Real-Time Chess Game with AI and Matchmaking (PERN + socket.io)

Tech stack

Frontend:

- React.js
- React Chessboard
- Chess.js
- Socket.io (Client-side)

Backend:

- Node.js
- Express.js
- Socket.io (Server-side)
- Stockfish Chess Engine (AI)

Database:

- PostgreSQL

Authentication & Security:

- JWT (JSON Web Tokens)
- bcrypt.js (Password hashing)

Features

- **Real-Time Multiplayer Gameplay**
  Play live chess matches with others using Socket.io.
- **AI-Powered Chess (Stockfish Integration)**
  Challenge yourself by playing against different levels of AI difficulty.
- **Random Matchmaking System**
  Get paired instantly with a random online player through a matchmaking queue.
- **Play with Friends**
  Create private rooms and invite friends to play.
- **In-Game Chat Support**
  Communicate with opponents during gameplay via a live chat system.

- **User Authentication & Secure Login**
  Secure registration and login system powered by JWT and bcrypt.js.
- **Responsive UI**
  Clean, modern, and user-friendly design using React.js and React Chessboard.

Working

- ### stockfish integration and working

What is Stockfish?

Stockfish is a powerful open-source chess engine written in C++. It is capable of analyzing millions of positions per second and is widely used in chess applications for move suggestions, evaluations, and game analysis.

In this project, Stockfish is compiled into WebAssembly (WASM) or JavaScript so that it can run in a browser environment.

What is a Web Worker?

A Web Worker is a JavaScript feature that allows you to run scripts in a separate thread from the main UI thread. This enables you to perform computationally heavy tasks like chess engine calculations without freezing or slowing down the web page interface.

✅ Benefits of Web Workers:

- Non-blocking background processing
- Better performance for complex calculations
- Smooth and responsive user experience

How Stockfish is integrated

```
const stockfishWorker = new Worker("/stockfish.js");
```

This line creates a new Web Worker instance that loads and runs the Stockfish engine code (stockfish.js) in the background.

```
stockfishWorker.postMessage("uci");
```

After creating the worker, the "uci" command is sent to initialize the engine in UCI (Universal Chess Interface) mode.

What is UCI?

UCI (Universal Chess Interface) is a standardized protocol that allows communication between a chess engine and a graphical user interface (GUI).

In UCI mode, the engine responds to specific commands like:

- uci → Initializes the engine

- isready → Checks if the engine is ready
- position → Sets the current board position
- go → Tells the engine to calculate the best move
- bestmove → The engine’s response with the best move

Example Flow:

```
stockfishWorker.postMessage("uci");           // Initialize engine

stockfishWorker.postMessage("isready");        // Check readiness

stockfishWorker.postMessage("position startpos moves e2e4 e7e5"); // Set position

stockfishWorker.postMessage("go depth 10");     // Ask engine to think
```

You will receive engine responses using:

```
stockfishWorker.onmessage = function (event) {

  console.log("Engine says:", event.data);

};
```

What is setEngine()?

```
setEngine(stockfishWorker);
```

This is a custom function defined elsewhere in the code, used to store the Stockfish Web Worker instance in a global or component state (such as a React state or a global object). This allows the app to use this worker instance wherever needed — like sending new commands or listening to output.

Full Example Code Flow:

```
// 1. Create the Web Worker

const stockfishWorker = new Worker("/stockfish.js");



// 2. Initialize the engine in UCI mode

stockfishWorker.postMessage("uci");



// 3. Store the engine instance globally

setEngine(stockfishWorker);



// 4. Handle responses from the engine

stockfishWorker.onmessage = function (event) {
  console.log("Engine response:", event.data);

};



// 5. Use UCI commands to interact

stockfishWorker.postMessage("isready");

stockfishWorker.postMessage("position startpos moves e2e4 e7e5");

stockfishWorker.postMessage("go depth 15");
```

- ## Chessboard logic and working

State Variables

The application maintains its state using React's useState hook:

- **Game Logic**:
  - game, position - Chess logic (using Chess() library) and current board position (FEN string).
  - highlightSquares, selectedSquare - Highlights legal moves and manages selected pieces.
  - statusMessage - Status updates (e.g., "Checkmate!", "Game Started").
  - playersJoined, gameStarted - Player status and game initiation.
- **Timers**:
  - whiteTime, blackTime - Remaining time for each player.
  - intervalId - Timer interval for countdown logic.
- **Chat**:
  - chatMessages, chatInput - Messages history and input field value.

