

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

**Prepared by:** Ezebuio AbbaJustice

---

## 1. Introduction

This thesis presents the design and development of a multiplayer, web-based Tic Tac Toe application built using Spring Boot, Thymeleaf, Spring Security, and JavaScript. The system supports real-time gameplay between two authenticated users, persistent player statistics, and session-based matchmaking.

Unlike a traditional single-browser implementation, this project enables multiple devices to connect to the same game session over a local network. It demonstrates a complete full-stack architecture integrating:

- Server-side rendering and MVC structure
- Secure login and user management
- RESTful APIs for game actions
- Polling-based real-time synchronization
- Database persistence of user profiles, game state, and leaderboard statistics using PostgreSQL

The application illustrates how modern Java web technologies can be combined to create an interactive, multi-user gaming environment.

---

## 2. Purpose of the Work

The purpose of this project is to demonstrate practical web application development using Spring Boot and Thymeleaf, while also addressing real-time synchronization between multiple clients and the server.

Key goals include:

- Showing correct implementation of the Model–View–Controller (MVC) pattern
- Demonstrating user authentication and authorization with Spring Security
- Implementing session-based multiplayer games
- Using REST endpoints and Fetch API for dynamic UI updates
- Persisting game and user data using JPA/Hibernate and a PostgreSQL relational database

The project also provides a foundation for extending the system into more advanced multiplayer game platforms.

---

### 3. Literature Review and Key Concepts

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

#### 3.1. 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). This separation of concerns ensures that the business logic (Service Layer) remains decoupled from the user interface (Thymeleaf Views) and data persistence (Repositories). This architectural choice is critical for maintainability, allowing the "Game Logic" to evolve independently of the frontend technology.

**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 (GET for state retrieval, POST for moves), the system achieves a stateless communication model. This enhances scalability, as the server does not need to maintain a continuous, resource-heavy socket connection for simple turn-based actions.

**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 and State Synchronization** In a distributed multi-user environment, maintaining a "Single Source of Truth" is paramount. The system addresses the "Lost Update" problem common in concurrent computing by utilizing transactional database locking (ACID properties) provided by the PostgreSQL engine. This ensures that simultaneous moves from different devices do not corrupt the game state, maintaining deterministic gameplay.

#### 3.2. Literature and Documentation Sources

The development of this project relied on the following official documentation and guidelines:

- **Spring Boot Reference Documentation** (spring.io) – For backend configuration and setup.
- **Spring Security Reference Guide** – For implementing authentication and secure access control.
- **Thymeleaf Documentation** (thymeleaf.org) – For server-side HTML rendering.

- **JPA and Hibernate ORM Guides** – For database persistence strategies.
- **RESTful API Design Principles** – For structuring client-server communication.

### 3.2. 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.

### 3.3. Testing Methods

Testing was conducted to ensure system stability and gameplay correctness. Methods included:

- **Manual Functional Testing:** Verifying all game features by hand.
- **Multi-Device Network Testing:** Connecting two distinct devices on a local network to verify real-time multiplayer synchronization.
- **REST Endpoint Verification:** Using browser DevTools and Postman to ensure the server accepts and returns the correct data.
- **Security Validation:** Testing login flows, logout functionality, and attempting to access restricted URLs without permission.
- **Game Logic Verification:** Testing edge cases such as draw conditions, undoing moves, and declaring the correct winner.

These methods were chosen because the system is small-scale, interactive, and UI-driven, making manual and network testing the most effective approach.

---

## 4. Software and Hardware Tools

### Software

- Java 17+
- Spring Boot 3+
- Spring Security
- Thymeleaf
- PostgreSQL Database
- IntelliJ IDEA
- HTML, CSS, JavaScript
- Fetch API

### Hardware

- Standard PC or laptop
- Modern web browser (Chrome, Firefox, Edge)
- Local network for multiplayer testing

---

## 5. 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
- 

## 6. Use Cases, User Stories, and Acceptance Criteria

### User Roles

- Registered Player – can log in, create games, join games, play vs Cpu and view leaderboard

## 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

## Acceptance Criteria

- Turns alternate correctly
- UI updates instantly after each move
- Both players receive final winner/draw overlay
- Undo reverts the last move correctly (Only accessible in vs Cpu mode)
- Invalid moves are rejected server-side
- Game state remains consistent across refreshes
- Leaderboard reflects player stats after every match.

### 6.1. User Stories

#### Authentication & Account Management

- US-01: As a new user, I want to register an account with a unique username and password, so that I can participate in multiplayer games and track my stats.
- US-02: As a registered player, I want to log in securely, so that I can access the game lobby and my personal dashboard.
- US-03: As a registered player, I want to log out of the system, so that I can secure my account when I am finished playing.

