# Seeiding the users

**import { config } from "dotenv";**

**import { connectDB } from "../lib/db.js";**

**import User from "../models/user.model.js";**

**config();**

**const seedUsers = [**

**{**

**email: "alexander.martin@example.com",**

**fullName: "Alexander Martin",**

**password: "123456",**

**profilePic: "https://randomuser.me/api/portraits/men/6.jpg",**

**},**

**{**

**email: "daniel.rodriguez@example.com",**

**fullName: "Daniel Rodriguez",**

**password: "123456",**

**profilePic: "https://randomuser.me/api/portraits/men/7.jpg",**

**},**

**];**

**const seedDatabase = async () => {**

**try {**

**await connectDB();**

**await User.insertMany(seedUsers);**

**console.log("Database seeded successfully");**

**} catch (error) {**

**console.error("Error seeding database:", error);**

**}**

**};**

**// Call the function**

**seedDatabase();**

****node src/seeds/user.seed.js****

# Socket IO implementation in node

### Socket.js for importing socket

*import {Server} from "socket.io";*

*import http from "http";*

*import express from "express";*

*const app=express();*

*const server = http.createServer(app);*

*const io = new Server(server,{*

*cors: {*

*origin: ["http://localhost:5173", "http://0.0.0.0:5173","http://192.168.1.65:5173"] ,*

*}*

*})*

*export function getReceiverSocketId(userId){*

*return userSocketMap[userId]*

*}*

*//for storing online users in userid:socketId format*

*const userSocketMap={}*

*io.on("connection", (socket) => {*

*const userId = socket.handshake.query.userId;*

*if(userId){*

*userSocketMap[userId]=socket.id*

*}*

*//broadcasting the users with keys(userid as we said we keep format userid: socket id previously getting only socket id)*

*io.emit("getOnlineUsers", Object.keys(userSocketMap))*

*console.log("user connected",socket.id);*

*socket.on("disconnect", () => {*

*console.log("user disconnected",socket.id);*

*delete(userSocketMap[userId])*

*io.emit("getOnlineUsers", Object.keys(userSocketMap))*

*})*

*})*

*export {io,server,app}*

### index,js modification (take off app express replace it with import from socket.js as follows)

*import express from "express";*

*import dotenv from "dotenv";*

*import cookieParser from "cookie-parser";*

*dotenv.config();*

*import authRoutes from "../routes/auth.route.js";*

*import { connectDB } from "../lib/db.js";*

*import Message from "../models/message.model.js";*

*import messageRoutes from "../routes/message.route.js";*

*import cors from "cors";*

*import { app,server } from "../lib/socket.js";*

*const PORT = process.env.PORT || 3000; // Default to port 3000 if PORT is not defined*

*app.use(express.json());*

*app.use(cookieParser());*

*// app.use(express.json({ limit: "1mb" }));*

*app.use(express.json({ limit: "50mb" })); // Increase the limit*

*app.use(express.urlencoded({ limit: "50mb", extended: true }));*

*app.use(cors({*

*origin: ["http://localhost:5173", "http://0.0.0.0:5173","http://192.168.1.65:5173"] ,*

