Here is a high-level design (HLD) for building a **Real-Time Collaborative Whiteboard** using the **MERN Stack**:

---

1. Architecture Overview

Tech Stack:

- **Frontend**: React.js with socket.io-client for real-time communication.
- **Backend**: Node.js, Express.js, and socket.io for WebSocket handling.
- **Database**: MongoDB for user management and optional session persistence.
- **Authentication**: JSON Web Tokens (JWT) for session management.

Key Components:

1. **WebSocket Server**:
   - Handles real-time communication using `socket.io`.
   - Manages drawing synchronization, user sessions, and notifications.
2. **Frontend Application**:
   - Provides an interactive whiteboard UI.
   - Uses `socket.io-client` to communicate with the WebSocket server.
3. **API Server**:
   - Provides RESTful APIs for user authentication and session management.
   - Stores user and session data in MongoDB.
4. **Database**:
   - **Users** collection for authentication and user management.
   - **Sessions** collection (optional) for saving whiteboard states.

---

2. System Workflow

User Authentication:

1. User logs in or registers via a React-based UI.
2. Credentials are sent to the Express server, which verifies them against MongoDB.
3. If authenticated, the server generates a JWT and sends it to the client.
4. Client attaches the JWT in the WebSocket handshake headers for authentication.

Real-Time Whiteboard:

1. User connects to a unique whiteboard session via WebSocket.
2. When a user performs an action (draw/erase), the client emits a `drawing` event to the WebSocket server.
3. The WebSocket server broadcasts the drawing event to all connected clients.
4. Each client updates the canvas in real-time.

Session Management:

1. Users can create or join a whiteboard session by specifying a session ID.
2. The server maintains a list of active sessions and connected users.
3. When a user joins/leaves a session, the server emits `userJoined` or `userLeft` events to all participants.

Optional State Persistence:

1. Whiteboard states (e.g., drawings, colors) are stored in MongoDB.
2. New users joining an existing session retrieve the saved state and sync with the ongoing session.

---

3. Key Features Design

WebSocket Server:

1. **Events**:
   - `connection`: Authenticate the user and establish a connection.
   - `drawing`: Broadcast drawing data (coordinates, color, tool) to all users in the session.
   - `clearCanvas`: Clear the canvas for all users.
   - `userJoined` / `userLeft`: Notify users about session changes.
2. **Rooms**:
   - Each whiteboard session is mapped to a unique room.
   - Users in the same room receive synchronized updates.

Frontend (React):

1. **Canvas UI**:
   - Built using the HTML5 `<canvas>` element or a library like `react-konva`.
   - Allows drawing, erasing, and color selection.
   - Emits drawing actions to the WebSocket server.
2. **User Authentication**:
   - Login/Signup forms.
   - Stores JWT in `localStorage` or `sessionStorage`.
3. **WebSocket Client**:
   - Establishes a connection with the WebSocket server.
   - Listens for real-time updates (e.g., drawing, user events).
4. **State Management**:
   - Uses React Context or Redux to manage user, session, and drawing states.

Backend (Node.js + Express.js):

1. **Authentication**:

    - Routes for login (`POST /login`) and registration (`POST /register`).
    - Middleware to validate JWTs for WebSocket and RESTful API requests.

2. **Session Management**:

    - Routes for creating (`POST /session`) and joining (`GET /session/:id`) sessions.
    - Stores session metadata in MongoDB.

3. **WebSocket Server**:

    - Handles socket connections, user authentication, and real-time events.

Database Schema (MongoDB):

1. **Users Collection**:

    ```
    {
      "_id": "userId",
      "username": "string",
      "password": "hashed_string",
      "email": "string"
    }
    ```

2. **Sessions Collection**:

    ```
    {
      "_id": "sessionId",
      "users": ["userId1", "userId2"],
      "canvasState": [
        {
          "action": "draw",
          "color": "string",
          "coordinates": [{ "x": "number", "y": "number" }]
        }
      ]
    }
    ```

---

4. Real-Time Communication Flow

1. **Client Side**:

    - Emit events like `drawing`, `clearCanvas`, and `joinSession` to the WebSocket server.
    - Listen for updates (`drawing`, `userJoined`, etc.) to update the UI.

2. **Server Side**:

    - Authenticate users during the WebSocket handshake using JWT.
    - Manage sessions and broadcast updates to users in the same session.

---

5. API Endpoints

Authentication APIs:

1. **POST /register**:
    - Input: `username`, `email`, `password`.
    - Output: Success message or error.
2. **POST /login**:
    - Input: `email`, `password`.
    - Output: JWT token.

Session APIs:

1. **POST /session**:
    - Input: Session details.
    - Output: Session ID.
2. **GET /session/:id**:
    - Input: Session ID.
    - Output: Session details (e.g., participants, canvas state).

---

6. Security Measures

1. **Authentication**:

    - Use JWT for user session management.
    - Secure WebSocket connections with tokens.

2. **Data Validation**:

    - Validate all inputs (e.g., user credentials, session data).

3. **Rate Limiting**:

    - Implement rate limiting for API endpoints and WebSocket events to prevent abuse.

4. **Secure Data Transmission**:

    - Use HTTPS and WSS for encrypted communication.

---

7. Deployment Strategy

1. **Frontend**:

    - Deploy the React app on a platform like Vercel or Netlify.

2. **Backend**:
    - Deploy the Node.js server on platforms like AWS, Heroku, or Render.
    - Use MongoDB Atlas for a managed database solution.
3. **WebSocket Scalability**:
    - Use a load balancer like Nginx or a WebSocket-compatible cloud service (e.g., AWS Elastic Beanstalk, Socket.io with Redis).

This HLD outlines a modular, scalable approach for developing a real-time collaborative whiteboard with authentication, session management, and WebSocket-based updates.