UNIVERSITÀ DI PISA

Computer Engineering
Distributed Systems and Middleware Technologies

WordGuess

Project Documentation

Ayoub El Ourrak

Academic Year : 2024/2025

Table of Contents

1. Project Specifications

WordGuess is a distributed web application that enables users to play a real-time word guessing game. The system is built with two distinct components: an Erlang backend that manages the game state and logic, and a Java/Spring frontend that provides the user interface.

1.1 Requirements

Actors involved in the application are:

- **Unregistered User**
- **Registered User**

1.1.1 Functional Requirements

The functional requirements are:

- **Unregistered User**
    - Register with a username and password
    - Login with existing credentials

- **Registered User**
    - Create new games (public or private)
    - Join existing games
    - Make guesses (letters or complete words)
    - View list of available games
    - Start games (when creator)
    - Leave games
    - Visualize game state in real-time

1.1.2 Non-Functional Requirements

The non-functional requirements are:

- Strong consistency for game states and user accounts
- Real-time updates for gameplay actions
- Concurrent handling of multiple games
- Turn-based gameplay with timeouts
- Fault tolerance in case of disconnections
- Session management

1.2 Synchronization and Communication Issues

The application faces the following synchronization and communication issues:

- **Real-time Game Updates**: Players need to see the same game state simultaneously.
- **Turn Management**: Only one player should be able to make a guess at a time.
- **Concurrent Game Access**: Multiple players might try to join or leave games simultaneously.
- **Disconnection Handling**: The system must handle temporary player disconnections gracefully.
- **Timing Constraints**: Player turns need to be time-limited to prevent stalling.
- **Race Conditions**: Multiple users might attempt the same action concurrently (e.g., joining a game).

2. System Architecture

The WordGuess application follows a client-server architecture with a clear separation between the frontend and backend. The architecture is designed to handle multiple concurrent games and players while maintaining consistency and responsiveness.

2.1 Deployment Architecture

The system is deployed across two nodes:

1. **Erlang Node (10.2.1.43)**: Hosts the backend services, including:
    - Game logic implementation
    - Mnesia database
    - REST API endpoints
    - WebSocket handlers

2. **Java Spring Node (10.2.1.44)**: Hosts the frontend application, including:
    - Web server
    - JSP pages
    - Controller logic
    - API integration services

Communication between the nodes occurs through HTTP for REST API calls and WebSockets for real-time updates.

2.2 Server Side (Erlang)

The server-side architecture is built around several key components that work together to manage game states and handle client requests.

2.2.1 Main Server

The main server (`word_guess_erlang_app.erl`) is responsible for initializing the application and setting up the necessary components:

- It initializes the Mnesia database
- Defines routes for REST endpoints and WebSocket connections
- Starts the Cowboy HTTP server
- Launches the supervisor tree

The core functionality is implemented in the `word_guess_server` module, which is a GenServer that manages:

- Game creation and joining
- Player guesses
- Turn timeouts
- WebSocket connections for real-time updates

2.2.2 Mnesia Database

The application uses Mnesia for persistent storage, which is crucial for maintaining game states and user accounts even in case of system restarts:

```
-record(user, {
    username :: string(),        % Primary key
    password_hash :: binary(),   % SHA-256 hash of password
    createdAt :: integer()       % Unix timestamp
}).

-record(game, {
    id :: string(),              % Game identifier (primary key)
    word :: string(),            % Word to guess
    maskedWord :: string(),      % Word with unguessed letters masked
    maxPlayers :: integer(),     % Maximum number of players
    isPublic :: boolean(),       % Whether the game is publicly visible
    creator :: string(),         % Username of game creator
    players = [] :: [string()],  % List of player usernames
    currentPlayerIndex = 0 :: integer(), % Index of current player
    guessedLetters = [] :: [char()], % List of guessed letters
    guessedWords = [] :: [string()], % List of guessed words
    status = waiting :: waiting | in_progress | completed, % Game status
    winner :: string(),          % Username of the winner (if completed)
    error :: string(),           % Error message if any
    createdAt :: integer()       % Unix timestamp
})
```

The database is configured for persistence with disc_copies to ensure data survival across application restarts.

