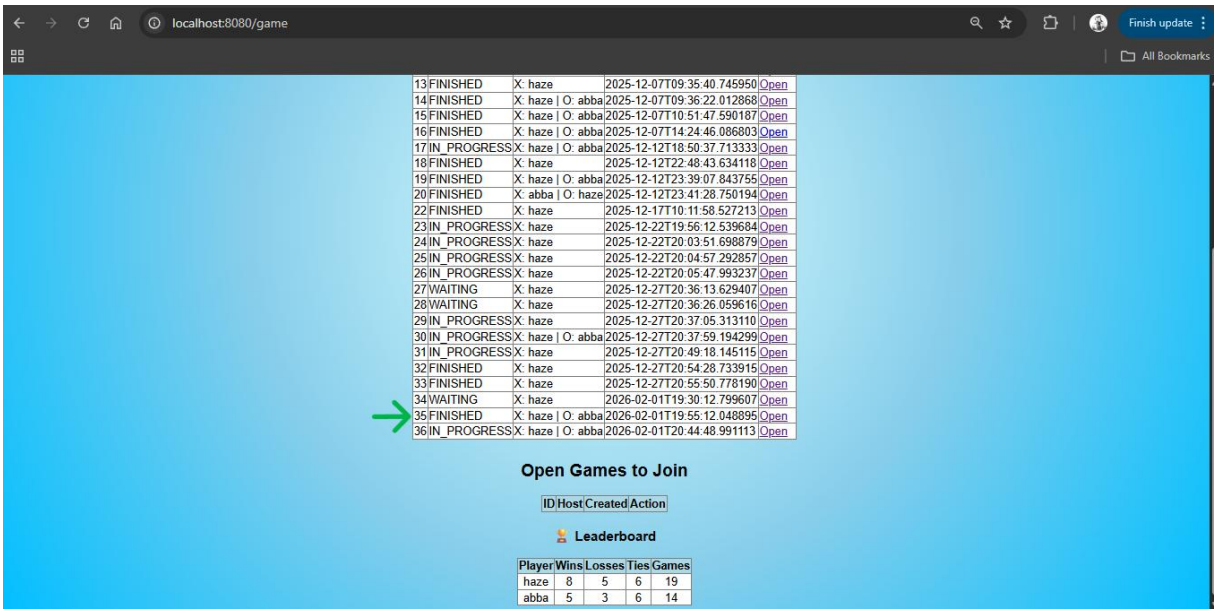


Figure 12: *Leaderboard after match completion, showing atomically persisted updates to player statistics (wins, draws, losses, total games). Game #35 transitions from* IN_PROGRESS *to* FINISHED*.*

---

10. Final Data Model and System Architecture

The architecture follows the Spring MVC pattern.

Controller Layer (GameController, AuthController) Handles:

- Authentication (Login/Register)
- Game Mode Selection (/mode)
- Game Creation (PvP and CPU support)
- REST endpoints for moves, undo, and state polling (`/move`, `/undo`, `/state`)
- Leaderboard data fetching

Service Layer (GameService)

Handles:
• Turn logic
• Undo operations
• Winner/draw detection
• Player stats updating
• State validation

Repository Layer

• AppUserRepository (PostgreSQL via JPA)
• GameRepository (PostgreSQL via JPA)

Thymeleaf Templates

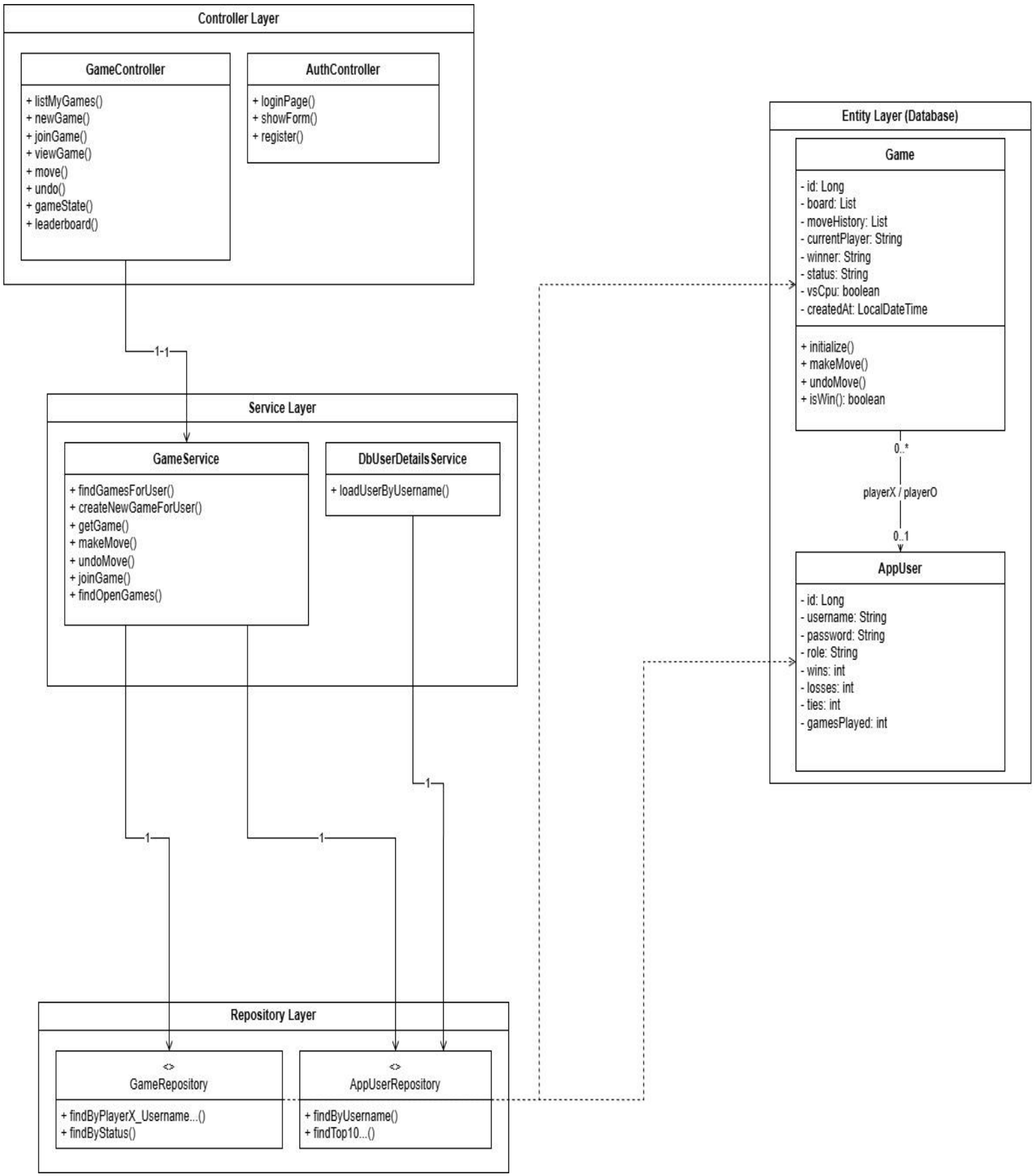
Generate:
• Dynamic HTML pages
• Board state
• Leaderboard entries
• Login/register pages

