SPEC-1-Draughts-Multiplayer-App

Background

The project is an online, turn-based implementation of **10x10 international draughts** (also known as Polish draughts). Initially built using HTML, CSS, and vanilla JavaScript, the goal is to transition the game to a **React-based web application** to improve scalability and maintainability. The app will support **real-time multiplayer**, **AI opponents**, **user authentication**, and **mobile deployment** via a cross-platform solution.

Requirements

🌟Must Have

- Real-time 2-player game logic for international draughts (10x10)
- React-based front-end architecture
- User authentication (login/signup)
- Game board with UI state sync between players
- Real-time communication using WebSockets (e.g. Socket.IO)
- REST API for player data (profile, history, etc.)
- AI opponent (basic difficulty level)
- Responsive design for mobile and desktop
- Deployable as a PWA (Progressive Web App)
- Match history view (games played, results)
- Profile settings with avatar and username
- Basic matchmaking (private game via invite or room code)
- Undo move (within certain rules or constraints)
- Sound effects and basic animations
- Matchmaking queue for random opponents
- Spectator mode for ongoing games
- In-game chat between players
- Tournament or leaderboard system
- Multiple AI difficulty levels
- Theming (dark/light mode, board themes)
- Notifications (turn reminders, invites)

🦜**Should Have**

💐**Could Have**

🦡**Won’t Have (in MVP)**

Method

High-Level Architecture

```
@startuml
actor Player

rectangle "Frontend (React PWA)" {
  component "Game UI"
  component "Login/Profile"
  component "Board Renderer"
  component "Matchmaker"
  component "Chat & Spectator UI"
}

rectangle "Backend (Node.js + Express)" {
  component "Auth API"
  component "Matchmaking API"
  component "Game State API"
  component "Chat Service"
  component "AI Engine"
}

rectangle "WebSocket Server (Socket.IO)" {
  component "Room Manager"
  component "Game Event Bus"
}

database "MongoDB Atlas" as DB

Player --> "Game UI"
"Game UI" --> "Login/Profile"
"Game UI" --> "Board Renderer"
"Game UI" --> "Matchmaker"
"Game UI" --> "Chat & Spectator UI"
"Login/Profile" --> "Auth API"
"Matchmaker" --> "Matchmaking API"
"Board Renderer" --> "Game State API"
"Board Renderer" --> "Room Manager"
"Room Manager" --> "Game Event Bus"
"Game Event Bus" --> "Board Renderer"
```

```
"Chat & Spectator UI" --> "Chat Service"
"Chat Service" --> DB
"Game State API" --> DB
"Auth API" --> DB
"Matchmaking API" --> DB
"AI Engine" --> "Game State API"
@enduml
```

Component Overview

Frontend (React)

- `GameBoard`: renders the 10x10 grid, interacts with game logic
- `Login`, `Signup`: user authentication forms
- `MatchScreen`: manages matchmaking and match state
- `Profile`: view/update user info
- `ChatBox`: real-time chat UI
- `SpectatorView`: for watching live games

Backend (Node.js)

- `AuthController`: login/signup/token
- `GameController`: move validation, turn logic, undo
- `MatchmakingService`: private and random match creation
- `GameSocketServer`: manages game room events via Socket.IO
- `AIEngine`: supports basic and multiple difficulty logic

Database Schema (MongoDB)

Users

```
{
  _id: ObjectId,
  username: String,
  email: String,
  passwordHash: String,
  avatarUrl: String,
  createdAt: Date
}
```

Games

```
{
  _id: ObjectId,
  player1: ObjectId,
```

```
  player2: ObjectId,
  moves: [ { from: String, to: String, timestamp: Date } ],
  status: "waiting" | "active" | "completed",
  winner: ObjectId | null,
  createdAt: Date
}
```

Messages (for chat)

```
{
  _id: ObjectId,
  gameId: ObjectId,
  senderId: ObjectId,
  text: String,
  sentAt: Date
}
```

Implementation

Step-by-Step Implementation Plan

Phase 1: Project Setup

- Initialize React app with Vite or Create React App
- Configure Tailwind CSS for styling
- Set up routing (React Router)
- Initialize Node.js + Express backend with folder structure
- Connect MongoDB Atlas
- Add Socket.IO support to both client and server

Phase 2: Authentication

- Build signup/login forms in React
- Implement JWT-based auth with Express
- Store user sessions using HttpOnly cookies or tokens
- Add protected routes on the frontend

Phase 3: Game Engine & Board Logic

- Create Draughts rules engine in JS module
- Build interactive `GameBoard` component in React
- Connect board state with move validation logic
- Enable local game vs AI (for testing)

Phase 4: Real-Time Multiplayer

- Implement `join-room`, `move`, `update` events in Socket.IO
- Sync game state across players
- Add `GameSocketServer` for handling rooms, turns, and reconnection

Phase 5: Player & Matchmaking System

- Allow players to create private games (room code)
- Add queue-based matchmaking for random opponents
- Update match status in DB

Phase 6: In-Game Chat and Spectator Mode

- Build `ChatBox` UI
- Enable `chat` and `spectate` events via Socket.IO
- Store chat logs per game session

Phase 7: User Profile and Match History

- Display past matches in profile view
- Show win/loss stats, avatars
- Allow simple profile edits

Phase 8: AI Integration

- Connect frontend to AI service endpoint
- Handle difficulty level selection
- Execute Minimax-based moves and render AI responses

Phase 9: PWA & Mobile App Wrap

- Convert React app to PWA (manifest + service worker)
- Use CapacitorJS to wrap into Android/iOS apps
- Test mobile responsiveness and deployment

Phase 10: Final Polishing

- Add sound effects, animations
- Implement settings panel (themes, notifications)
- Improve reconnection logic

Would you like to proceed to **Milestones** next?### Multiplayer Game Event Flow

To enable real-time play between two players, Socket.IO manages live game events. Here is a summary of key events and how they are handled:

WebSocket Events

Event	Payload Example	Description
`join-room`	`{ gameId, userId }`	User joins a specific game room
`move`	`{ from, to, gameId }`	A move is made by a player
`update`	`{ boardState }`	Server broadcasts updated board state
`chat`	`{ sender, message }`	Chat messages exchanged in-game
`disconnect`	`{ userId }`	Handle temporary disconnection
`reconnect`	`{ userId, gameId }`	Rejoin and resync the game session
`spectate`	`{ gameId }`	User joins as a viewer

WebSocket Game Flow

```
@startuml
actor Player1
actor Player2
participant "Socket.IO Server" as WS
participant "Game State API" as API

Player1 -> WS: join-room
Player2 -> WS: join-room
WS -> Player1: update (initial state)
WS -> Player2: update (initial state)

Player1 -> WS: move (E3 to F4)
WS -> API: validate + save
API --> WS: new board state
WS -> Player1: update
WS -> Player2: update

Player2 -> WS: move
...

Player1 -> WS: chat message
WS -> Player2: chat message

Player1 -> WS: disconnect
Player1 -> WS: reconnect
@enduml
```