2.2.3 Game Management

Game logic is encapsulated in the `word_guess_game.erl` module, which handles:

- Processing letter and word guesses
- Updating game state based on guesses
- Managing turn transitions
- Detecting win conditions
- Handling player disconnections and reconnections

The system define a state for each game:

- `waiting`: Game created but not yet started
- `in_progress`: Game in active play
- `completed`: Game finished with a winner or due to all players leaving

2.2.4 WebSocket Handlers

Two WebSocket handlers manage real-time communications:

1. **Game List Handler** (`word_guess_games_handler.erl`):

    - Sends updates when games are created, started, or completed
    - Maintains a list of connected clients
    - Provides filtering for public games

2. **Gameplay Handler** (`word_guess_gameplay_handler.erl`):

    - Manages per-game WebSocket connections
    - Broadcasts state changes to all players in a game
    - Handles player actions like guesses

These handlers enable real-time, bidirectional communication which is essential for the interactive nature of the game.

2.2.5 REST API Handlers

The application exposes REST endpoints for non-real-time operations:

1. **User Handler** (`word_guess_rest_user_handler.erl`):

    - User registration
    - User login

2. **Game Handler** (`word_guess_rest_game_handler.erl`):
    - Creating games
    - Joining games
    - Getting game details
    - Starting games
    - Leaving games

The REST API provides the foundation for the frontend to interact with the backend in a structured manner.

2.3 Client Side (Java/Spring)

The client-side application is built using Spring Boot with JSP for view rendering. It handles user interactions and communicates with the Erlang backend via HTTP and WebSockets.

2.3.1 Web Application

The web application is structured using the Spring MVC pattern:

- **Controllers**: Handle HTTP requests and manage session data
- **Services**: Implement business logic and API integration
- **Models**: Represent data structures (User, Game)
- **Views**: JSP templates for rendering UI

2.3.2 ApiService

The `ApiService` class manages all communication with the Erlang backend:

```java
@Service
public class ApiService {
    // HTTP client for REST calls
    private final HttpClient httpClient = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .build();
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Value("${api.base-url}")
    private String apiBaseUrl;

    // Methods for user management
    public Map<String, Object> registerUser(User user) { ... }
    public Map<String, Object> loginUser(User user) { ... }

    // Methods for game management
    public List<Game> getPublicGames(String username) { ... }
    public Game getGame(String gameId, String username) { ... }
    public Game createGame(Game game, String username) { ... }
```

```
    public Game joinGame(String gameId, String username) { ... }
    public Map<String, Object> startGameWithResponse(String gameId,
                                    String username) { ... }
    public boolean leaveGame(String gameId, String username) { ... }

    // Helper methods
    private Map<String, Object> parseResponse(String responseBody) {...}
}
```

2.3.3 Controllers

The application defines two main controllers:

1. **UserController**: Handles user authentication and registration
    - Login form display and processing
    - Registration form display and processing
    - Session management
    - Logout processing

2. **GameController**: Manages game-related operations
    - Display available games
    - Create new games
    - Join existing games
    - View gameplay page
    - Start games
    - Leave games

These controllers coordinate between the user interface and the API service.

2.3.4 WebSocket Integration

WebSocket connections are established directly from the browser to the Erlang backend, bypassing the Java layer for real-time updates:

```
// Connect to the game list WebSocket
function connectWebSocket() {
    const wsUrlWithParams = wsUrl + '?username=' +
                            encodeURIComponent(username);
    socket = new WebSocket(wsUrlWithParams);
    socket.onopen = function(e) {
        console.log('WebSocket connection established');
    };
```

```
    socket.onmessage = function(event) {
        try {
            const message = JSON.parse(event.data);
            if (message.type === 'games_list_update') {
                updateGamesList(message.data);
            } else if (message.type === 'error') {
                showAlert(message.message, 'danger');
            }
        } catch (error) {
            console.error('Error processing message:', error);
        }
    };
}
```

2.3.5 User Interface

The user interface is built with JSP, JavaScript, and CSS. Key pages include:

1. **Login/Register Pages**: Forms for authentication
2. **Game List Page**: Displays available games with options to create or join
3. **Gameplay Page**: Interactive interface showing:
    - The masked word
    - List of players
    - Current player's turn indicator
    - Guessed letters and words
    - Input forms for making guesses
    - Turn timer display

The UI is designed to update dynamically as WebSocket messages arrive, without requiring page refreshes.