#### Game Management

- US-04: As a registered player, I want to create a new game as Player X, so that I can host a match and wait for an opponent to join.
- US-05: As a registered player, I want to view a list of available games and join one as Player O, so that I can play against a host who is waiting.

#### Gameplay & Synchronization

- US-06: As a player (X or O), I want to click an empty cell on the 3×3 grid during my turn, so that I can mark my move on the board.
- US-07: As a player, I want the board to update automatically when my opponent makes a move, so that I can see the current game state without manually refreshing the page.
- US-08: As a player, I want the ability to undo the last move, so that I can correct a mistake if needed.

- US-09: As a player, I want to see a visual overlay immediately when a win, loss, or draw occurs, so that I know the game has ended.

## Statistics

- US-10: As a registered player, I want to view the global leaderboard, so that I can see my ranking based on total wins and losses compared to other players.
  - US-11: As a player, I want to select "Player vs CPU" mode, so that I can practice the game without waiting for a human opponent.
  - US-12: As a player in CPU mode, I want the computer to move automatically after I complete my turn, so that the game progresses smoothly.
- 

## 7. Multiplayer Synchronization Model

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

### Polling Mechanism

Every client executes:

```
setInterval(() => {  
    fetch(`/game/state/${gameId}`);  
}, 1000);
```

The server responds with:

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

### Why Polling Works

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

### 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 state
5. Player O immediately updates UI
6. If winner is set → both clients show overlay and redirect

This guarantees synchronized, deterministic gameplay.

---

## 8. Security Layer (Spring Security)

The system uses Spring Security for:

- Login page customization
- Password encoding
- Session-based authentication
- URL protection

### Key SecurityConfig Features

- Custom login page `/login`
- Access rules:
  - `/login, /register, /css/**, /js/**` → public
  - `/game/**` → requires authentication
    - Logout via `/logout`
    - CSRF disabled only for game-related AJAX POST calls
    - Database-backed authentication through PostgreSQL

### Authentication Flow

1. User enters credentials
2. Spring Security calls `DbUserDetailsService`
3. Password is compared using the configured encoder
4. On success → redirect to `/game/mode`

This provides a secure, maintainable, and scalable foundation for multi-user interaction.

---

## 9. Database Schema

### AppUser Table (PostgreSQL)

Field	Type	Description
id	Long	Primary Key
username	String	Unique

Field	Type	Description
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

### Game Table (PostgreSQL)

Field	Type	Description
id	Long	Primary Key(Auto-generated)
board	List<String>	3x3 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

### Relationships

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

## 10. 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.

---

## 11. 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.

---

## 12. 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.

### 12.1. Future Plans and Improvements

**Fixing Access Issues (Cloud Deployment)** Right now, the application runs on a local server, meaning it is restricted to the Local Area Network (LAN). This creates a limitation where users need to know the specific IP address to connect.

- **Moving to the Cloud:** The project uses standard settings that make it ready for cloud platforms like Render or AWS. Moving it there would solve the local IP issue and give the game a public URL so anyone can access it easily.

**Improving the Game Logic and AI** The `GameService` layer keeps the game rules separate from the rest of the code. This makes it straightforward to upgrade the gameplay without breaking the existing app.

- **Minimax AI:** The current CPU is basic. It can be swapped out for the Minimax algorithm, which would make the computer "unbeatable" and provide a real challenge.

- **Puzzle Mode:** The database structure is set up to handle different types of game data. This allows for adding a "Puzzle" mode where the user has to solve specific board scenarios.

**Better Security** The project uses Spring Security, which is a standard tool for handling secure access. While the current version focuses on logging users in , the setup allows for important security upgrades:

- **Password Encryption:** The system can be updated to use BCrypt hashing for passwords, which is the industry standard for keeping user data safe.
- **Lost Password Recovery:** Since the backend is Java, it is easy to integrate email services to let users reset their passwords if they forget them.

**Real-Time Updates** The polling method used right now works fine for syncing the game . However, since the frontend uses the Fetch API, a future version could switch to WebSockets. This would make the game feel even faster by removing the need to check for updates every second.

## 12.2. Conclusion

This thesis successfully demonstrates the development of a secure, multiplayer Tic Tac Toe game using Spring Boot, Thymeleaf, and PostgreSQL. By using the MVC pattern, the project bridges the gap between storing user data and handling real-time gameplay.

The system effectively manages user logins, game states, and the database . By following these standard coding patterns, the project not only provides a working game but also creates a flexible foundation that is ready for the future upgrades mentioned above.