MAKING CHAT app USING MONGODB, EXPRESS, JS, NODEjs , HTML,CSS,EJS , API INTEGRATION, POSTMAN, OAuth .   VIBECONNECT PROJECT

Directory by debabrata behera-

```
vibeconnect/
│
├── auth/
│  ├── route.js
│  ├── auth.js
│  └── token.env
│
├── db/
│  └── index.js
│
├── middleware/
│  └── auth.js
│
├── model/
│  ├── user.js
│  └── chat.js
│
├── socket/
│  └── index.js
│
├── views/
│  ├── admin.ejs
│  ├── chat.ejs
│  ├── login.ejs
│  ├── register.ejs
│  ├── user.ejs
│  └── home.ejs
│
├── server.js
└── package.json
```

Initialize a New Node.js Project
Run npm init to create a package.json file. This file will manage your project's dependencies and scripts.

```
npm init -y
```

then install this
npm install express ejs mongoose dotenv cookie-parser jsonwebtoken bcrypt bcryptjs multer nodemon passport passport-facebook passport-google-oauth20 socket.io auth0

Install socket.io - npm install socket.io

express: Web framework for Node.js.
ejs: Templating engine.
mongoose: MongoDB ODM for Node.js.
dotenv: Environment variable management.
cookie-parser: Middleware for cookie parsing.
jsonwebtoken: JSON Web Token implementation.
bcrypt and bcryptjs: Password hashing.
multer: Middleware for handling file uploads.
nodemon: Development tool that automatically restarts the server.
passport, passport-facebook, passport-google-oauth20: Authentication middleware.

1- create a folder named Auth
add files i-auth.js
ii-route.js

in auth.js file contains authentication and user management logic, which is crucial for a chat application. Here's how each part contributes to the functionality of a chat app.

User Authentication and Authorization:

adminAuth Middleware: Ensures that only users with the "admin" role can access certain routes (e.g., update or delete users). This is important for maintaining security and proper role management within the app.
userAuth Middleware: Ensures that users with a "Basic" role can access specific routes. This is used to verify that users are authenticated and authorized to perform certain actions.
User Registration and Login:

register Function: Handles user registration by creating a new user with hashed passwords. It generates a JWT (JSON Web Token) for session management, allowing users to log in and stay authenticated across requests.
login Function: Handles user login by verifying credentials. If successful, it generates a JWT and sets it as an HTTP-only cookie, ensuring that the user's session is secure.
User Management:

update Function: Allows an admin to change the role of a user, such as promoting a user to "admin". This is important for managing user roles and permissions dynamically.
deleteUser Function: Allows for the deletion of a user. This function also clears the JWT cookie for the user being deleted, which helps in invalidating their session.
User Retrieval:

getUsers Function: Retrieves a list of all users from the database, excluding the currently logged-in user. It also checks if users are online by consulting UserStore. This information can be used to display user lists with online status in the chat interface.

code with explanation

```
const bcrypt = require('bcrypt'); // Import bcrypt for hashing passwords
const jwt = require('jsonwebtoken'); // Import jsonwebtoken for creating and verifying JWT tokens

const User = require("../model/user"); // Import the User model for database operations
const { UserStore } = require("../socket"); // Import UserStore from socket for online user tracking

const maxAge = 3 * 60 * 60; // Token expiration time in seconds (3 hours)
const jwtSecret =
'4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd'; // Secret key for JWT

// Middleware to check if a request is made by an admin
```

```
exports.adminAuth = (req, res, next) => {
const token = req.cookies.jwt; // Retrieve JWT from cookies
if (token) {
jwt.verify(token, jwtSecret, (err, decodedToken) => { // Verify JWT token
if (err) {
return res.status(401).json({ message: "Not authorized" }); // Token verification failed
} else {
if (decodedToken.role !== "admin") { // Check if user role is not admin
return res.status(401).json({ message: "Not authorized" }); // User is not an admin
} else {
next(); // Proceed to the next middleware or route handler
}
}
});
} else {
return res.status(401).json({ message: "Not authorized, token not available" }); // No token found
}
};

// Controller for user registration
exports.register = async (req, res, next) => {
console.log(req.body); // Log request body for debugging
const { username, password } = req.body; // Extract username and password from request body
if (password.length < 6) {
return res.status(400).json({ message: "Password less than 6 characters" }); // Password is too short
}
try {
console.log(username, 'string', password); // Log username and password for debugging
bcrypt.hash(password, 10).then(async (hash) => { // Hash the password with a salt rounds of 10
await User.create({
username,
password: hash, // Save hashed password to the database
})
.then((user) => {
const token = jwt.sign(
{ id: user._id, username: user.username, role: user.role },
jwtSecret,
{
expiresIn: maxAge, // Set token expiration time
}
);
res.cookie("jwt", token, {
httpOnly: true,
maxAge: maxAge * 1000, // Set cookie expiration time in milliseconds
});
res.status(201).json({
message: "User successfully created",
user: user._id,
});
})
.catch((error) =>
res.status(400).json({
message: "User not successfully created",
error: error.message,
})
);
});
} catch (err) {
```

```
res.status(401).json({
message: "User not successfully created",
err: err.message,
});
}
};

// Controller for user login
exports.login = async (req, res, next) => {
const { username, password } = req.body; // Extract username and password from request body
// Check if username and password are provided
console.log('username::', username); // Log username for debugging
if (!username || !password) {
return res.status(400).json({
message: "Username or Password not present",
});
}
try {
const user = await User.findOne({ username }); // Find user by username
if (!user) {
res.status(400).json({
message: "Login not successful",
error: "User not found",
});
} else {
// Compare provided password with hashed password in the database
bcrypt.compare(password, user.password).then(function (result) {
if(result) {
const token = jwt.sign(
{ id: user._id, username: user.username, role: user.role },
jwtSecret,
{
expiresIn: maxAge, // Set token expiration time
}
);
res.cookie("jwt", token, {
httpOnly: true,
maxAge: maxAge * 1000, // Set cookie expiration time in milliseconds
});
res.status(200).json({
message: "Login successful",
user,
});
} else {
res.status(400).json({ message: "Login not successful" }); // Password mismatch
}
});
}
} catch (error) {
res.status(400).json({
message: "An error occurred",
error: error.message,
});
}
};

// Controller for updating user roles
exports.update = async (req, res, next) => {
```

```
const { role, id } = req.body; // Extract role and id from request body
// Check if role and id are provided
if (role && id) {
// Check if the provided role is 'admin'
if (role === "admin") {
await User.findById(id) // Find user by ID
.then((user) => {
// Check if the user is not already an admin
if (user.role !== "admin") {
user.role = role; // Update user role
user.save() // Save the updated user role
.then((savedUser) => {
res.status(201).json({ message: "Update successful", user: savedUser });
})
.catch((err) => {
res.status(400).json({ message: "An error occurred", error: err.message });
});

} else {
res.status(400).json({ message: "User is already an Admin" }); // User is already an admin
}
})
.catch((error) => {
res.status(400).json({ message: "An error occurred", error: error.message });
});
} else {
res.status(400).json({
message: "Role is not admin",
});
}
} else {
res.status(400).json({ message: "Role or Id not present" }); // Role or ID missing
}
};

// Controller for deleting a user
exports.deleteUser = async (req, res, next) => {
const { id } = req.body; // Extract user ID from request body
await User.findById(id) // Find user by ID
.then(user => User.deleteOne({ _id: id })) // Delete user from database
.then(user => {
res.cookie("jwt", "", { maxAge: "1" }); // Clear JWT cookie
res.status(201).json({ message: "User successfully deleted", user });
})
.catch(error =>
res.status(400).json({ message: "An error occurred", error: error.message })
);
};

// Middleware to check if a request is made by a basic user
exports.userAuth = (req, res, next) => {
const token = req.cookies.jwt; // Retrieve JWT from cookies
if (token) {
jwt.verify(token, jwtSecret, (err, decodedToken) => { // Verify JWT token
if (err) {
return res.status(401).json({ message: "Not authorized" }); // Token verification failed
} else {
if (decodedToken.role !== "Basic") { // Check if user role is not basic
return res.status(401).json({ message: "Not authorized" }); // User is not a basic user
} else {
next(); // Proceed to the next middleware or route handler
}
}
});
} else {
return res.status(401).json({ message: "Not authorized, token not available" }); // No token found
}
};

// Controller for retrieving all users
exports.getUsers = async (req, res) => {
const token = req.cookies.jwt; // Retrieve JWT from cookies
let id;

jwt.verify(token, jwtSecret, (err, decodedToken) => { // Verify JWT token
if (err) {
return res.status(401).json({ message: "Not authorized" }); // Token verification failed
} else {
id = decodedToken.id; // Extract user ID from decoded token
}
});
await User.find({}) // Find all users
.then(users => {
const userFunction = users.reduce((allUsers, user) => { // Reduce users to an array with necessary info
if(user.id == id) return allUsers; // Exclude the current user
const container = {};
container.isOnline = false; // Default status for online
container.username = user.username;
container.role = user.role;
container.id = user.id;
container.isOnline = !!UserStore.getOnlineUser(user.id); // Check if user is online
allUsers.push(container);
return allUsers;
}, []);
res.status(200).json({ user: userFunction }); // Respond with user data
})
.catch(err =>
res.status(401).json({ message: "Not successful", error: err.message })
);
};
```