3. Synchronization and Communication Approach

The WordGuess application addresses several key synchronization and communication challenges through careful design and implementation.

3.1 WebSocket Real-time Updates

To ensure all players see the same game state:

1. **WebSocket Architecture**:
    - Game list updates via the `word_guess_games_handler`
    - Per-game updates via the `word_guess_gameplay_handler`

2. **Registration and Broadcasting**:
    - WebSocket connections register with the appropriate handler
    - Updates are broadcast to all registered connections
    - JSON is used for message serialization

3. **Client-side Update Handling**:
    - Messages are processed based on their type
    - UI elements are updated without page refreshes
    - Error messages are displayed when appropriate

This approach ensures immediate propagation of game state changes to all players.

3.2 Turn-based Gameplay Management

Turn management is implemented through:

1. **Server-side Turn Control**:
    - Each game tracks the current player index
    - Only the current player can make valid guesses
    - The server enforces turn rules atomically

2. **Turn Timeouts**:
    - A 30-second timer starts when a player's turn begins
    - If no guess is made, the turn automatically advances
    - Timers are cancelled and reset on valid guesses

3. **Turn Transitions**:
    - Correct letter guesses allow the player to continue
    - Incorrect guesses advance to the next player
    - Turn changes are broadcast via WebSockets

3.3 Disconnection Handling

The system handles player disconnections gracefully:

1. **Temporary Disconnection Detection**:
    - WebSocket closure triggers a disconnection timer
    - Players have 30 seconds to reconnect before being removed
    - The server tracks disconnection status in the `disconnect_timers` map

2. **Reconnection Handling**:
    - Players can reconnect within the time window
    - Reconnection cancels the disconnection timer
    - Game state is sent immediately upon reconnection

3. **Player Removal Process**:
    - After timeout, players are automatically removed
    - Game state is updated and broadcast
    - If sufficient players remain, the game continues
    - If not, the game is marked as completed

This approach ensures that temporary network issues don't disrupt gameplay while preventing abandoned games from stalling.

4. Implementation Details

4.1 Erlang Backend

4.1.1 Database Design

The Mnesia database schema is designed for performance and data integrity:

```
init() ->
    % Create schema if it doesn't exist
    case mnesia:create_schema([node()]) of
        ok -> io:format("Schema created successfully~n");
        {error, {_, {already_exists, _}}} -> io:format("Schema already
                                                        exists~n");
        Error -> io:format("Error creating schema: ~p~n", [Error])
    end,

    % Start Mnesia
    ok = mnesia:start(),

    % Create user table with disc_copies
    mnesia:create_table(user, [
        {attributes, record_info(fields, user)},
        {disc_copies, [node()]},
        {type, set}
    ]),

    % Create game table with disc_copies
    mnesia:create_table(game, [
        {attributes, record_info(fields, game)},
        {disc_copies, [node()]},
        {type, set}
    ]),

    % Wait for tables to be available
    mnesia:wait_for_tables([user, game], 30000),
    ok.
```

Key design choices include:

- Using `disc_copies` for persistence
- Indexing for efficient queries
- Atomic transactions for data consistency
- Proper error handling and reporting

4.1.2 Game Logic

The core game logic in `word_guess_game.erl` handles the following operations:

1. **Word Selection**:
    - Words are loaded from a JSON file in the `priv` directory
    - A random word is chosen at game creation
    - A masked version is created with underscores

2. **Letter Guessing**:
    - Players can attempt to guess a letter of the word
    - Correct letter guesses let the player play another turn
    - Incorrect letter guesses advance turn to the next player

3. **Word Guessing**:
    - Players can attempt to guess the entire word
    - Correct word guesses immediately end the game
    - Incorrect word guesses advance turn to the next player

4. **Win Condition Detection**:
    - Word completely revealed
    - Correct word guess
    - Sets the game status to `completed`
    - Records the winner