*credentials: true*

*}));*

*app.use("/api/auth", authRoutes);*

*app.use("/api/message", messageRoutes);*

*server.listen(PORT , "0.0.0.0", () => {*

*console.log(`Running on https://localhost:${PORT}`);*

*connectDB()*

*});*

### useAuthStore.js (for state management of socket)

*import { create } from "zustand";*

*import { axiosInstance } from "../lib/axios";*

*import { toast } from "react-hot-toast";*

*import { io } from "socket.io-client";*

*const BASE\_URL = "http://localhost:5001";*

*export const useAuthStore = create((set, get) => ({*

*authUser: null,*

*isSigningUp: false,*

*isLoggingin: false,*

*isUpdatingProfile: false,*

*isCheckingAuth: true,*

*onlineUsers:[],*

*socket: null,*

*// Check authentication*

*checkAuth: async () => {*

*try {*

*const response = await axiosInstance.get("/auth/check");*

*set({ authUser: response.data });*

*await get().connectSocket(); // Connect socket after auth success*

*} catch (error) {*

*set({ authUser: null });*

*console.error("Error in checkAuth:", error);*

*} finally {*

*set({ isCheckingAuth: false });*

*}*

*},*

*// Signup function*

*signup: async (data) => {*

*set({ isSigningUp: true });*

*try {*

*const response = await axiosInstance.post("/auth/signup", data);*

*toast.success("Account created successfully!");*

*set({ authUser: response.data });*

*} catch (error) {*

*const errorMessage = error.response?.data?.msg || "Signup failed!";*

*toast.error(errorMessage);*

*} finally {*

*set({ isSigningUp: false });*

*}*

*},*

*// Logout function*

*logout: async () => {*

*try {*

*console.log("Logout clicked");*

*await axiosInstance.get("/auth/logout");*

*set({ authUser: null });*

*toast.success("Logout successful");*

*await get().disconnectSocket(); // Disconnect socket after logout*

*} catch (error) {*

*console.error("Error in logout:", error);*

*toast.error("An error occurred while logging out.");*

*}*

*},*

*// Login function*

*login: async (data) => {*

*set({ isLoggingin: true });*

*try {*

*const response = await axiosInstance.post("/auth/signin", data);*

*toast.success("Login successful");*

*set({ authUser: response.data });*

*await get().connectSocket(); // Connect socket after login*

*} catch (error) {*

*const errorMessage = error.response?.data?.msg || "Login failed!";*

*toast.error(errorMessage);*

*} finally {*

*set({ isLoggingin: false });*

*}*

*},*

*// Update profile function*

*updateProfile: async (data) => {*

*set({ isUpdatingProfile: true });*

*try {*

*const response = await axiosInstance.put("/auth/update-profile", data);*

*toast.success("Profile updated successfully!");*

*set({ authUser: response.data });*

*} catch (error) {*

*const errorMessage = error.response?.data?.msg || "Profile update failed!";*

*toast.error(errorMessage);*

*} finally {*

*set({ isUpdatingProfile: false });*

*}*

*},*

*// Function to connect to a socket server*

*connectSocket: async () => {*

*const { authUser } = get(); // Retrieve the currently authenticated user from the state*

*const existingSocket = get().socket; // Retrieve the current socket instance from the state*

*// Check if the user is authenticated and if a socket connection already exists*

*if (!authUser || (existingSocket && existingSocket.connected)) {*

*console.log("Socket is already connected or user is not authenticated");*

*return; // Exit the function if conditions are met*

*}*

*// Initialize a new socket connection with user details*

*const socket = io(BASE\_URL, { // `BASE\_URL` is the server's URL*

*query:{*

*userId:authUser.\_id // Pass the authenticated user's ID to the server*

*}*

*});*

*socket.connect(); // Establish the socket connection*

*// Save the newly created socket instance in the state*

*set({ socket: socket });*

*// Set up event listener for successful connection*

*socket.on("connect", () => {*

*console.log("Socket connected", socket.id); // Log the socket ID upon successful connection*

*});*

*// Handle the reception of online user IDs from the server*

*socket.on("getOnlineUsers", (usersIds) => {*

*set({ onlineUsers: usersIds }); // Update the list of online users in the state*

*});*

*// Set up event listener for connection errors*

*socket.on("connect\_error", (err) => {*

*console.error("Socket connection error:", err); // Log the error details*

*});*

*},*

*// Function to disconnect from the socket server*

*disconnectSocket: async () => {*

*const existingSocket = get().socket; // Retrieve the current socket instance from the state*

*// Check if there's an active socket connection to disconnect*

*if (existingSocket) {*

*existingSocket.disconnect(); // Disconnect the socket from the server*

*set({ socket: null }); // Clear the socket instance from the state*

*console.log("Socket disconnected");*

*} else {*

*console.log("No socket to disconnect"); // Inform if no socket connection exists*