Purpose of route.js in the Chat App:

The route.js file serves as the central routing configuration for your Express application. It defines the routes and associates them with specific handler functions from the auth module. Here's a breakdown of its purpose:

Routing: It sets up routes for user-related operations such as registration, login, and user management. This routing helps in managing different endpoints of the API, making it easier to handle different HTTP requests.

Authorization Middleware: For routes that require administrative privileges (e.g., updating or deleting a user), it applies the adminAuth middleware to ensure that only users with admin roles can perform these actions. This adds a layer of security to the API.

Centralized Route Management: By using this file, you maintain a clean and organized structure for routing, which improves code readability and maintainability.

code with explanation

```
// Import the express module to create a router
const express = require("express");

// Import functions for handling various routes from the auth module
const { register, login, update, deleteUser, getUsers } = require("./auth");
// Import the adminAuth middleware for route protection
const { adminAuth } = require("../middleware/auth")

// Create an instance of the express Router
const router = express.Router();

// Define a POST route for user registration
// The register function will handle requests to this route
router.route("/register").post(register);

// Define a POST route for user login
// The login function will handle requests to this route
router.route("/login").post(login);

// Define a GET route to fetch all users
// The getUsers function will handle requests to this route
router.route("/getUsers").get(getUsers)

// Define a PUT route for updating user details
// The update function will handle requests to this route
// The adminAuth middleware is applied to ensure only admins can access this route
router.route("/update").put(adminAuth, update)

// Define a DELETE route for deleting a user
// The deleteUser function will handle requests to this route
// The adminAuth middleware is applied to ensure only admins can access this route
router.route("/deleteUser").delete(adminAuth, deleteUser)

// Export the router to be used in other parts of the application
module.exports = router;
```

token.env d97489e1957c134abc050fb93a17f739ba1c8f33340a61d7dfce9881ef43bfb3c3b012

2nd folder db (database/mongodb)

Purpose of db.js
The db.js file serves as a utility module for interacting with the MongoDB database in the chat application. Here's a summary of its roles:

Database Connection: Provides functionality to connect to a local MongoDB instance, ensuring that the application can interact with the database.

User Management:
Retrieves user information based on user ID.
This is useful for getting details about users, such as their username and role, which may be needed for various application features.

Message Management:
Allows for the creation of chat messages between users.
Retrieves chat history between two users, handling messages sent in both directions and sorting them by date.

code with explanation

```
// Import the necessary modules and models
const User = require("../model/user");
const Chat = require("../model/chat");
const Mongoose = require("mongoose");

// Define the local MongoDB database connection string
const localDB = `mongodb://localhost:27017/role_auth`;

// Function to connect to the MongoDB database
exports.connectDB = async () => {
await Mongoose.connect(localDB, {
useNewUrlParser: true, // Use the new URL parser to avoid deprecation warnings
useUnifiedTopology: true, // Use the new server discovery and monitoring engine
});
console.log("MongoDB Connected"); // Log a message indicating successful connection
};

// Function to retrieve a user by their ID
exports.getUser = async (id) => {
const user = await User.findById(id); // Find the user by ID in the User collection
const container = {}; // Create an empty object to hold user information
container.username = user.username; // Add the username to the container
container.role = user.role; // Add the role to the container
container.id = user.id; // Add the user ID to the container
return container; // Return the container with user information
};

// Function to create a new chat message
exports.createMessage = async ({ fromId, toId, message }) => {
await Chat.create({ userFrom: fromId, userTo: toId, message }); // Create a new chat entry in the Chat
collection
console.log("creat"); // Log a message indicating the creation of the message
};

// Function to retrieve chat history between two users
exports.getChat = async ({ fromId, toId }) => {
// Create two promises to find chat messages from both directions
const promise1 = new Promise((resolve, reject) => {
try {
resolve(Chat.find({ userFrom: fromId, userTo: toId })); // Find messages from fromId to toId
} catch (e) {
reject(e); // Reject the promise if an error occurs
}
});

const promise2 = new Promise((resolve, reject) => {
try {
resolve(Chat.find({ userFrom: toId, userTo: fromId })); // Find messages from toId to fromId
} catch (e) {
reject(e); // Reject the promise if an error occurs
}
});

// Wait for both promises to resolve
const [chatList1, chatList2] = await Promise.all([promise1, promise2]);

// Check if the retrieved chat lists are arrays
if (!Array.isArray(chatList1) || !Array.isArray(chatList2)) {
console.log("ERROR: getChat not working::", chatList1, chatList2); // Log an error message if not
return []; // Return an empty array if there is an error
}

// Combine and sort the chat messages by the date they were sent
const container = [...chatList1, ...chatList2].reduce((container, { userFrom, userTo, message, sentOn }) => {
container.push({ userFrom, userTo, message, sentOn }); // Add each chat message to the container
return container; // Return the updated container
}, []);

console.log("container::", container); // Log the sorted chat messages
return container.sort(function (a, b) {
return new Date(a.sentOn) - new Date(b.sentOn); // Sort messages by the sentOn date
});
};
```