Client-Side JavaScript

Handles:
• Fetch calls
• Polling
• DOM updates
• Game-over overlay

Together, these components maintain consistent gameplay and secure interaction across multiple users.

Figure 13.: UML Class diagram showing backend connection



11. Future Enhancements and Conclusion

The main goal of this thesis was to build a solid, working architecture for a multiplayer web application. While the current version works well and delivers the core features, the project was built in a way that makes it easy to add more features later on. The following sections outline the logical next steps that the current code can support.

11.1. Future Roadmap and Scalability

- **Containerization & Cloud Deployment:** The current monolithic application is a candidate for containerization via **Docker**. This would facilitate deployment to managed cloud platforms (e.g., **AWS Elastic Beanstalk** or **Railway**), resolving the current LAN-only limitation by exposing the application via a public HTTPS endpoint.
- **Algorithmic AI Enhancement:** The current heuristic-based CPU opponent can be upgraded to a deterministic **Minimax Algorithm** with Alpha-Beta pruning. This would transform the "Single Player" mode from a random-move generator into a mathematically unsolvable opponent, providing a higher difficulty ceiling for users.
- **Security Hardening:** Transitioning from the current `NoOpPasswordEncoder` to **BCrypt** hashing (as supported by the existing `SecurityConfig` architecture) is the immediate next step for production readiness, ensuring compliance with modern data protection standards.

11.2. Conclusion

This project successfully validates the efficacy of using Java-based enterprise frameworks for interactive web applications. By strictly adhering to the **MVC architectural pattern**, the system achieves a clean separation of concerns, where business logic (Service Layer) remains isolated from the interface (Thymeleaf/JavaScript).

The implementation of **ACID-compliant transactions** via PostgreSQL proved critical in resolving concurrency issues inherent to distributed turn-based systems, effectively eliminating race conditions during simultaneous user inputs. While the current **Polling mechanism** provides a stable "Minimum Viable Product" (MVP) for Local Area Network (LAN) environments, the modular design allows for seamless future migration to event-driven architectures (e.g., WebSockets) without refactoring the core domain logic. Ultimately, this thesis serves as a proof-of-concept for scalable, secure, and persistent multiplayer application development.

12. References

[1] Oracle. (2023). *Java Platform, Standard Edition (JDK 17) Documentation*. Available: https://docs.oracle.com/en/java/javase/17/

[2] VMware Tanzu. (2024). *Spring Boot Reference Documentation*. Available: https://docs.spring.io/spring-boot/documentation.html

[3] VMware Tanzu. (2024). *Spring Security Reference Documentation*. Available: https://docs.spring.io/spring-security/reference/index.html

[4] VMware Tanzu. (2025). *Spring Data JPA Reference Documentation*. Available: https://docs.spring.io/spring-data/jpa/reference/index.html

[5] Thymeleaf. (2023). *Thymeleaf User Guide*. Available: https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

[6] PostgreSQL Global Development Group. (2024). *PostgreSQL 15 Documentation*. Available: https://www.postgresql.org/docs/15/index.html

[7] Mozilla Developer Network (MDN). (2024). *Using the Fetch API*. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[8] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD Dissertation, University of California, Irvine.

[9] Fielding, R. T., & Taylor, R. N. (2002). Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2), 115–150.

[10] Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley.

[11] Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

[12] Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill Education.

[13] Pimentel, V., & Nickerson, B. G. (2012). Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing*, 16(4), 45–53.

[14] OWASP Foundation. (2023). *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Available: https://owasp.org/www-project-cheat-sheets/