# Table of contents

# Introduction

The backend is implemented with **Spring Boot (Java) and PostgreSQL. PostgreSQL** was chosen for its reliability and familiarity in rapid development. Since this is not a production environment and no CI/CD pipelines are being built, no database migration tool (e.g., Flyway) is used; instead, Hibernate DDL auto is employed for simplicity.

Hexagonal architecture is implemented following the principles of clean architecture with integration tests ensuring system reliability. Lombok is used to reduce boilerplate code.
In a bigger project I would have chosen a **feature based hexagonal architecture**.

Techniques and design patterns in this project to ensure clean code include Dependency Injection with Spring, Factory pattern for model creation and Data mapper pattern to transform domain model to adapter entities.

In the database system, **UUIDs are used to support horizontal scaling**, ensuring global uniqueness and avoiding ID conflicts, particularly if a shared or distributed infrastructure is implemented in the future.

On the frontend, **React and Typescript** are used with an **atomic design architecture is used with Vite as the build tool**. Axios handles POST requests to the backend, while React-SWR is used for GET requests. Additionally, custom hooks have been implemented to manage backend communication efficiently.

Some of the 12-factor app principles have been implemented in both the backend and the application.

# Git Structure

While a single project repository was created to simplify review for the interview team, in a production environment I would have maintained separate repositories for the backend and frontend to improve modularity and maintainability.

The **main branch** is used to maintain the current state of the application, while **feature/feature-name** branches are used to implement the Tic-Tac-Toe application in its various phases.

# Design for scale horizontally

While these measures have not been implemented, In a horizontally scaled system, measures like **pessimistic locking** would prevent multiple servers from joining the same existing matchmaking game at the same time. However, it would not prevent a race condition when no matchmaking game exists and multiple servers attempt to create one simultaneously. While this could be addressed with a **uniqueness constraint or a dedicated matchmaking lock**, I chose not to implement it here to avoid overengineering for this small-scale application.

UUIDs are used as table IDs to support a potential future database with horizontal sharding and scaling

# Global Exception Handler

**GlobalExceptionHandler handles unhandled and custom exceptions,** and returns appropriate HTTP responses.  Also logs them with full stack trace when its an unexpected exception, and log as a controlled exception instead

# Logging and Observability

The GlobalExceptionHandler logs all exceptions and validation errors to provide observability. Unhandled and custom exceptions are logged with full stack traces, while validation errors are logged with field-level details. Logs use appropriate levels (ERROR) and are sent to the standard error stream for easy monitoring.
Additionally, key application events related to Games are logged with appropriate levels (INFO or WARN) to provide a clear trace of the service's behavior. All logs are sent to the standard error stream for monitoring.

If an error occurs on any endpoint(e.g., invalid input, authentication failure, resource not found), the service responds with the appropriate HTTP status code (400, 401, 403, 404, 409, 500, etc.) and a JSON error message:

```
{
   "statusMessage": "GAME_NOT_FOUND",
   "message": "Game not found"
}
```