3rd folder Middleware
Purpose of auth.js
The auth.js file in the middleware directory is responsible for providing middleware functions that handle user authentication and authorization based on JSON Web Tokens (JWTs). Here’s what each part does:

JWT Authentication:

adminAuth: This middleware function checks if a user has a valid JWT and if their role is "admin". If the token is valid and the role matches, the user is granted access to the protected route. Otherwise, they receive a 401 Unauthorized status.

userAuth: This middleware function ensures that users have a valid JWT with the role "Basic" to access certain routes. If the token is missing or invalid, or if the user's role does not match, they are redirected to the login page.
Securing Routes:

These functions are used to protect routes that require specific user roles. For example, only an admin can access admin-specific routes, while regular users can access user-specific routes.
Error Handling and Redirection:

If the token is missing or invalid, or if the user’s role does not match the required role, appropriate responses are returned (either a 401 status or redirection to the login page).

code with explanation

```
// Import the jsonwebtoken module for handling JWT authentication
const jwt = require("jsonwebtoken");

// Define the secret key used for signing and verifying JWTs
const jwtSecret =
"4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd";

// Middleware function to authorize admin users
exports.adminAuth = (req, res, next) => {
```

```
// Retrieve the JWT token from cookies
const token = req.cookies.jwt;

// Check if the token exists
if (token) {
// Verify the token using the secret key
jwt.verify(token, jwtSecret, (err, decodedToken) => {
if (err) {
// If verification fails, respond with a 401 Unauthorized status
return res.status(401).json({ message: "Not authorized" });
} else {
// Check if the role of the user is "admin"
if (decodedToken.role !== "admin") {
// If the role is not admin, respond with a 401 Unauthorized status
return res.status(401).json({ message: "Not authorized" });
} else {
// If the user is an admin, proceed to the next middleware or route handler
next();
}
}
});
} else {
// If no token is present, respond with a 401 Unauthorized status
return res.status(401).json({ message: "Not authorized, token not available" });
}
};

// Middleware function to authorize basic users
exports.userAuth = (req, res, next) => {
// Retrieve the JWT token from cookies
const token = req.cookies.jwt;

// Check if the token exists
if (token) {
// Verify the token using the secret key
jwt.verify(token, jwtSecret, (err, decodedToken) => {
if (err) {
// If verification fails, respond with a 401 Unauthorized status
return res.status(401).json({ message: "Not authorized" });
} else {
// Check if the role of the user is "Basic"
if (decodedToken.role !== "Basic") {
// If the role is not Basic, respond with a 401 Unauthorized status
return res.status(401).json({ message: "Not authorized" });
} else {
// If the user is a basic user, proceed to the next middleware or route handler
next();
}
}
});
} else {
// If no token is present, redirect to the login page
return res.redirect('/login');
}
};
```

4th folder model

Purpose of chat.js
The chat.js file defines a Mongoose schema and model for storing chat messages in a MongoDB database. Here's what each part does:

Schema Definition:

userFrom: Represents the ID or username of the user who sent the message.
userTo: Represents the ID or username of the user who received the message.
message: Contains the content of the message being sent.
sentOn: A timestamp indicating when the message was sent, with a default value set to the current date and time.
Model Creation:

The ChatSchema is used to create a Mongoose model named Chat. This model will be used to interact with the chat collection in the MongoDB database, allowing for operations such as creating, querying, updating, and deleting chat messages.

Exporting the Model:

The Chat model is exported so that it can be imported and used in other parts of the application, such as controllers or services responsible for handling chat-related functionality.
Overall Purpose

The chat.js file provides the necessary schema and model for managing chat messages in the application. It allows the application to store and retrieve chat messages between users, which is essential for the chat functionality in the chat application.

code with explanation

```
// Import the Mongoose library to interact with MongoDB
const Mongoose = require("mongoose");

// Define a schema for the Chat model
const ChatSchema = new Mongoose.Schema({
// User who sent the message
userFrom: {
type: String, // Data type is a string
required: true, // This field is required
},
// User who received the message
userTo: {
type: String, // Data type is a string
required: true, // This field is required
},
// The message content
message: {
type: String, // Data type is a string
required: true, // This field is required
},
// Timestamp when the message was sent
sentOn: {
type: Date, // Data type is a Date
default: new Date(), // Default value is the current date and time
require: true // This field is required
}
});

// Create a Mongoose model based on the ChatSchema
```

```
const Chat = Mongoose.model("chat", ChatSchema);

// Export the Chat model so it can be used in other parts of the application
module.exports = Chat;
```

user.js file

Purpose of user.js
The user.js file defines a Mongoose schema and model for managing user data in a MongoDB database. Here's what each part does:

Schema Definition:

username: Represents the user's username. It must be unique and is required for every user. This ensures that no two users can have the same username.

password: Stores the user's password. It must be at least 6 characters long and is required. This field will typically store hashed passwords for security.

role: Indicates the user's role within the application (e.g., "Basic" or "Admin"). The default role is "Basic", and this field is required for each user.

Model Creation:

The UserSchema is used to create a Mongoose model named User. This model represents the user collection in the MongoDB database and provides methods to perform operations such as creating, querying, updating, and deleting user records.