4.1.3 Request Handling

The backend uses Cowboy for HTTP and WebSocket request handling:

```
Dispatch = cowboy_router:compile([
    {'_', [
        % REST endpoints
        {"/api/users/register", word_guess_rest_user_handler,
                        [{operation, register}]},
        {"/api/users/login", word_guess_rest_user_handler,
                        [{operation, login}]},
        {"/api/games", word_guess_rest_game_handler,
                        [{operation, list}]},
        {"/api/games/create", word_guess_rest_game_handler,
                        [{operation, create}]},
        {"/api/games/:game_id", word_guess_rest_game_handler,
                        [{operation, get}]},
        {"/api/games/:game_id/join", word_guess_rest_game_handler,
                        [{operation, join}]},
        {"/api/games/:game_id/start", word_guess_rest_game_handler,
                        [{operation, start}]},
        {"/api/games/:game_id/leave", word_guess_rest_game_handler,
                        [{operation, leave}]},

        % WebSocket endpoints
        {"/ws/games", word_guess_games_handler, []},
        {"/ws/gameplay/:game_id", word_guess_gameplay_handler, []}
    ]}
]),

{ok, _} = cowboy:start_clear(
    word_guess_http_listener,
    [{ip, {0,0,0,0}}, {port, 8080}],
    #{env => #{dispatch => Dispatch}}
)
```

Key aspects of the request handling:

- Clean separation of REST and WebSocket endpoints
- Path-based routing for REST operations
- Parameter extraction from URL paths
- Operation routing based on request type and path
- JSON serialization for all responses

4.2 Java Frontend

4.2.1 Authentication and Authorization

The frontend implements authentication through:

Login Process:

```
 @PostMapping("/login")
public String processLogin(@ModelAttribute User user,
                           HttpSession session,
                           RedirectAttributes redirectAttributes) {

    Map<String, Object> response = apiService.loginUser(user);

    if ((boolean) response.get("success")) {
        // Store user data in session
        session.setAttribute("username", user.getUsername());
        return "redirect:/games";
    } else {
        // Add error message and return to login form
        redirectAttributes.addFlashAttribute("error",
                                             response.get("error"));
        return "redirect:/login";
    }
}
```

Registration Process:

1. Validation of user input
2. API call to create the user
3. Redirect to login on success

Authorization Filter:

```
@WebFilter("/*")
public class AuthorizationFilter implements Filter {
    private static final List<String> PUBLIC_PATHS = Arrays.asList(
            "/login", "/register"
    );

    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain chain)
```

```
            throws IOException, ServletException {

        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse =
(HttpServletResponse)response;

        String path = httpRequest.getRequestURI()
                      .substring(httpRequest.getContextPath().length());

        // Check if public path
        if (PUBLIC_PATHS.stream().anyMatch(path::startsWith)) {
            chain.doFilter(request, response);
            return;
        }

        // Check if user is logged in
        HttpSession session = httpRequest.getSession(false);
        boolean isLoggedIn = session != null &&
                             session.getAttribute("username") != null;

        if (isLoggedIn) {
            chain.doFilter(request, response);
        } else {
            httpResponse.sendRedirect(httpRequest.getContextPath() +
                                      "/login");
        }
    }
}
```

This ensures that only authenticated users can access protected functionality.

4.2.2 Game List Management

The game list page implements:

1. **Tab-based UI**:
    - Public games list
    - Private game joining
    - Game creation form

2. **Real-time Updates**:
    - WebSocket connection for game list changes
    - Dynamic rendering of available games
    - Status indicators for games

3. **Filtering**:
    - Shows only joinable games
    - Displays player counts
    - Indicates game status

4.2.3 Gameplay Interaction

The gameplay page implements:

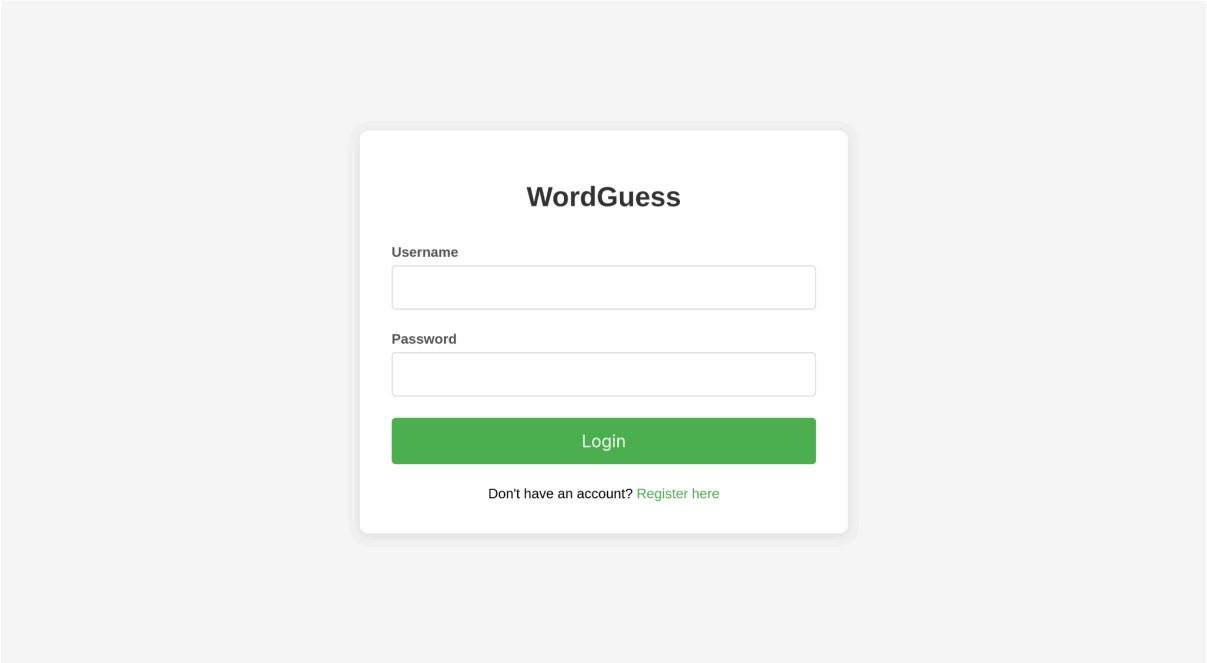
1. **Real-time Game State**:
    - WebSocket connection specific to the game
    - Updates to the masked word display
    - Player list with current player highlight
    - Turn indicator and timer

2. **Guessing Interface**:
    - Letter guessing input with validation
    - Word guessing input with validation
    - Buttons enabled only for current player

3. **Game History Display**:
    - List of guessed letters with color coding
    - List of guessed words with color coding

