Multiplayer Tic-Tac-Toe Setup Guide

Project Structure

```
multiplayer-tictactoe/
├── client/                # React app
│   ├── src/
│   │   ├── App.jsx
│   │   └── main.jsx
│   ├── package.json
│   └── vite.config.js
└── server/                # Express WebSocket server
    ├── server.js
    └── package.json
```

Server Setup

1. Create server directory and package.json

bash

```
mkdir server
cd server
npm init -y
```

2. Install dependencies

bash

```
npm install express ws
```

3. Server package.json

json

```
{
  "name": "tictactoe-server",
  "version": "1.0.0",
  "description": "WebSocket server for multiplayer tic-tac-toe",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "ws": "^8.14.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

4. Run the server

```
bash

npm start
```

The server will run on `http://localhost:8080`

Client Setup

1. Create React app with Vite

```
bash

npm create vite@latest client -- --template react
cd client
npm install
```

2. Replace App.jsx with the provided code

3. Update vite.config.js (optional - for proxy)

```
javascript
```

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    port: 3000
  }
})
```

4. Run the client

```
bash

npm run dev
```

The client will run on `http://localhost:3000`

How to Play

1. **Start the server** (Terminal 1):

```
bash

  cd server
  npm start
```

2. **Start the client** (Terminal 2):

```
bash

  cd client
  npm run dev
```

3. **Open two browser windows/tabs**:
   - Window 1: `http://localhost:3000`
   - Window 2: `http://localhost:3000`
4. **Create/Join a room**:
   - Both players enter the same room ID (e.g., "room1")
   - Click "Join Room"
   - First player becomes X, second player becomes O
5. **Play the game**:
   - X goes first

- Players take turns clicking squares
- Winner is announced when someone gets 3 in a row
- Click "Play Again" to reset the board
- Click "Leave Room" to exit

Features

✅ Real-time multiplayer gameplay
✅ Automatic room creation
✅ In-memory room management
✅ Rooms auto-delete when empty
✅ Turn validation
✅ Winner detection
✅ Draw detection
✅ Player disconnect handling
✅ Clean UI with Tailwind CSS

API Endpoints

- **GET /rooms** - View all active rooms (debugging)

```bash
curl http://localhost:8080/rooms
```

WebSocket Messages

Client → Server

```javascript
// Join room
{ type: "join", roomId: "room1" }

// Make move
{ type: "move", roomId: "room1", index: 0 }

// Reset game
{ type: "reset", roomId: "room1" }

// Leave room
{ type: "leave", roomId: "room1" }
```

Server → Client

```javascript
// Joined successfully
{ type: "joined", player: "X", playerCount: 1 }

// Player joined room
{ type: "playerJoined", playerCount: 2 }

// Game state update
{
  type: "gameState",
  board: [...],
  currentTurn: "X",
  winner: null
}

// Player left
{ type: "playerLeft", playerCount: 1 }

// Error
{ type: "error", message: "Room is full" }
```

Troubleshooting

WebSocket connection fails:

- Ensure server is running on port 8080
- Check firewall settings
- Try `ws://127.0.0.1:8080` instead of localhost

Room not working:

- Both players must use exact same room ID (case-sensitive)
- Room IDs are temporary and deleted when empty
- Check server console for error messages

Player can't make moves:

- Wait for both players to join
- Ensure it's your turn
- Check that square isn't already occupied

Production Deployment

For production, you'll need to:

1. Use environment variables for WebSocket URL
2. Deploy server to a hosting service (Heroku, Railway, etc.)
3. Update client WebSocket URL to use `wss://` (secure WebSocket)
4. Consider adding authentication
5. Use a proper database (Redis, MongoDB) instead of in-memory storage