Exporting the Model:

The User model is exported so that it can be imported and used in other parts of the application, such as controllers or services that need to interact with user data.
Overall Purpose

The user.js file provides the structure and functionality for managing user data in the application. By defining the schema and model for users, it enables the application to handle user-related operations, including registration, authentication, and role management. This is crucial for implementing user authentication and authorization features within the chat application.

code with explanation

```
// Import the Mongoose library to interact with MongoDB
const Mongoose = require("mongoose");

// Define a schema for the User model
const UserSchema = new Mongoose.Schema({
// Username of the user
username: {
type: String, // Data type is a string
unique: true, // Username must be unique across the collection
required: true, // This field is required
},
// Password of the user
password: {
type: String, // Data type is a string
minlength: 6, // Password must be at least 6 characters long
required: true, // This field is required
},
// Role of the user (e.g., Basic, Admin)
role: {
type: String, // Data type is a string
default: "Basic", // Default value is "Basic"
required: true, // This field is required
},
});

// Create a Mongoose model based on the UserSchema
const User = Mongoose.model("user", UserSchema);

// Export the User model so it can be used in other parts of the application
module.exports = User;
```

5th Folder SOCKET

Purpose of socket/index.js

The socket/index.js file is responsible for setting up and managing real-time communication using Socket.IO in the chat application. Here's a summary of its purpose:

User Management:

UserStore: Manages online users and their socket connections. This in-memory store keeps track of which users are currently online and their corresponding socket IDs.
Real-Time Communication:

Socket.IO Initialization: Sets up the Socket.IO server to handle real-time events such as private messaging and user disconnection.

Event Handling: Listens for and processes real-time events like sending private messages and managing user connections/disconnections.

User Authentication and Data Retrieval:

JWT Authentication: Verifies JWT tokens to authenticate users and retrieve their information. This ensures that only authorized users can connect and interact via sockets.

Chat Data Management: Handles the storage and retrieval of chat messages, enabling users to view their chat history and current online status of other users.

Integration with Application:

Rendering Chat: Provides functionality to render chat pages with the relevant chat history and user information.

Socket Middleware: Uses middleware to authenticate users based on their JWT tokens and associate user data with their sockets.

Overall, this file plays a crucial role in enabling real-time chat functionality, managing user sessions, and ensuring secure and efficient communication between users in the chat application.

code with explanation

```
// Import the Server class from the 'socket.io' package
const { Server } = require('socket.io');

// Import JSON Web Token library for authentication
```

```
const jwt = require('jsonwebtoken');

// Import utility functions from the db module
const { getUser, createMessage, getChat } = require('../db/index');

// Define the maximum age of a JWT token (3 hours)
const maxAge = 3 * 60 * 60;

// Secret key used for signing and verifying JWT tokens
const jwtSecret =
'4715aed3c946f7b0a38e6b534a9583628d84e96d10fbc04700770d572af3dce43625dd';

// Create an in-memory store to manage online users
const UserStore = (function () {
const _data = {};

// Add a user to the online users store
const addOnlineUser = (key, val) => {
_data[key] = val;
return val;
};

// Retrieve an online user's socket ID
const getOnlineUser = (key) => {
return _data[key];
};

// Remove a user from the online users store
const deleteOnlineUser = (key) => {
const val = _data[key];
delete _data[key];
return val;
};

// Get a list of all online users' IDs
const getAllOnlineUsers = () => {
return Object.keys(_data);
};

// Return an object exposing the functions to manage the online users store
return {
addOnlineUser,
getOnlineUser,
deleteOnlineUser,
getAllOnlineUsers,
};
}());

// Export the UserStore to be used in other parts of the application
exports.UserStore = UserStore;

// Render chat page with data, including chat history and user online status
exports.renderChatWithData = async (req, res) => {
const otherUser = await getUser(req.params.id);
const token = req.cookies.jwt;

if (token) {
jwt.verify(token, jwtSecret, async (err, decodedToken) => {
```

```
if (err) {
return res.status(401).json({ message: "Not authorized" });
} else {
const currUser = await getUser(decodedToken.id);
const chats = await getChat({ fromId: currUser.id, toId: otherUser.id });
otherUser.isOnline = !!UserStore.getOnlineUser(otherUser.id);
res.render("chat", { currUser, otherUser, chats });
}
});
} else {
return res.status(401).json({ message: "Not authorized, token not available" });
}
};

// Retrieve user information from the token and associate the user with the socket
const getUserInfo = async (cookie, socket) => {
// Extract the token from the cookie string
const token = cookie ? cookie.split('jwt=')[1] : null;

// Log the token to check its value
console.log(token);

// Verify the token if it exists
if (token) {
jwt.verify(token, jwtSecret, async (err, decodedToken) => {
if (err) {
console.error('JWT verification failed:', err);
return;
}

// Fetch user details and store in the socket data
const user = await getUser(decodedToken.id);
socket.data = user;

// Add the user to the online users store
UserStore.addOnlineUser(user.id, socket.id);
});
}
};

// Initialize Socket.IO server and set up event listeners
exports.creatSocket = (app) => {
const io = new Server(app);

// Middleware to handle authentication and associate user data with socket
io.use((socket, next) => {
getUserInfo(socket.handshake.headers.cookie, socket);
next();
});

// Handle socket connections and events
io.on('connection', (socket) => {
// Handle private messages between users
socket.on("private message", (msg) => {
const anotherSocketId = UserStore.getOnlineUser(msg.to);
socket.to(anotherSocketId).emit("private message", { ...msg, from: socket.data });
createMessage({ fromId: socket?.data?.id, toId: msg.to, message: msg.message });
});
```

```
// Handle user disconnection
socket.on('disconnect', () => {
let disconnected;
try {
disconnected = UserStore.deleteOnlineUser(socket?.data?.id);
} catch (e) {
console.log("User is already disconnected", e);
}
console.log(socket?.data?.id, 'user disconnected', disconnected);
});
});
};
```

6th folder VIEWS

Purpose of views/admin.ejs
The views/admin.ejs file is a server-side rendered HTML template for the admin page of the chat application. It provides the following functionalities:

Display User List:

Fetch Users: Retrieves the list of users from the server and displays them in an unordered list.
User Information: Shows the username and role of each user, except for the admin user.
User Management:

Edit User Role: Allows the admin to change a user's role to 'admin' by clicking the "Edit Role" button.
Delete User: Enables the admin to delete a user by clicking the "Delete User" button.
Status Messages:

Display Messages: Uses a <div> with a red background to show status messages related to user management actions (e.g., errors or success messages).
Logout:

Logout Button: Provides a button to log out and redirect the user to the /logout route.

code with explanation

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Admin page</title>
</head>
<body>
<!-- A div to display status messages, styled with a red background color -->
<div class="display" style="background-color: red;"></div>

<!-- Heading for the admin page -->
<h1>Users</h1>

<!-- Unordered list to display the list of users -->
<ul></ul>

<!-- Button for logging out, which links to the /logout route -->
```

```
<button class="logout"> <a href="/logout">Log Out</a></button>

<script>
// Select the unordered list and the display div from the DOM
const ul = document.querySelector('ul');
const display = document.querySelector('.display');

// Function to fetch and display users
const getUsers = async () => {
// Fetch the list of users from the API
const res = await fetch('/api/auth/getUsers');
const data = await res.json();