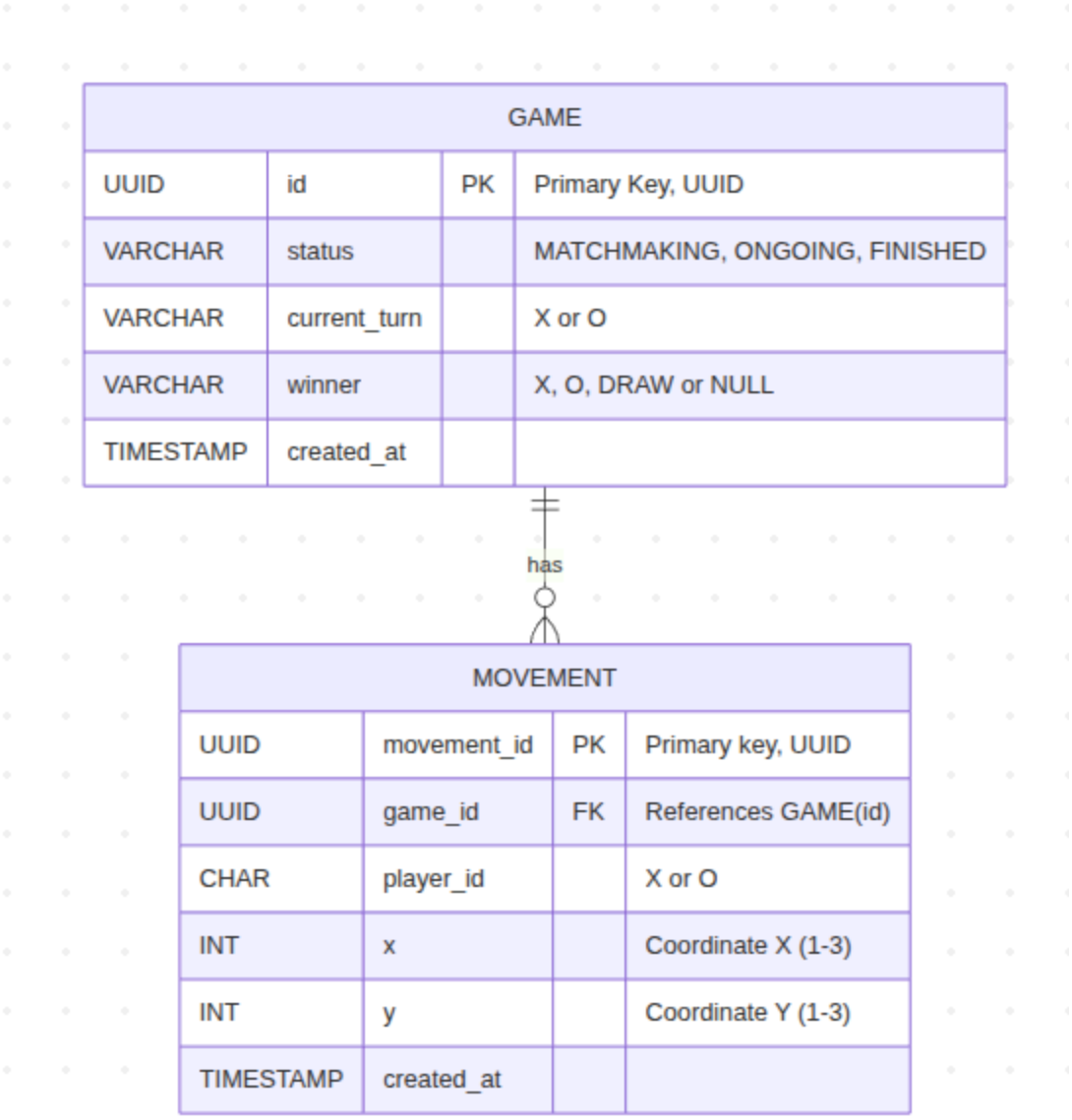
# Testing

The **backend server includes unit and integration tests using spring-boot-starter-test with an H2** in-memory database, focusing primarily on domain logic services as well as adapter services and repositories.

The **frontend application also includes unit and integration tests using Jest and React Testing Library**, with a focus on components related to the game board.

# Phase 1 - Basic gameplay functionality

Using the branch **feature/tic-tac-toe-basic-gameplay** , implement a proof-of-concept online Tic-Tac-Toe web service with basic gameplay functionality.
The game will use an anonymous identity-based architecture on the first phase, as illustrated in the following diagram




The **GAME** table represents each match and includes a **status** column to indicate the current phase of the game. This status is used for example to manage matchmaking between two players.

The **MOVEMENT** table uses an auto-generated primary key (**movement_id**) **for simplicity and performance.** Although a composite key could be created using (game_id, player_id, x, y) using a single auto-incremented id is a more optimal database indexing and query performance solution