Key Functions

i. **Game State Management**
  - handleSquareClick: Validates moves, updates game state, and emits move to the server.
  - getLegalMoves: Retrieves all possible moves for a given square.
  - updateGameStatus: Checks for conditions like check, checkmate, draw, or end of the game.

ii. **Socket Event Handlers**
  - playerJoined **&** bothPlayersJoined: Notifies about players entering the room.
  - gameState: Updates the game state based on server data.
  - startGame: Starts the game upon server command.
  - gameOver: Handles game conclusion logic.

iii. **Timer Management**
  - startTimer: Begins the countdown timer for the current player's turn.
  - stopTimer: Stops the active timer.
  - handleTimeOver: Ends the game if a player's time runs out.

iv. **Chat System**
  - handleSendMessage: Sends chat messages to the server.
  - scrollToBottom: Keeps the latest message in view.

v. **Game Control**
  - handleSurrender: Allows a player to surrender, notifying the server.

- startGame: Initializes the game, resetting states and timers.

vi. **Utility**
- formatTime: Formats time for display in MM:SS format.

Socket Communication

The app uses socket.on and socket.emit for real-time event handling:

- **Listening**:
  - receiveMessage: Updates chat messages when a new message is received.
  - playerJoined, bothPlayersJoined: Updates player status.
  - gameState: Synchronizes board position and player's turn.
  - gameOver: Triggers game-over logic.
- **Emitting**:
  - pieceMove: Sends move data (from, to, roomId) to the server.
  - startGame, gameOver, surrender: Notifies the server of key game actions.

UI Components

- **Chessboard**:
  - Displays the board and interacts with handleSquareClick for moves.
  - Highlights legal moves dynamically.
- **Chat Section**:
  - Displays messages using map().
  - Provides an input field and a send button for players.
- **Control Buttons**:
  - **Start Game**: Enabled when both players are ready.
  - **Surrender**: Allows a player to forfeit the match.

Logic Flow

i. **Player Joins**:
- Players join a room using roomId.
- Updates player states via socket events.

ii. **Game Start**:
- Both players ready → Start game (via startGame).

iii. **Gameplay**:
- Players take turns; moves validated using Chess() library.
- Updates board state and player timers in real time.

iv. **Game End**:
- Checkmate, draw, time over, or surrender triggers handleGameOver.

v. **Chat**:
- Real-time communication managed via socket events.

- **Authentication logic and working**
  - **Client-Side**
    - **LoginPage**:
      - Authenticates users with email and password, saves JWT and username to localStorage, and navigates to the home page.
      - Displays alerts for invalid login attempts.
    - **RegisterPage**:
      - Allows users to sign up by submitting a username, email, and password.
      - Shows success notifications or logs errors on failure.
  - **Server-Side**

- **/register**: Handles user registration by hashing passwords with bcrypt and storing user details in the database.
- **/login**: Validates user credentials, compares hashed passwords, and generates a JWT for authenticated access.
- **/user**: Verifies JWT to fetch and return user details. Validates tokens for secure data retrieval.

- **Random Matchmaking logic and working**
  - **Purpose:** Handles matchmaking by placing users into a queue and matching them with another available player.
  - **Logic:**
    - Push the player (username and socket) into the queue.
    - When at least two players are available:
      - Remove them from the queue.
      - Check if both players are still connected.
      - Assign random sides (white or black).
      - Create a unique roomId and join both players to the room.
      - Store game state in randomGames object with a new Chess instance.
      - Emit match_found event to both players with opponent details, assigned side, and room ID.
    -
      - **Event: cancel_search**
        - **Purpose:** Removes a user from the matchmaking queue if they cancel the search.
        - **Logic:** Finds the user's socket in the queue and removes it if present.
      - **Event: disconnect**
        - **Purpose:** Handles cleanup when a user disconnects.
        - **Logic:**
          - If the user had an opponent, notifies the opponent with opponent_left event.
          - Removes user from the queue.
        - **Function:** handleRandomGameOver(roomId, message, winnerUsername, loserUsername, result)
      - **Event: randomPieceMove**
        - **Purpose:** Handles a player's chess move.
        - **Logic:**
          - Validates the room and player.
          - Verifies player is moving their own piece.
          - Performs the move using the chess.js library.
          - Emits updated FEN and turn state to both players via gameState.
      - **Event: randomSurrender**
        - **Purpose:** Handles surrender logic.
        - **Logic:**
          - Identifies winner and loser based on winnerRole.
          - Calls handleRandomGameOver() with surrender message.
          - Removes the game from memory.
      - **Event: randomStartGame**
        - **Purpose:** Starts the match after both players are connected.
        - **Logic:** Emits startGame event to the room.
      - **Event: randomSendMessage**
        - **Purpose:** Handles chat messages between players during the game.
        - **Logic:** Forwards messages from one player to the other in the same room using receiveMessage.
      - **Event: randomGameOver**
        - **Purpose:** Handles end-of-game events initiated from the client side.
        - **Logic:**
          - Identifies winner and loser by role.
          - Emits randomGameOver to the room.
          - Deletes game from memory.