*}*

*},*

*}));*

### Message.controller.js

*// Import required utilities and models*

*import { getReceiverSocketId, io } from "../lib/socket.js"; // For socket-based communication*

*import Message from "../models/message.model.js"; // Message schema/model*

*import User from "../models/user.model.js"; // User schema/model (ensure .js extension is correct)*

*// Controller function to get users for the sidebar*

*export const getUsersForSidebar = async (req, res) => {*

*try {*

*// Retrieve the logged-in user's ID*

*const loggedInUser = req.user.\_id;*

*// Find all users except the logged-in user, excluding their passwords*

*const filteredUsers = await User.find({ \_id: { $ne: loggedInUser } }).select(*

*"-password" // Exclude the password field for security*

*);*

*// Send the filtered list of users as a response*

*res.status(200).json(filteredUsers);*

*} catch (error) {*

*// Log the error and return a 500 status with an error message*

*console.error(error);*

*res.status(500).json({ message: "Error fetching users" });*

*}*

*};*

*// Controller function to get messages between two users*

*export const getMessages = async (req, res) => {*

*try {*

*// Extract the ID of the user to chat with from the URL parameters*

*const { id: userToChatId } = req.params;*

*// Get the logged-in user's ID*

*const myId = req.user.\_id;*

*// Fetch messages where the logged-in user is either the sender or receiver*

*const messages = await Message.find({*

*$or: [ // Either condition can be true*

*{ senderId: myId, receiverId: userToChatId }, // Sent by me to the other user*

*{ senderId: userToChatId, receiverId: myId }  // Sent by the other user to me*

*]*

*});*

*// Send the messages as a response*

*res.status(200).json(messages);*

*} catch (error) {*

*// Log the error and return a 500 status with an error message*

*console.log("Error in getMessages controller: " + error.message);*

*res.status(500).json({ message: "Error fetching messages" });*

*}*

*};*

*// Controller function to send a message*

*export const sendMessage = async (req, res) => {*

*try {*

*// Destructure the text and image from the request body*

*const { text, image } = req.body;*

*// Extract the receiver's ID from the URL parameters*

*const { id: receiverId } = req.params;*

*// Get the sender's ID from the logged-in user's details*

*const senderId = req.user.\_id;*

*// Debugging logs to verify sender and receiver IDs*

*console.log("this is here");*

*console.log(receiverId);*

*console.log(senderId);*

*// Create a new message instance with sender, receiver, text, and optional image*

*const newMessage = new Message({*

*senderId, // ID of the sender*

*receiverId, // ID of the receiver*

*text, // Text content of the message*

*image // Optional image associated with the message*

*});*

*// Save the new message to the database*

*await newMessage.save();*

*// Get the receiver's socket ID for real-time messaging*

*const receiverSocketId = getReceiverSocketId(receiverId);*

*// If the receiver is connected via socket, emit the new message*

*if (receiverSocketId) {*

*io.to(receiverSocketId).emit("newMessage", newMessage); // Real-time message delivery*

*}*

*// Respond with the created message*

*res.status(201).json(newMessage);*

*} catch (error) {*

*// Log the error and return a 500 status with an error message*

*console.log("Error in sendMessage controller: ", error.message);*

*res.status(500).json({ error: "Internal server error" });*

*}*

*};*

### **Message.model.js**

*import mongoose from "mongoose";*

*const messageSchema = new mongoose.Schema({*

*senderId: { type: mongoose.Schema.Types.ObjectId,ref: "User", required: true },*

*receiverId: { type: mongoose.Schema.Types.ObjectId,ref:"User", required: true },*

*text: { type: String},*

*image: { type: String}*

*}, {*

*timestamps: true*

*});*

*const Message = mongoose.model("Message", messageSchema);*

*export default Message;*

### **Index.js**

*import express from "express";*

*import dotenv from "dotenv";*

*import cookieParser from "cookie-parser";*

*dotenv.config();*

*import authRoutes from "../routes/auth.route.js";*

*import { connectDB } from "../lib/db.js";*

*import Message from "../models/message.model.js";*

*import messageRoutes from "../routes/message.route.js";*

*import cors from "cors";*

*import { app,server } from "../lib/socket.js";*

*const PORT = process.env.PORT || 3000; // Default to port 3000 if PORT is not defined*