// Map through the users and create list items for each user
data.user.map((mappedUser) => {
if (mappedUser.username !== 'admin') {
// Create a list item with user information and buttons for editing/deleting
let li = `<li> <b>Username</b> => ${mappedUser.username} <br> <b>Role</b> => ${mappedUser.role}
</li> <button class="edit">Edit Role</button> <button class="delete">Delete User</button>`;
ul.innerHTML += li;
} else {
return null;
}

// Select the edit and delete buttons
const editRole = document.querySelectorAll('.edit');
const deleteUser = document.querySelectorAll('.delete');

// Add event listeners to the edit buttons
editRole.forEach((button, i) => {
button.addEventListener('click', async () => {
// Clear any previous messages
display.textContent = '';

// Get the ID of the user to be updated
const id = data.user[i+1].id;

// Send a PUT request to update the user's role to 'admin'
const res = await fetch('/api/auth/update', {
method: 'PUT',
body: JSON.stringify({ role: 'admin', id }),
headers: { 'Content-Type': 'application/json' }
});
const dataUpdate = await res.json();

// Handle errors and redirect if successful
if (res.status === 400 || res.status === 401) {
document.body.scrollTop = 0;
document.documentElement.scrollTop = 0;
return display.textContent = `${dataUpdate.message}. ${dataUpdate.error ? dataUpdate.error : ''}`;
}
location.assign('/admin');
});
});

// Add event listeners to the delete buttons
deleteUser.forEach((button, i) => {
button.addEventListener('click', async () => {
```

```
// Clear any previous messages
display.textContent = '';

// Get the ID of the user to be deleted
const id = data.user[i+1].id;

// Send a DELETE request to delete the user
const res = await fetch('/api/auth/deleteUser', {
method: 'DELETE',
body: JSON.stringify({ id }),
headers: { 'Content-Type': 'application/json' }
});
const dataDelete = await res.json();

// Handle errors and redirect if successful
if (res.status === 401) {
document.body.scrollTop = 0;
document.documentElement.scrollTop = 0;
return display.textContent = `${dataDelete.message}. ${dataDelete.error ? dataDelete.error : ''}`;
}
location.assign('/admin');
});
});
});
};

// Call the getUsers function to initialize the user list
getUsers();
</script>
</body>
</html>
```

views/chat.ejs

Purpose of views/chat.ejs
The views/chat.ejs file is a template for the chat interface of the VibeConnect application. It includes:

Chat Interface:

Chat Area: A main container for the chat area, which includes the chat header, message list, and input form.
Header: Displays the chat title (username of the person being chatted with) and icons for video and voice calls.
Messages List: A list where individual chat messages are displayed.
Message Input and Controls:

Input Field: For typing and sending messages.
Send Button: To submit messages.
Attach Button: For attaching files (though the functionality for file attachments isn't detailed in the provided code).
Styling:

Background: Uses a dynamic gradient background with an animation for a sunset effect.
Chat Bubbles: Styled message bubbles to differentiate between sent and received messages.
JavaScript Functionality:

Socket.IO: Connects to a Socket.IO server for real-time messaging.
Date Formatting: Uses Moment.js for formatting message timestamps.
Event Handling: Handles form submissions to send messages and updates the chat interface with new

messages from the server.
Dynamic Content:

User Data: Displays the chat header with the other user's name and initializes the chat with existing messages.
In summary, this file creates a responsive and visually appealing chat interface, allowing real-time communication between users with support for handling messages and integrating with a backend server using Socket.IO.

code with explanation

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>VibeConnect Chat</title>
<!-- Include FontAwesome for icons -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css">
<!-- Include Socket.IO client library -->
<script src="https://cdn.socket.io/4.0.0/socket.io.min.js"></script>
<!-- Include Moment.js for date formatting -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.1/moment.min.js"></script>
<!-- Include Moment Timezone for timezone support -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/moment-timezone/0.5.34/moment-timezone-with-
data.min.js"></script>
<style>
/* Style for the entire page */
body {
margin: 0;
font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica, Arial, sans-serif;
background: linear-gradient(to bottom, #FF512F, #DD2476);
animation: sunset 10s infinite;
height: 100vh;
display: flex;
justify-content: center;
align-items: center;
}

/* Animation for background gradient */
@keyframes sunset {
0% {
background: linear-gradient(to bottom, #FF512F, #DD2476);
}
50% {
background: linear-gradient(to bottom, #FE6D73, #FF8966);
}
100% {
background: linear-gradient(to bottom, #FF512F, #DD2476);
}
}

/* Container for chat area */
.chat-area {
position: absolute;
width: min((100% - 6px), 750px);
border: none;
top: 0;
```

```
left: 0;
bottom: 0;