5. User Manual

5.1 Registration and Login



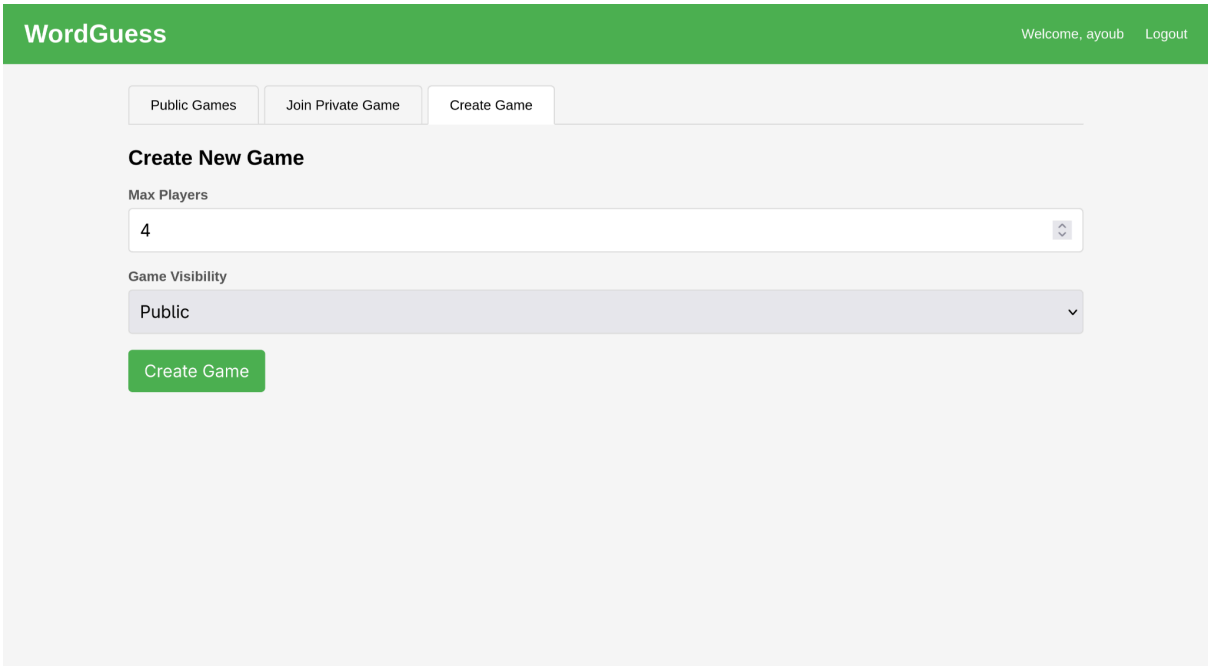
1. **Registration:**

    - Access the application at http://10.2.1.44:8081/word_guess
    - Click "Register here" to open the registration form
    - Enter a unique username and password
    - Click "Register" to create your account
    - You will be redirected to the login page

2. **Login:**

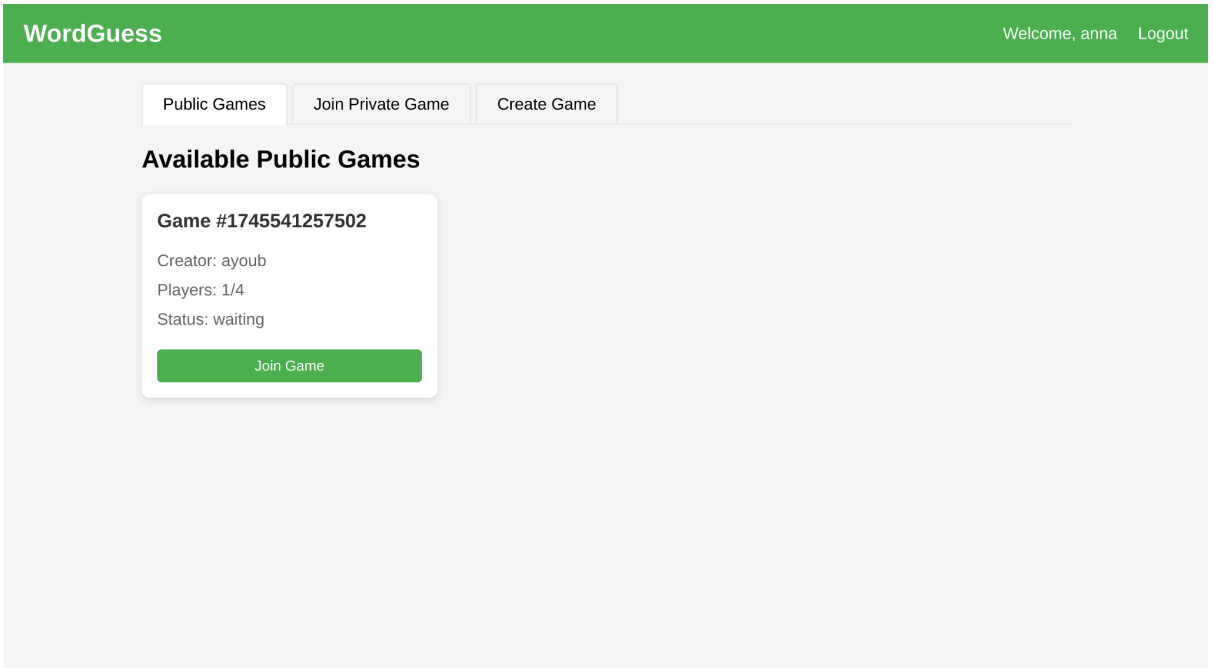
    - Enter your username and password
    - Click "Login"
    - Upon successful login, you will be redirected to the game list

5.2 Creating a Game



1. Navigate to the game list page
2. Click the "Create Game" tab
3. Configure game settings:
    - Set maximum number of players (2-50)
    - Choose public or private visibility
4. Click "Create Game"
5. You will be automatically redirected to the game page
6. Share the game ID with friends for private games