*app.use(express.json());*

*app.use(cookieParser());*

*// app.use(express.json({ limit: "1mb" }));*

*app.use(express.json({ limit: "50mb" })); // Increase the limit*

*app.use(express.urlencoded({ limit: "50mb", extended: true }));*

*app.use(cors({*

*origin: ["http://localhost:5173", "http://0.0.0.0:5173","http://192.168.1.65:5173"] ,*

*credentials: true*

*}));*

*app.use("/api/auth", authRoutes);*

*app.use("/api/message", messageRoutes);*

*server.listen(PORT , "0.0.0.0", () => {*

*console.log(`Running on https://localhost:${PORT}`);*

*connectDB()*

*});*

### **UseChatstore.js**

*// Import necessary modules*

*import { create } from "zustand"; // Zustand library for state management*

*import toast from "react-hot-toast"; // Library for displaying toast notifications*

*import { axiosInstance } from "../lib/axios"; // Custom Axios instance for API requests*

*import { useAuthStore } from "./useAuthStore"; // Authentication store for managing user authentication state*

*// Zustand store for chat-related state management*

*export const useChatStore = create((set, get) => ({*

*// Initial state variables*

*messages: [], // Array to store messages*

*users: [], // Array to store user list*

*selectedUser: null, // Currently selected user for chat*

*isUsersLoading: false, // Loading state for fetching users*

*isMessagesLoading: false, // Loading state for fetching messages*

*// Function to fetch users*

*getUsers: async () => {*

*set({ isUsersLoading: true }); // Set loading state to true*

*try {*

*const res = await axiosInstance.get("/message/users"); // Fetch users from API*

*set({ users: res.data }); // Update state with fetched users*

*} catch (error) {*

*// Show an error notification if fetching fails*

*toast.error(error.response?.data?.message || "Failed to load users.");*

*} finally {*

*set({ isUsersLoading: false }); // Set loading state to false after completion*

*}*

*},*

*// Function to fetch messages for the selected user*

*getMessages: async (userId) => {*

*if (!userId) {*

*toast.error("User ID is missing."); // Show error if user ID is not provided*

*return; // Exit function*

*}*

*set({ isMessagesLoading: true }); // Set loading state to true*

*try {*

*const res = await axiosInstance.get(`/message/${userId}`); // Fetch messages from API*

*set({ messages: res.data }); // Update state with fetched messages*

*} catch (error) {*

*// Show an error notification if fetching fails*

*toast.error(error.response?.data?.message || "Failed to load messages.");*

*} finally {*

*set({ isMessagesLoading: false }); // Set loading state to false after completion*

*}*

*},*

*// Function to send a message*

*sendMessage: async (messageData) => {*

*const { selectedUser } = get(); // Retrieve the currently selected user from state*

*const { authUser } = useAuthStore.getState(); // Retrieve the authenticated user from auth store*

*console.log("Selected User:", selectedUser); // Debugging log for selected user*

*console.log("Logged-in User:", authUser); // Debugging log for authenticated user*

*if (!selectedUser) {*

*toast.error("No user selected."); // Notify if no user is selected*

*console.error("Error: No user selected."); // Debugging log for error*

*return; // Exit function*

*}*

*if (!authUser?.\_id) {*

*toast.error("Sender ID (your user ID) is missing."); // Notify if sender ID is missing*

*console.error("Error: Sender ID (user.\_id) is missing."); // Debugging log for error*

*return; // Exit function*

*}*

*try {*

*console.log("Message Data:", messageData); // Debugging log for message data*

*console.log("Sending message to:", `/message/send/${selectedUser.\_id}`); // Debugging log for API endpoint*

*const res = await axiosInstance.post(`/message/send/${selectedUser.\_id}`, {*

*...messageData, // Include message details*

*senderId: authUser.\_id, // Explicitly set sender ID*

*});*

*console.log("Message sent successfully. Response:", res.data); // Debugging log for success response*