background: rgba(0, 0, 0, 0.1);
backdrop-filter: blur(10px);
display: flex;
flex-direction: column;
}

/* Header style */
#head {
background: rgba(0, 0, 0, 0.15);
color: #fff;
padding: 24px;
padding-top: 12px;
font-size: 24px;
padding-bottom: 12px;
backdrop-filter: blur(10px);
display: flex;
align-items: center;
justify-content: space-between;
}

/* Icons in the header */
#head .icons {
display: flex;
gap: 15px;
}

#head .icons i {
cursor: pointer;
font-size: 1.5rem;
}

/* Style for chat input form */
#form {
background: rgba(0, 0, 0, 0.15);
padding: 0.25rem;
display: flex;
height: 3rem;
box-sizing: border-box;
backdrop-filter: blur(10px);
position: fixed;
left: 0;
bottom: 0;
width: min((100% - 6px), 750px);
}

/* Style for chat input field */
#input {
border: none;
padding: 0 1rem;
flex-grow: 1;
border-radius: 2rem;
margin: 0.25rem;
}

#input:focus {
outline: none;
```

```
}

/* Style for buttons in the form */
#form>button {
background: #333;
border: none;
padding: 0 1rem;
margin: 0.25rem;
border-radius: 3px;
outline: none;
color: #fff;
display: flex;
align-items: center;
justify-content: center;
}

#form>button#attach-button {
background: #444;
}

#form>button i {
font-size: 1.2rem;
}

/* Style for messages container */
#messages {
flex-grow: 1;
display: flex;
flex-direction: column;
list-style-type: none;
margin: 0;
padding: 0;
overflow-y: auto;
padding-bottom: 3rem;
}

/* Style for message bubbles */
.bubble {
--r: 1em;
--t: 1.5em;
max-width: 300px;
padding-right: 1em;
padding-left: 1em;
border-inline: var(--t) solid #0000;
border-radius: calc(var(--r) + var(--t)) / var(--r);
mask: radial-gradient(100% 100% at var(--_p) 0, #0000 99%, #000 102%) var(--_p) 100% / var(--t) var(--t)
no-repeat, linear-gradient(#000 0 0) padding-box;
background: linear-gradient(135deg, #FE6D00, #1384C5) border-box;
color: #fff;
margin-top: 16px;
width: 75%;
}

.bubble>.name {
padding: 4px;
font-size: 12px;
padding-left: 0;
}
```

```
.bubble>.msg {
font-size: 16px;
background: rgba(0, 0, 0, 0.15);
padding: 12px;
border-radius: 4px;
}

.bubble>.sent_on {
padding: 4px;
font-size: 12px;
padding-left: 0;
}

.left {
--_p: 0;
border-bottom-left-radius: 0 0;
place-self: start;
}

.right {
--_p: 100%;
border-bottom-right-radius: 0 0;
place-self: end;
}
</style>
</head>
<body>
<!-- Main chat area -->
<div class="chat-area">
<!-- Header with chat title and icons -->
<div id="head">
<div id="head-title">Chat with <span id="other-username"></span></div>
<div class="icons">
<i class="fa-solid fa-video" id="video-call"></i>
<i class="fa-solid fa-phone" id="voice-call"></i>
</div>
</div>
<!-- List to display messages -->
<ul id="messages"></ul>
<!-- Form for sending messages -->
<form id="form" action="">
<input id="input" autocomplete="off" />
<button type="submit"><i class="fa-regular fa-comments"></i></button>
<button type="button" id="attach-button"><i class="fa-solid fa-paperclip"></i></button>
<input type="file" id="file-input" style="display: none;" />
</form>
</div>

<!-- Include Socket.IO client library -->
<script src="/socket.io/socket.io.js"></script>
<script>
// Select DOM elements
const head = document.getElementById('head');
const form = document.getElementById('form');
const input = document.getElementById('input');
const messages = document.getElementById('messages');
```

```
// Parse user data from server-side
const currUser = JSON.parse('<%- JSON.stringify(currUser) %>');
const otherUser = JSON.parse('<%- JSON.stringify(otherUser) %>');
const chats = JSON.parse('<%- JSON.stringify(chats) %>');

// Connect to Socket.IO server
const socket = io('localhost:5000');

// Function to format date
const getFormatedDate = (sent) => {
const sentDate = moment(sent);
const currentDate = moment();
let sentOn = "";
const diffDays = currentDate.diff(sentDate, 'days');
if (diffDays > 364) {
return sentDate.format("DD-MM-YYYY");
} else if (diffDays > 30) {
return sentDate.format("DD-MMM");
}
return sentDate.format("HH:mm");
};

// Function to set message in chat
const setMessage = ({
userFrom,
userTo,
message,
sentOn
}) => {
const fromName = otherUser.id == userFrom ? otherUser.username : 'You';
getFormatedDate(sentOn);
const item = document.createElement('div');
item.className = userFrom == currUser.id ? "bubble right" : "bubble left";
item.innerHTML = `
<div class="name">${fromName}</div>
<div class="msg">${message}</div>
<div class="sent_on">${getFormatedDate(sentOn)}</div>
`;
messages.appendChild(item);
};

// Handle form submission
form.addEventListener('submit', (e) => {
e.preventDefault();
if (input.value) {
socket.emit("private message", {
message: input.value,
from: currUser.id,
to: otherUser.id
});
setMessage({
userFrom: currUser.id,
userTo: otherUser.id,
message: input.value,
sentOn: new Date()
});
input.value = '';
}
});

// Handle incoming messages
socket.on("private message", (msg) => {
setMessage(msg);
window.scrollTo(0, document.body.scrollHeight);
});

// Initialize chat with existing messages
if (Array.isArray(chats)) {
chats.map(setMessage);
}

// Set chat header title
head.innerHTML = otherUser.username;
</script>
</body>
</html>
```

