🚀 Chess Game Backend — app.js Fully Explained

---

1. Importing Packages

const express = require("express");
Loads the Express framework. This is used to build your HTTP server and define routes (like serving the homepage or static assets).

const socket = require("socket.io");
Imports the Socket.IO library — used to enable real-time, event-based communication between browser and server.

const http = require("http");
Brings in Node’s built-in HTTP module. You use this to create a raw HTTP server, which is required to integrate Socket.IO with Express.

const {Chess} = require("chess.js");
Imports the core Chess engine from the chess.js library. This gives you access to a full chess rules engine: move legality, check/checkmate detection, FEN notation, etc.

const path = require("path");
Imports Node’s built-in path module. It helps resolve absolute file paths, like when serving your frontend files with Express.

---

2. Creating the Server and Setting Up Socket.IO

const app = express();
Initializes an Express application. This becomes your base web server.

const server = http.createServer(app);
Wraps your Express app into a raw HTTP server. This step is needed because Socket.IO cannot attach directly to Express; it needs a pure HTTP server to bind with.

const io = socket(server);
This creates a Socket.IO instance on top of your HTTP server. You now have the ability to listen for and emit WebSocket events.

---

3. Initial Game Variables

const chess = new Chess();
Creates a new game instance using chess.js. This holds the entire game state — board position, who's turn it is, history of moves, etc.

let players = {};
This object will hold the connected players. You store their socket IDs here: one for the white player, one for black.

let currentPlayer = "w";
This string tracks whose turn it is: "w" for white, "b" for black.

let gameStarted = false;
This boolean tracks whether the game has officially started. Until both players are connected, it stays false.

---

4. Configuring EJS and Static Files

app.set("view engine", "ejs");
Tells Express to use EJS (Embedded JavaScript) as the view engine. This lets you embed dynamic values (like <%= title %>) inside HTML.

app.use(express.static(path.join(__dirname, "public")));
Configures Express to serve static files from the /public directory — including CSS, JS, and images. __dirname refers to the current directory of the app file.

---

5. Defining the Homepage Route

app.get("/", (req, res) => { res.render("index", {title: "Chess game"}); });
This sets up the homepage route. When a user opens your site at /, Express renders the index.ejs file and passes a title variable (Chess game) to be displayed on the page.

---

6. Socket.IO Connection Handler

io.on("connection", function(uniquesocket){ ... });
This function runs every time a new client connects to the server. The uniquesocket represents that user's unique WebSocket connection.

Here's what happens inside this block:

A. Assigning Roles

if(!players.white){ players.white = uniquesocket.id; uniquesocket.emit("playerRole", "w"); }
If there's no white player yet, the connecting user becomes white. You store their socket ID and emit "playerRole" with value "w" to tell them their color.

else if(!players.black){ players.black = uniquesocket.id; uniquesocket.emit("playerRole", "b"); }
If white is taken but black is empty, assign the user as black and emit "playerRole" with "b".

else{ uniquesocket.emit("spectatorRole"); }
If both roles are filled, any further user is treated as a spectator — they can see the board, but cannot play.

After assigning roles, you handle **player disconnections**:

uniquesocket.on("disconnect", function(){ ... })
If a user disconnects:

- You check if their socket ID matches the white or black player.
- If yes, you delete them from the players object.

Additionally:
if (Object.keys(players).length === 0) { gameStarted = false; }
This means: if both players leave the game, reset the gameStarted flag to stop any future moves from being accepted until a new game starts.

🟡 Handling Moves from Clients

When a player tries to move a piece, you listen via:
uniquesocket.on("move", (move) => { ... })

Inside this block:

1. You **check if the game has started**. If not, return.
2. You ensure **the correct player is making the move** based on turn and socket ID.
3. You **validate and apply the move** using chess.move(move). If valid:
    - Update the currentPlayer.
    - Emit the "move" event to **all** clients.
    - Emit the "boardState" (FEN string) so all clients update their internal board view.

If the move is invalid, the player is notified via:
uniquesocket.emit("invalidMove", move);

A try-catch is used to handle any exceptions (e.g., illegal syntax, piece not found).

🟢 Game Start Trigger

You have this condition:
if (Object.keys(players).length === 2 && !gameStarted) { ... }

This checks:

- If both white and black players are connected
- AND the game hasn't started yet

If true:

- The game is marked as started
- The server sends out:
    - "gameStarted" → shows message like "Enjoy the game!"
    - "boardState" → sends the full current board using FEN

🟣 Server Listening

server.listen(3000, function(){ console.log("listening on 3000") })
This boots the server at port 3000 and logs confirmation. You can now visit http://localhost:3000 to play.

🟣 **index.ejs — Frontend Template (UI Layer)**

HTML structure starts with:

<!DOCTYPE html> defines the document type as HTML5.
<html lang="en"> starts the HTML tag with English as the language.