*// Update state with the newly sent message*

*set({ messages: [...get().messages, res.data] });*

*} catch (error) {*

*console.error("Failed to send message:", error); // Debugging log for error*

*toast.error(error.response?.data?.message || "Failed to send message."); // Notify about the error*

*}*

*},*

*// Function to subscribe to real-time message updates*

*subscribeToMessages: () => {*

*const { selectedUser } = get(); // Retrieve the selected user from state*

*const { user } = useAuthStore.getState(); // Retrieve the authenticated user*

*if (!selectedUser || !user?.\_id) return; // Exit if conditions are not met*

*const socket = useAuthStore.getState().socket; // Retrieve the socket instance from auth store*

*// Listen for new messages from the server*

*socket.on("newMessage", (newMessage) => {*

*// Check if the message belongs to the selected user or the logged-in user*

*if (newMessage.senderId === selectedUser.\_id || newMessage.senderId === user.\_id) {*

*set({ messages: [...get().messages, newMessage] }); // Update state with the new message*

*}*

*});*

*},*

*// Function to unsubscribe from real-time message updates*

*unsubscribeFromMessages: () => {*

*const socket = useAuthStore.getState().socket; // Retrieve the socket instance from auth store*

*socket.off("newMessage"); // Remove the event listener for new messages*

*},*

*// Function to set the currently selected user*

*setSelectedUser: (selectedUser) => set({ selectedUser }),*

*}));*

### **Homepage.jsx**

*// Import the chat store and necessary components*

*import { useChatStore } from "../store/useChatStore"; // Zustand store to manage chat-related state*

*import Sidebar from "../components/Sidebar"; // Sidebar component for displaying user list or navigation*

*import NoChatSelected from "../components/NoChatSelected"; // Component shown when no chat is selected*

*import ChatContainer from "../components/ChatContainer"; // Component that renders the chat interface*

*// Functional component for the Home Page*

*const HomePage = () => {*

*const { selectedUser } = useChatStore(); // Retrieve the currently selected user from the chat store*

*return (*

*// Full-screen container with a base background*

*<div className="h-screen bg-base-200">*

*{/\* Center the main content with padding at the top \*/}*

*<div className="flex items-center justify-center pt-20 px-4">*

*{/\* Main container with a shadow and rounded corners \*/}*

*<div className="bg-base-100 rounded-lg shadow-cl w-full max-w-6xl h-[calc(100vh-8rem)]">*

*{/\* Flex container for the layout \*/}*

*<div className="flex h-full rounded-lg overflow-hidden">*

*<Sidebar /> {/\* Sidebar for user list or navigation \*/}*

*{/\* Show either NoChatSelected or ChatContainer based on whether a user is selected \*/}*

*{!selectedUser ? <NoChatSelected /> : <ChatContainer />}*

*</div>*

*</div>*

*</div>*

*</div>*

*);*

*};*

*// Export the HomePage component as the default export*

*export default HomePage;*

### **ChatContainer.jsx**

*// Import necessary hooks, stores, components, and utilities*

*import { useChatStore } from "../store/useChatStore"; // Zustand store for managing chat state*

*import { useEffect, useRef } from "react"; // React hooks for side effects and element references*

*import ChatHeader from "./ChatHeader"; // Component for the chat header*

*import MessageInput from "./MessageInput"; // Component for sending messages*

*import MessageSkeleton from "./skeletons/MessageSkeleton"; // Placeholder skeleton for loading messages*

*import { useAuthStore } from "../store/useAuthStore"; // Store for authentication state*

*import { formatMessageTime } from "../lib/utils"; // Utility function to format message timestamps*

*// ChatContainer component handles the chat interface*

*const ChatContainer = () => {*

*// Extract necessary state and actions from stores*

*const {*

*messages, // List of chat messages*

*getMessages, // Function to fetch messages for the selected user*

*isMessagesLoading, // State to check if messages are being loaded*

*selectedUser, // The currently selected user for chat*

*subscribeToMessages, // Function to start listening to real-time messages*

*unsubscribeFromMessages, // Function to stop listening to real-time messages*

*} = useChatStore();*

*const { authUser } = useAuthStore(); // Retrieve authenticated user info*

*const messageEndRef = useRef(null); // Reference to the last message for auto-scrolling*