views/home.ejs

Purpose of views/home.ejs
The views/home.ejs file is the template for the homepage of the VibeConnect application. It includes:

Page Layout:

Header: Displays the main title "VibeConnect" with a RocketChat icon, styled with a font from Google Fonts and FontAwesome.
Buttons: Provides links to the login and registration pages, styled with different background colors for differentiation.
Visual Design:

Background Gradient: Animated gradient background that transitions through multiple colors.
Button Styling: Custom styles for buttons, including hover effects and shadowing for a modern look.
Chat Rain Effect: A visual effect where chat messages fall from the top of the screen, created with JavaScript to animate chat bubbles.
JavaScript:

Chat Rain Effect: Script that dynamically generates and animates chat bubbles to create a falling effect, giving the page a unique and engaging visual element.
Credit:

Footer: Includes a credit section with a custom font and styling.
In summary, this file serves as the landing page for the VibeConnect application, featuring an eye-catching gradient background, engaging chat rain effect, and interactive elements like buttons for user login and registration. The design focuses on providing a vibrant and dynamic user experience.

code with explanation

```
<!DOCTYPE html>
<html lang="en">

<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>VibeConnect</title>
<!-- Include Google Fonts for custom typography -->
<link rel="stylesheet" href="https://fonts.googleapis.com/css2?
```

```
family=Roboto:wght@700&family=Pacifico&display=swap">
<!-- Include FontAwesome for icons -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css">
<!-- Include Bootstrap Icons for additional icons -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-icons/1.10.5/font/bootstrap-
icons.min.css">
<!-- Favicon for the site -->
<link rel="icon" href="/path/to/your/favicon.ico" type="image/x-icon">
<style>
/* Style for the entire page */
body {
margin: 0;
font-family: 'Roboto', sans-serif;
background: linear-gradient(135deg, #ff007f, #ff7f00, #7fff00, #00ff7f, #00ffff, #7f00ff);
background-size: 600% 600%;
animation: gradient 15s ease infinite;
color: #333;
text-align: center;
display: flex;
justify-content: center;
align-items: center;
height: 100vh;
flex-direction: column;
overflow: hidden;
cursor: default;
}

/* Style for main heading */
h1 {
font-family: 'Pacifico', cursive;
font-size: 5rem;
color: #fff;
margin: 0;
text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.5);
position: relative;
z-index: 1;
}

/* Style for RocketChat icon within the heading */
h1 .fa-rocketchat {
font-size: 4rem;
margin-left: 10px;
}

/* Animation for background gradient */
@keyframes gradient {
0% {
background-position: 0% 0%;
}
50% {
background-position: 100% 100%;
}
100% {
background-position: 0% 0%;
}
}

/* Style for button container */
```

```
.button-container {
margin-top: 30px;
}

/* General button style */
.button {
font-family: 'Roboto', sans-serif;
font-size: 1.2rem;
font-weight: 700;
text-transform: uppercase;
padding: 10px 20px;
margin: 10px;
border: none;
border-radius: 10px;
cursor: pointer;
transition: transform 0.3s, box-shadow 0.3s;
position: relative;
display: flex;
align-items: center;
justify-content: center;
color: #fff;
background-color: #25D366;
box-shadow: 0 10px 20px rgba(0, 0, 0, 0.2);
}

/* Hover effect for buttons */
.button:hover {
transform: translateY(-5px);
box-shadow: 0 15px 30px rgba(0, 0, 0, 0.3);
}

/* Specific button styles for login and register */
.login-btn {
background-color: #128C7E;
}

.register-btn {
background-color: #34B7F1;
}

/* Style for credit section */
.credit {
font-family: 'Pacifico', cursive;
font-size: 1rem;
color: #fff;
margin: 20px 0 0;
text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.5);
}

/* Style for chat rain effect */
.chat-rain {
position: absolute;
top: 0;
left: 0;
width: 100vw;
height: 100vh;
overflow: hidden;
z-index: 0;
}

/* Style for individual chat bubbles in the rain effect */
.chat {
position: absolute;
font-size: 0.8rem;
font-family: 'Roboto', sans-serif;
color: #333;
animation: fall linear infinite;
}

/* Animation for falling chat bubbles */
@keyframes fall {
to {
transform: translateY(100vh);
}
}
</style>
</head>

<body>
<!-- Container for falling chat bubbles effect -->
<div class="chat-rain" id="chatRain"></div>
<!-- Main heading with site name and RocketChat icon -->
<h1>
VibeConnect
<i class="fa-brands fa-rocketchat" id="logo"></i>
</h1>
<!-- Container for login and register buttons -->
<div class="button-container">
<a href="/login" class="button login-btn">Login</a>
<a href="/register" class="button register-btn">Register</a>
</div>
<!-- Credit section -->
<div class="credit">
by Db
</div>
<!-- JavaScript for generating falling chat bubbles effect -->
<script>
document.addEventListener('DOMContentLoaded', () => {
const chatRainContainer = document.querySelector('#chatRain');
const chats = ['hi', 'hello', 'bye', 'take care', 'asap', 'kkrh'];
const numChats = 100;

for (let i = 0; i < numChats; i++) {
const chat = document.createElement('div');
chat.classList.add('chat');
chat.innerHTML = `<i class="bi bi-chat"></i> ${chats[Math.floor(Math.random() * chats.length)]}`;
chat.style.left = `${Math.random() * 100}vw`;
chat.style.animationDuration = `${Math.random() * 2 + 3}s`;
chat.style.opacity = `${Math.random() * 0.5 + 0.3}`;
chatRainContainer.appendChild(chat);
}
});
</script>
</body>

</html>
```

views/login.ejs

Purpose of login.ejs
User Authentication:

The primary function of this page is to allow users to log in to the chat application. Users input their username and password, which are then sent to the server for authentication.
Visual Appeal:

The file includes various design elements to make the login page visually engaging. This includes background colors, animated shapes, and a rain animation effect, which helps in creating a pleasant user experience.
Responsive Design:

The page is styled to be responsive, ensuring it looks good on different devices and screen sizes. This is achieved using CSS properties and media queries.
Social Media Integration:

The page includes icons for social media logins (Google, Meta, Facebook). This allows users to log in using their social media accounts, providing an alternative to the traditional username and password login.
Error Handling:

It displays error messages if the login attempt fails, such as incorrect username or password. This feedback helps users understand what went wrong and how to correct it.
Navigation:

It provides links for users to navigate to other parts of the application, such as "Forgot Password" and "Sign Up" pages. This improves user flow and accessibility.