# Phase 2 – User Management (MVP Design)

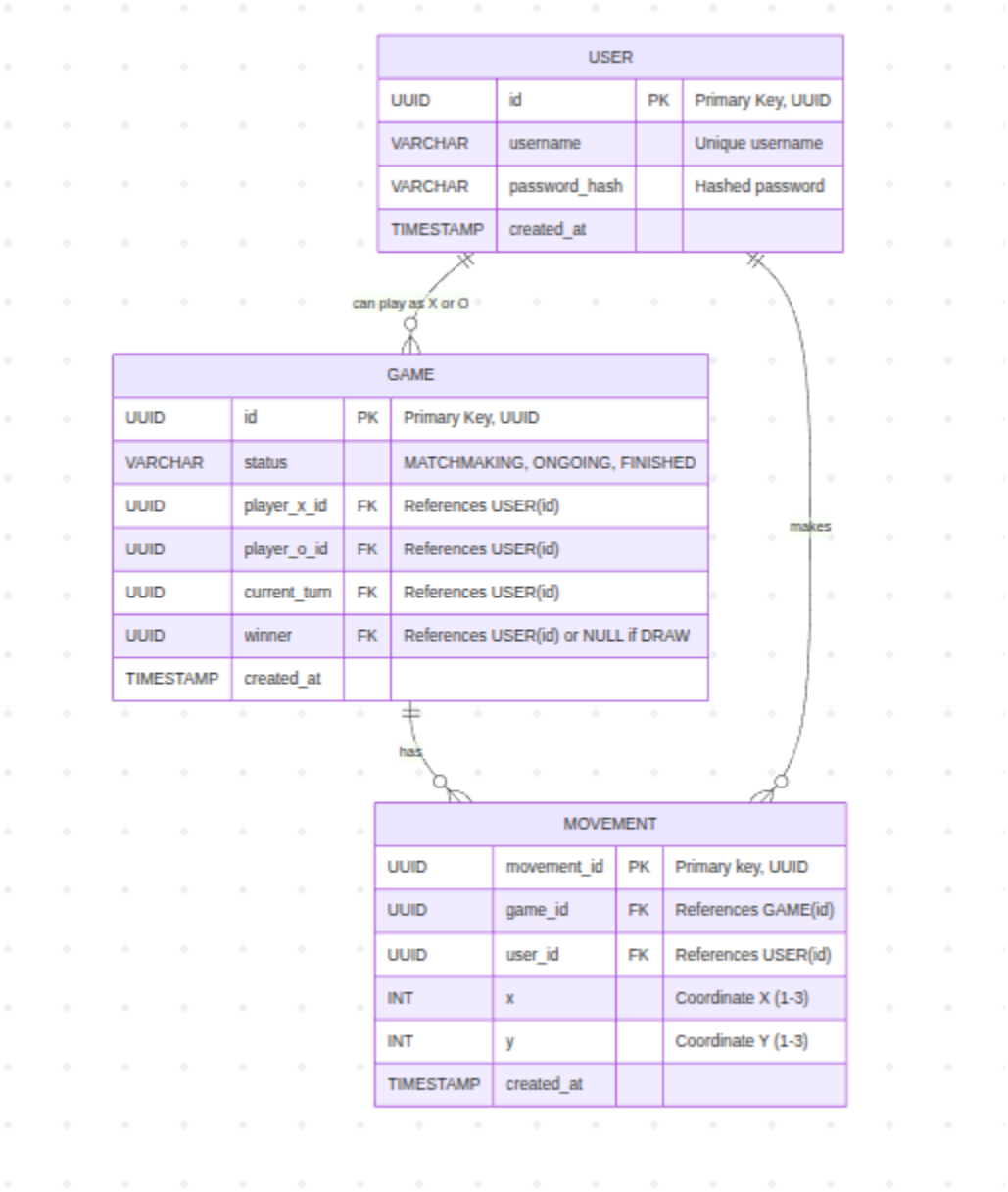
## High-level feature overview

- Register with username and password
- Authentication via jwt token
- Authenticated for protected endpoints (creating a match ,making a move)
- A foundation for future features such as leaderboards, matchmaking, multiples simultaneous games and stats.

Joel Pegueroles - Tic tac toe

# API Design

## New db diagram




1. Added User table
2. Changed game table:
    3.1 Instead of saving “X” or “O” directly, now stores the IDs of the users playing as X and O
    3.2 current_turn now points to a user ID
    3.3 Winner now points to the winning user’s id (null if it was a draw)

3.4 Added winner type

3. Changes Movement table
    3.1. Instead of saving “X” or “O”, now stores the IDs of the user made the move

## New endpoints

### /users/register

This endpoint registers a new user and provides the JWT Token
Method: POST
Json Payload:

```
{
  "username": "player1",
  "password": "mypassword"
}
```

Success status code: 201
Sucess Headers response:

```
{
  "Set-Cookie": "jwt-token=<token>"
}
```

### /users/login

This endpoint authenticates a user and provides the JWT Token
Method: POST
Json Payload:

```
{
  "username": "player1",
  "password": "mypassword"
}
```

Success status code: 201
Success Headers response:

```
{
  "Set-Cookie": "jwt-token=<token>"
}
```

## Changes existing endpoints

### /create

This endpoint will now require authentication.

Method: POST

Headers:

```
Authorization: Bearer <jwt-token>
```

Json Payload: Empty
Success status code: 200
Success Json Response:

```
{
  "matchId": "<The new match's ID>"
}
```

### /move

This endpoint will now require authentication

Method: POST

Headers:

```
Authorization: Bearer <jwt-token>
```

Json Payload:

```
{
 "matchId": "<The match's ID>",
 "square": {
  "x": "<x>",
  "y": "<y>"
 }
```

```
}
```

Success status code: 200

### Architectural Considerations:

- Authentication: JWT tokens ensure stateless sessions, allowing horizontal scaling without shared session storage
- Password storage: ecurely hashed credentials
- Extensibility: The system is designed to support future features,( multiple matches can be going simultaneously, leaderboards, ranking, stats, ….) with minor changes

# Whats missing

On the server side, several use cases and other components were not fully tested or implemented due to time constraints. With additional time, I would add integration tests covering edge cases, concurrency scenarios, and aim for full test coverage. The existing code is structured to make this extension straightforward.

On the frontend, basic UI components such as Text and Button were not tested. I would implement unit tests using React Testing Library and end-to-end tests with **Cypress** and use **Storybook** to document React components along with their multiple possible states, ensuring both reliability and maintainability.

On the feature side, I would implement a “Restart Game” button to allow the user to start a new game when the current one finishes.