*// Fetch messages and set up real-time subscriptions when a user is selected*

*useEffect(() => {*

*getMessages(selectedUser.\_id); // Fetch messages for the selected user*

*subscribeToMessages(); // Subscribe to real-time message updates*

*// Cleanup: Unsubscribe from messages when the component unmounts*

*return () => unsubscribeFromMessages();*

*}, [selectedUser.\_id, getMessages, subscribeToMessages, unsubscribeFromMessages]);*

*// Auto-scroll to the latest message whenever messages are updated*

*useEffect(() => {*

*if (messageEndRef.current && messages) {*

*messageEndRef.current.scrollIntoView({ behavior: "smooth" });*

*}*

*}, [messages]);*

*// Render loading skeleton while messages are being fetched*

*if (isMessagesLoading) {*

*return (*

*<div className="flex-1 flex flex-col overflow-auto">*

*<ChatHeader /> {/\* Display the chat header \*/}*

*<MessageSkeleton /> {/\* Display loading skeleton for messages \*/}*

*<MessageInput /> {/\* Keep the input visible \*/}*

*</div>*

*);*

*}*

*return (*

*<div className="flex-1 flex flex-col overflow-auto">*

*<ChatHeader /> {/\* Display the chat header \*/}*

*{/\* Messages display area \*/}*

*<div className="flex-1 overflow-y-auto p-4 space-y-4">*

*{/\* Map over messages to display each one \*/}*

*{messages.map((message) => (*

*<div*

*key={message.\_id} // Unique key for each message*

*className={`chat ${message.senderId === authUser.\_id ? "chat-end" : "chat-start"}`} // Align messages based on sender*

*ref={messageEndRef} // Reference for auto-scrolling*

*>*

*{/\* User avatar \*/}*

*<div className="chat-image avatar">*

*<div className="size-10 rounded-full border">*

*<img*

*src={*

*message.senderId === authUser.\_id*

*? authUser.profilePic || "/avatar.png" // Use authenticated user's profile picture*

*: selectedUser.profilePic || "/avatar.png" // Use selected user's profile picture*

*}*

*alt="profile pic" // Alt text for image*

*/>*

*</div>*

*</div>*

*{/\* Chat header with timestamp \*/}*

*<div className="chat-header mb-1">*

*<time className="text-xs opacity-50 ml-1">*

*{formatMessageTime(message.createdAt)} {/\* Format the message's timestamp \*/}*

*</time>*

*</div>*

*{/\* Chat bubble containing text and optional image \*/}*

*<div className="chat-bubble flex flex-col">*

*{message.image && (*

*<img*

*src={message.image} // Display image if present*

*alt="Attachment"*

*className="sm:max-w-[200px] rounded-md mb-2"*

*/>*

*)}*

*{message.text && <p>{message.text}</p>} {/\* Display text if present \*/}*

*</div>*

*</div>*

*))}*

*</div>*

*{/\* Message input field \*/}*

*<MessageInput />*

*</div>*

*);*

*};*

*export default ChatContainer; // Export the ChatContainer component*

### Sidebar.jsx

*// Importing necessary React hooks and components*

*import { useEffect, useState } from "react";*

*// Importing functions and states from custom stores for managing chat and authentication*

*import { useChatStore } from "../store/useChatStore";*

*import { useAuthStore } from "../store/useAuthStore";*

*// Importing a skeleton component for loading state*

*import SidebarSkeleton from "./skeletons/SidebarSkeleton";*

*// Importing an icon component for rendering user icons*

*import { Users } from "lucide-react";*

*// Defining the Sidebar functional component*

*const Sidebar = () => {*

*// Destructuring required state and functions from `useChatStore`*

*const { getUsers, users, selectedUser, setSelectedUser, isUsersLoading } = useChatStore();*