Inside <head>, you set metadata like charset and viewport.
<title><%= title %></title> dynamically inserts the title passed from Express. In your server, you passed "Chess game".

Next:
<script src="https://cdn.tailwindcss.com"></script> loads Tailwind CSS — a utility-first CSS framework. You use Tailwind for your layout, spacing, colors, and responsiveness.

Then you define a large <style> block. This contains CSS for:

- .container: wraps the chessboard and move logs in a flex layout.
- .chessboard: creates an 8x8 square grid with display: grid.
- .square.light and .square.dark: define the alternating colors of the board.
- .piece.white and .piece.black: control font color and effects for white/black pieces.
- .flipped: rotates the board 180° for black player.

Inside <body class="bg-zinc-900 ...">, you define three core sections:

1. **Status Message**
    - <div id="statusMessage">Waiting for the opponent...</div> is used to dynamically show messages like "White's Turn", "Checkmate", etc.
2. **Game Container**
    - <div class="container"> wraps the main content.
    - Inside it:
        - <div id="whiteMovesContainer"> and <div id="blackMovesContainer"> each show a scrollable move log for black's and white's moves respectively using <ul> tags.
        - <div class="chessboard"> is where the board UI will be generated dynamically by JavaScript.
3. **Scripts**
    - You load:

  - socket.io.min.js from CDN — required for connecting to your server
  - chess.js from CDN — frontend validation + piece identification
  - /js/chessgame.js — your custom logic file

🧠 **Full Explanation — chessgame.js (Line by Line)**

---

const socket = io();
This automatically connects your browser to the server using Socket.IO. You don't need to specify the server URL when it's on the same host.

---

const chess = new Chess();
Creates a new instance of the chess game using the chess.js library. This object manages all board logic: piece positions, legality of moves, detecting checkmate, and more.

---

const boardElement = document.querySelector(".chessboard");
Grabs the main board container from the DOM — this is where you'll dynamically build the 8x8 grid with pieces.

---

const whiteMovesElement = document.getElementById("whiteMoves");
const blackMovesElement = document.getElementById("blackMoves");
These two hold the move logs — a list of all moves made by black and white respectively.

---

const statusMessage = document.getElementById("statusMessage");
This element displays messages like "Waiting for opponent", "Game Started", or "Checkmate".

---

let draggedPiece = null;
let sourceSquare = null;
let playerRole = null;
These variables store temporary game state during drag-and-drop:

- draggedPiece stores the DOM element being dragged
- sourceSquare holds its coordinates
- playerRole holds the color assigned to this user ("w", "b", or null for spectator)

🔁 **renderBoard() function**

This function generates the visual board every time the game updates.

It uses chess.board() which returns an 8x8 array representing current positions.

Inside nested forEach loops, you:

- Create a <div> square for each cell
- Assign it light or dark class depending on its coordinates
- If the square has a piece, you:
    - Create a .piece div
    - Add white or black class
    - Insert the correct Unicode symbol using getPieceUnicode(piece)
    - Enable dragstart and dragend events if the piece belongs to the current player

Each square also listens to dragover and drop. When a piece is dropped, it calls handleMove(source, target).

Finally, if the player is black, you apply a .flipped class to rotate the board 180 degrees for their POV.

🔁 handleMove(source, target)

This function constructs a move object like this:

- from: calculated by converting col and row to chess notation (e.g., e2)
- to: the same conversion for target square
- promotion: "q": automatically promotes pawns to queens when reaching the last rank

Then it emits "move" to the server using socket.emit("move", move).

♟ getPieceUnicode(piece)

Maps each piece.type to a Unicode symbol like:

- p → ♙ (white pawn)
- r → ♖
- k → ♔
  This makes the UI more visual and intuitive.

✅ checkForGameOver()

Checks if the current board state is checkmate using chess.in_checkmate().
If true, displays the winner in the status bar using the current turn.

🌐 Socket.IO Event Handlers

socket.on("playerRole", function(role){ ... })
Sets playerRole to "w" or "b" and shows the correct message in statusMessage. It then renders the board from that player's perspective.

socket.on("spectatorRole", function(){ ... })
Sets playerRole = null and disables drag logic. Also updates the UI to say "Spectators cannot participate".

socket.on("boardState", function(fen){ ... })
Receives a full FEN string from the server and loads it into chess, then calls renderBoard().

socket.on("move", function(move){ ... })
Applies the move to your local chess instance, re-renders the board, updates move logs, and checks for checkmate.

socket.on("gameStarted", function() { ... })
Updates the message bar to say "Enjoy the game!"

📝 updateMoveLogs(move)

Retrieves the piece at the destination square using chess.get(move.to) and builds a readable string like:
♘ e2 to f4

Then creates a <li> element and appends it to the correct list (whiteMoves or blackMoves) based on turn.

🟢 At the end: renderBoard();

This ensures the board is drawn as soon as the page loads, even before any socket events.
