Project Plan: Skribbl.io Clone

This document outlines the core concepts, recommended technology stack, and a phased development plan for creating a real-time drawing and guessing game inspired by Skribbl.io.

1. Core Concepts Involved

At its heart, Skribbl.io is a game built on real-time, many-to-many communication. Here are the fundamental pillars:

- **Real-time Communication:** The most critical aspect. When one user draws, every other user in the room must see it appear on their screen instantly. Chat messages, score updates, and game state changes (like the timer) must also be synchronized in real-time. This is typically handled by **WebSockets**.
- **Game State Management:** The server is the single source of truth. It needs to manage the state for each game room independently, including:
    - The list of players in the room.
    - The current player who is drawing.
    - The secret word being drawn.
    - The current round number.
    - The score for each player.
    - The round timer.
- **Lobby / Room System:** Players don't play in one massive global game; they are isolated in rooms. The system needs to allow users to create new rooms or join existing ones, each with its own separate game state.
- **Turn-Based Round Logic:** The game proceeds in rounds. The server needs logic to:
    - Start the game once enough players have joined.
    - Rotate the "drawer" role among players each round.
    - Provide the drawer with a choice of words.
    - Manage the round lifecycle (start timer, end round, show scores).
- **Shared Drawing Canvas:** A digital canvas where one user's input (mouse movements, color changes, brush size) is captured and broadcast to all other users in the room to be redrawn on their canvases.
- **Guessing & Chat System:** A chat interface where players submit their guesses. The server must intercept these messages to:
    - Check if a guess matches the secret word.
    - If correct, award points and notify the room without revealing the word.
    - If incorrect, broadcast the message to the room as a normal chat message.

2. Optimal Tech Stack

This stack is chosen for its strong real-time capabilities, developer productivity, and extensive community support.

- **Backend:**
    - **Runtime: Node.js**. Its event-driven, non-blocking architecture is perfect for handling many simultaneous WebSocket connections efficiently.
    - **Framework: Express.js**. A minimalist and flexible framework for setting up the web server and any potential API endpoints (e.g., for creating rooms).
    - **Real-time Engine: Socket.IO**. This library simplifies working with WebSockets immensely. It provides features like rooms, automatic reconnection, and fallbacks out-of-the-box, which are perfect for this project.
- **Frontend:**
    - **Framework: React (or Next.js)**. A modern component-based framework that makes it easy to manage the UI state, which will be constantly changing based on events from the server.
    - **Canvas API:** The native **HTML `<canvas>` API**. You will use JavaScript to listen for mouse events on the canvas, draw lines, and handle drawing data received from the server.
    - **Styling: Tailwind CSS**. For rapidly building a modern and clean user interface without writing a lot of custom CSS.
- **Database (Optional - for later stages):**
    - **MongoDB** or **Redis**. Initially, you can store game state in server memory. If you want to add features like user accounts, persistent settings, or custom word packs, a NoSQL database like MongoDB or a fast in-memory store like Redis would be a great choice.

3. Project Breakdown (Backend First)

Here is a step-by-step plan to build your game, starting with the backend logic.

Part 1: The Backend Foundation & WebSocket Server

Goal: Set up a basic server that can accept WebSocket connections.

1. **Initialize Project:** Create a new Node.js project (`npm init -y`).
2. **Install Dependencies:** `npm install express socket.io`. Also, install `nodemon` as a dev dependency for automatic server restarts.
3. **Create Express Server:** Write a basic `index.js` file to create an Express server that listens on a port.
4. **Integrate Socket.IO:** Attach Socket.IO to your Express server.

5. **Handle Connections:** Set up a `connection` event listener. Inside, log a message whenever a new user connects and a `disconnect` event when they leave. This confirms your real-time channel is working.

Part 2: Room & Lobby System (Server-Side)

Goal: Implement the logic for players to join and be grouped into isolated game rooms.

1. **Room Management Data Structure:** On your server, create an object or a `Map` to hold the state of all active rooms (e.g., `const rooms = new Map();`).
2. **Join Room Logic:** Create a Socket.IO event listener for `"joinRoom"`. When a player emits this event, use `socket.join(roomId)` to add them to a room.
3. **Store Player Info:** Update your `rooms` data structure to track which players are in which room.
4. **Broadcast Events:** When a player joins or leaves a room, broadcast a message *only to clients in that room* to notify them. For example: `io.to(roomId).emit('playerJoined', playerInfo)`.
5. **Handle Disconnects:** When a player disconnects, make sure to remove them from the room's state and notify the other players in that room.

Part 3: The Core Game Logic (Server-Side)

Goal: Implement the turn-based logic, word selection, and guessing mechanism.

1. **Word List:** Create a simple array of words on the server.
2. **Start Game Logic:** Create a `"startGame"` event. When triggered by a player (e.g., the room host), the server should initialize the game state for that room (set round to 1, scores to 0, etc.).
3. **Turn Management:** Write a function that selects the first player as the drawer and starts the first round.
4. **Word Selection:** Send a few random words from your list to the drawing player using a private message (e.g., `io.to(drawerSocketId).emit('wordChoice', words)`).
5. **Timer:** Once the drawer chooses a word, start a countdown timer on the server (`setInterval`). Broadcast a "tick" event every second to all players in the room.
6. **Guessing Logic:** Create a `"chatMessage"` event listener.
    - Check if the message content matches the room's secret word (be sure to make the check case-insensitive and trim whitespace).
    - If it matches, calculate points, update the score, and broadcast a `"correctGuess"` event.
    - If it's wrong, just broadcast the message as a normal chat message.

7. **End of Round:** When the timer reaches zero or all players have guessed correctly, end the round. Broadcast a "round over" summary, then start the next round with a new drawer.

Part 4: Drawing Data Relay

Goal: Make the server relay drawing coordinates from the drawer to all other players in the room.

1. **Create Drawing Event:** Set up a listener for a `"drawing"` event. This event will carry a payload with drawing data (e.g., start coordinates, end coordinates, color).
2. **Broadcast Drawing Data:** When the server receives a `"drawing"` event from a client, it should immediately broadcast that same event and payload to *every other client* in the same room. Use `socket.to(roomId).emit('drawing', data)`. The server doesn't need to understand the drawing data; it just acts as a high-speed relay.
3. **Handle 'Clear Canvas':** Create a simple `"clearCanvas"` event that, when received, is broadcast to the whole room.

4. Ideas for Your Own Twists

Once you have the core game working, here are some ideas to make it unique:

- **New Game Modes:**
    - **Speed Mode:** Rounds are only 30 seconds long.
    - **One Line:** You can only draw one continuous line without lifting the "pen."
    - **Blind Mode:** The drawer can't see what they are drawing.
- **Customization:**
    - Allow players to create and use their own **custom word packs**.
    - Implement **private, password-protected lobbies**.
    - Add simple avatar customization.
- **Advanced Features:**
    - **User Accounts:** To track stats, achievements, and friends (this would require a database).
    - **Hint System:** The server can reveal a letter from the secret word every 20 seconds.
    - **Drawing Replays:** At the end of a round, quickly replay the drawing process.