*// Destructuring online user data from `useAuthStore`*

*const { onlineUsers } = useAuthStore();*

*// Local state to toggle "show online users only"*

*const [showOnlineOnly, setShowOnlineOnly] = useState(false);*

*// useEffect hook to call `getUsers` function when the component mounts or `getUsers` changes*

*useEffect(() => {*

*getUsers();*

*}, [getUsers]);*

*// Filtered users based on whether the "show online only" toggle is active*

*const filteredUsers = showOnlineOnly*

*? users.filter((user) => onlineUsers.includes(user.\_id)) // Only show users who are online*

*: users; // Show all users when toggle is inactive*

*// If users are still loading, render the `SidebarSkeleton` loading component*

*if (isUsersLoading) return <SidebarSkeleton />;*

*// Main component rendering starts here*

*return (*

*<aside className="h-full w-20 lg:w-72 border-r border-base-300 flex flex-col transition-all duration-200">*

*{/\* Header section with title and optional toggle \*/}*

*<div className="border-b border-base-300 w-full p-5">*

*{/\* Sidebar title \*/}*

*<div className="flex items-center gap-2">*

*<Users className="size-6" /> {/\* Icon for Contacts \*/}*

*<span className="font-medium hidden lg:block">Contacts</span> {/\* Title visible on large screens \*/}*

*</div>*

*{/\* TODO: Online filter toggle \*/}*

*<div className="mt-3 hidden lg:flex items-center gap-2">*

*{/\* Checkbox for toggling "show online only" \*/}*

*<label className="cursor-pointer flex items-center gap-2">*

*<input*

*type="checkbox"*

*checked={showOnlineOnly} // Checkbox state bound to `showOnlineOnly`*

*onChange={(e) => setShowOnlineOnly(e.target.checked)} // Update state when checkbox toggles*

*className="checkbox checkbox-sm"*

*/>*

*<span className="text-sm">Show online only</span> {/\* Label for checkbox \*/}*

*</label>*

*{/\* Placeholder to show the count of online users \*/}*

*{/\* <span className="text-xs text-zinc-500">({onlineUsers.length - 1} online)</span> \*/}*

*</div>*

*</div>*

*{/\* User list rendering \*/}*

*<div className="overflow-y-auto w-full py-3">*

*{filteredUsers.map((user) => (*

*<button*

*key={user.\_id} // Unique identifier for each user*

*onClick={() => setSelectedUser(user)} // Set the clicked user as selected*

*className={`*

*w-full p-3 flex items-center gap-3*

*hover:bg-base-300 transition-colors*

*${selectedUser?.\_id === user.\_id ? "bg-base-300 ring-1 ring-base-300" : ""}*

*`}*

*>*

*{/\* User profile picture and online status \*/}*

*<div className="relative mx-auto lg:mx-0">*

*<img*

*src={user.profilePic || "/avatar.png"} // Default avatar if profile picture is unavailable*

*alt={user.name} // Accessible alt text for user image*

*className="size-12 object-cover rounded-full"*

*/>*

*{onlineUsers.includes(user.\_id) && (*

*<span*

*className="absolute bottom-0 right-0 size-3 bg-green-500*

*rounded-full ring-2 ring-zinc-900"*

*/>*

*)}*

*</div>*

*{/\* User information, visible on larger screens \*/}*

*<div className="hidden lg:block text-left min-w-0">*

*<div className="font-medium truncate">{user.fullName}</div> {/\* Full name \*/}*

*<div className="text-sm text-zinc-400">*

*{onlineUsers.includes(user.\_id) ? "Online" : "Offline"} {/\* Online/offline status \*/}*

*</div>*

*</div>*

*</button>*

*))}*

*{/\* Placeholder for no users available \*/}*

*{/\* {filteredUsers.length === 0 && (*

*<div className="text-center text-zinc-500 py-4">No online users</div>*

*)} \*/}*

*</div>*

*</aside>*

*);*

*};*

*// Exporting the Sidebar component as default for usage elsewhere*

*export default Sidebar;*