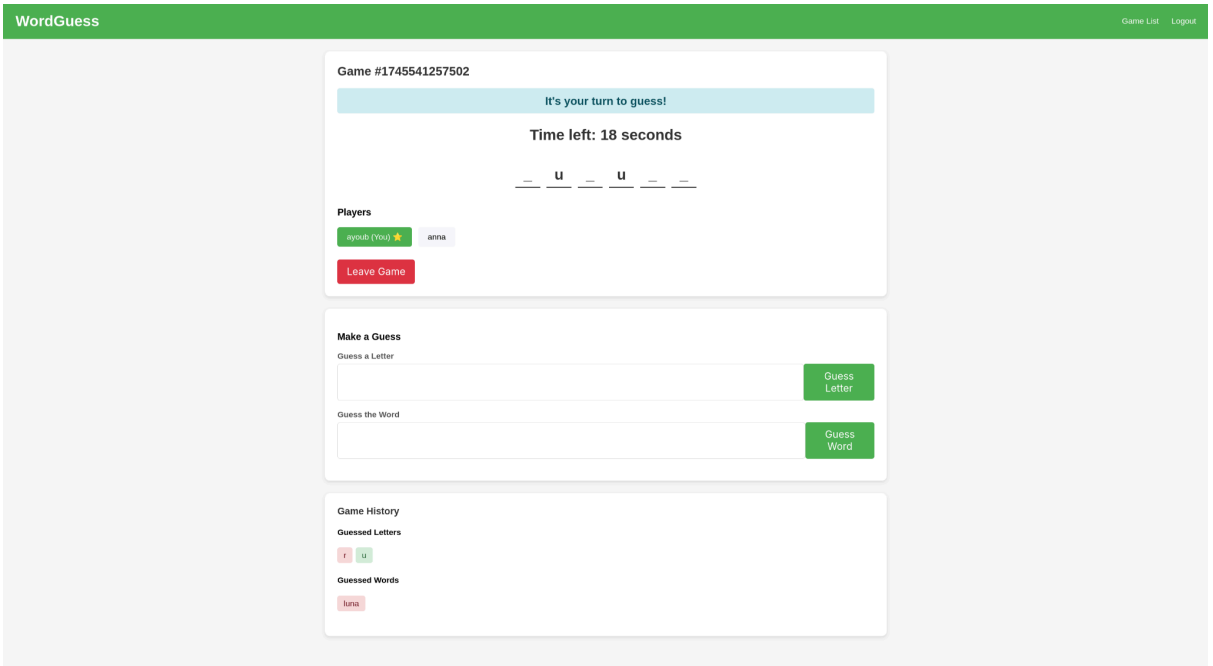
5.3 Joining a Game



1. **Joining a Public Game:**
    - Navigate to the game list page
    - Browse the available public games
    - Click "Join Game" on any available game

2. **Joining a Private Game:**
    - Click the "Join Private Game" tab
    - Enter the game code provided by the game creator
    - Click "Join Game"

5.4 Playing the Game



1. **Starting the Game:**
    - The game creator can click "Start Game" once enough players have joined
    - The game requires at least 2 players to start
2. **Taking Your Turn:**
    - When it's your turn, the "Make a Guess" section will be enabled
    - You have 30 seconds to make a guess (indicated by the timer)
    - You can either:
        - Guess a single letter: Enter a letter and click "Guess Letter"
        - Guess the entire word: Enter the word and click "Guess Word"
3. **Turn Outcomes:**
    - Correct letter: You continue your turn
    - Incorrect letter: Turn passes to next player
    - Correct word: You win the game
    - Incorrect word: Turn passes to next player
    - Timeout: Turn automatically passes to next player
4. **Winning the Game:**
    - The game ends when a player correctly guesses the entire word
    - The winning player is highlighted in the player list
    - The complete word is revealed
5. **Leaving a Game:**
    - Click "Leave Game" to exit at any time
    - If you are the last player, the game will end

References

1. Erlang Documentation: https://www.erlang.org/docs
2. Spring Boot Documentation: https://docs.spring.io/spring-boot/index.html
3. Cowboy Documentation: https://ninenines.eu/docs/en/cowboy/2.13/guide/
5. Mnesia Database: https://www.erlang.org/doc/apps/mnesia/mnesia.html