code with explanation

```
<!DOCTYPE html>
<html lang="en">

<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Login Form</title>
<!-- Include Google Fonts for custom typography -->
<link href="https://fonts.googleapis.com/css2?family=Poppins:wght@300;500;600&display=swap"
rel="stylesheet">
<link href="https://fonts.googleapis.com/css2?family=Pacifico&display=swap" rel="stylesheet">
<!-- Include FontAwesome for icons -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css">
<style>
/* Reset margin and padding, set box-sizing and font-family */
* {
margin: 0;
padding: 0;
box-sizing: border-box;
font-family: "Poppins", sans-serif;
}

/* Body style for hidden overflow */
body {
overflow: hidden;
position: relative;
}
```

```
/* Style for the main section */
section {
display: flex;
min-height: 100vh;
align-items: center;
justify-content: center;
background: linear-gradient(to bottom, #f7f7fe, #dff1ff);
position: relative;
z-index: 1;
}

/* Styles for background color shapes */
section .colour {
position: absolute;
filter: blur(150px);
}

section .colour:nth-child(1) {
top: -350px;
width: 600px;
height: 600px;
background: #bf4ad4;
}

section .colour:nth-child(2) {
left: 100px;
width: 500px;
height: 500px;
bottom: -150px;
background: #ffa500;
}

section .colour:nth-child(3) {
right: 100px;
bottom: 50px;
width: 300px;
height: 300px;
background: #2b67f3;
}

/* Style for animated squares */
.box {
position: relative;
z-index: 2;
}

.box .square {
position: absolute;
border-radius: 10px;
backdrop-filter: blur(5px);
background: rgba(255, 255, 255, 0.1);
animation-delay: calc(-1s * var(--i));
animation: animate 10s linear infinite;
box-shadow: 0 25px 45px rgba(0, 0, 0, 0.1);
border: 1px solid rgba(255, 255, 255, 0.5);
border-right: 1px solid rgba(255, 255, 255, 0.2);
border-bottom: 1px solid rgba(255, 255, 255, 0.2);
```

```css
}

@keyframes animate {
0%,
100% {
transform: translateY(-40px);
}

50% {
transform: translateY(40px);
}
}

.box .square:nth-child(1) {
top: -50px;
left: -60px;
width: 100px;
height: 100px;
}

.box .square:nth-child(2) {
z-index: 2;
top: 150px;
left: -100px;
width: 120px;
height: 120px;
}

.box .square:nth-child(3) {
z-index: 2;
width: 80px;
height: 80px;
right: -50px;
bottom: -60px;
}

.box .square:nth-child(4) {
left: 100px;
width: 50px;
height: 50px;
bottom: -80px;
}

.box .square:nth-child(5) {
top: -80px;
left: 140px;
width: 60px;
height: 60px;
}

/* Style for the login container */
.container {
width: 400px;
display: flex;
min-height: 400px;
position: relative;
border-radius: 10px;
align-items: center;
```

```
justify-content: center;
backdrop-filter: blur(5px);
background: rgba(255, 255, 255, 0.1);
box-shadow: 0 25px 45px rgba(0, 0, 0, 0.1);
border: 1px solid rgba(255, 255, 255, 0.5);
border-right: 1px solid rgba(255, 255, 255, 0.2);
border-bottom: 1px solid rgba(255, 255, 255, 0.2);
z-index: 2;
}

/* Style for the login form */
.form {
width: 100%;
height: 100%;
padding: 40px;
position: relative;
}

.form h2 {
color: #000000;
font-size: 24px;
font-weight: 600;
position: relative;
letter-spacing: 1px;
margin-bottom: 40px;
}

.form h2::before {
left: 0;
width: 80px;
height: 4px;
content: "";
bottom: -10px;
background: #080808;
position: absolute;
}

.form .input__box {
width: 100%;
margin-top: 20px;
}

.form .input__box input {
width: 100%;
color: #000000;
border: none;
outline: none;
font-size: 16px;
padding: 10px 20px;
letter-spacing: 1px;
border-radius: 35px;
background: rgba(255, 255, 255, 0.2);
border: 1px solid rgba(255, 255, 255, 0.5);
box-shadow: 0 5px 15px rgba(0, 0, 0, 0.05);
border-right: 1px solid rgba(255, 255, 255, 0.2);
border-bottom: 1px solid rgba(255, 255, 255, 0.2);
}
```

```css
.form ::placeholder {
color: #fff;
}

.form .input__box input[type="submit"] {
color: #666;
cursor: pointer;
background: #fff;
max-width: 100px;
font-weight: 600;
margin-bottom: 20px;
}

.forget {
color: #000000;
margin-top: 5px;
}

.forget a {
color: #070707;
font-weight: 600;
text-decoration: none;
}

/* Style for social media icons */
.social-icons {
margin-top: 20px;
}

.social-icons i {
font-size: 24px;
margin: 0 10px;
color: #333;
cursor: pointer;
transition: color 0.3s;
}

.social-icons i:hover {
color: #000;
}

/* Style for rain animation */
.rain {
position: absolute;
top: 0;
left: 0;
width: 100vw;
height: 100vh;
overflow: hidden;
z-index: 0;
}

.rain .drop {
position: absolute;
width: 2px;
height: 90px;
background: rgba(255, 255, 255, 0.6);
animation: fall linear infinite;
}

@keyframes fall {
to {
transform: translateY(100vh);
}
}

/* Style for top corner text */
.top-corner-text {
position: absolute;
top: 20px;
left: 20px;
font-family: 'Pacifico', cursive;
font-size: 32px;
color: black;
z-index: 2;
}
</style>
</head>

<body>
<section>
<!-- Background color shapes -->
<div class="colour"></div>
<div class="colour"></div>
<div class="colour"></div>
<div class="box">
<!-- Animated squares -->
<div class="square" style="--i: 0"></div>
<div class="square" style="--i: 1"></div>
<div class="square" style="--i: 2"></div>
<div class="square" style="--i: 3"></div>
<div class="square" style="--i: 4"></div>
<!-- Login form container -->
<div class="container">
<div class="form">
<h2>LOGIN</h2>
<form id="loginForm">
<div class="input__box">
<input type="text" placeholder="Username" id="username" />
</div>
<div class="input__box">
<input type="password" placeholder="Password" id="password" />
</div>
<div class="input__box">
<input type="submit" value="Login" />
</div>
<p class="forget">
Forgot Password? <a href="#">Click Here</a>
</p>
<p class="forget">
Don't have an account? <a href="/register">Sign Up</a>
</p>
<!-- Social media login icons -->
<div class="social-icons">
<i class="fa-brands fa-google" id="googleLogin"></i>
<i class="fa-brands fa-meta" id="metaLogin"></i>
```

```
<i class="fa-brands fa-facebook" id="facebookLogin"></i>
</div>
<!-- Error message display -->
<h5 class="error"></h5>
</form>
</div>
</div>
</div>
<!-- Rain animation container -->
<div class="rain" id="rain"></div>
<!-- Top corner text -->
<div class="top-corner-text">VibeConnect</div>
</section>
<script>
document.addEventListener('DOMContentLoaded', () => {
const form = document.querySelector('#loginForm');
const username = document.querySelector('#username');
const password = document.querySelector('#password');
const display = document.querySelector('.error');

form.addEventListener('submit', async (e) => {
e.preventDefault();
display.textContent = '';
try {
const res = await fetch('/api/auth/login', {
method: 'POST',
body: JSON.stringify({ username: username.value, password: password.value }),
headers: { 'Content-Type': 'application/json' }
});
const data = await res.json();
if (res.status === 400 || res.status === 401) {
return display.textContent = `${data.message}. ${data.error ? data.error : ''}`;
}
data.role === "admin" ? location.assign('/admin') : location.assign('/basic');
} catch (err) {
console.log(err.message);
}
});

// Rain animation
const rainContainer = document.querySelector('#rain');
const numDrops = 100;

for (let i = 0; i < numDrops; i++) {
const drop = document.createElement('div');
drop.classList.add('drop');
drop.style.left = `${Math.random() * 100}vw`;
drop.style.animationDuration = `${Math.random() * 1.5 + 0.5}s`;
drop.style.opacity = `${Math.random() * 0.5 + 0.3}`;
rainContainer.appendChild(drop);
}

// Social media login
document.getElementById('googleLogin').addEventListener('click', () => {
location.assign('/auth/google');
});

document.getElementById('metaLogin').addEventListener('click', () => {
```

```
location.assign('/auth/meta');
});

document.getElementById('facebookLogin').addEventListener('click', () => {
location.assign('/auth/facebook');
});
});
</script>
</body>

</html>
```

views/Register.ejs

Purpose of the File in a Chat Application
The register.ejs file is a template for a registration page in a web application. Here's a breakdown of its purpose:

User Registration: It provides a form where users can enter their username, email, password, and confirm their password. This data is sent to the server to create a new user account.

Styling and Design: The file includes various CSS styles to create a visually appealing page. This includes a background gradient, animated shapes, and a rain effect to enhance the user experience.

Form Validation: The embedded JavaScript checks that the password and confirm password fields match before submitting the form. It also handles server responses to display error messages if registration fails or redirects users based on their role upon successful registration.

Error Handling: Displays appropriate error messages if the form submission encounters issues, such as password mismatches or server errors.

code with explanation

```
<!DOCTYPE html>
<html lang="en">
<head>
<!-- Charset and viewport settings for responsive design -->
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<!-- Title of the page -->
<title>Register</title>
<!-- Importing fonts from Google Fonts -->
<link href="https://fonts.googleapis.com/css2?family=Poppins:wght@300;500;600&display=swap"
rel="stylesheet">
<link href="https://fonts.googleapis.com/css2?family=Pacifico&display=swap" rel="stylesheet">
<style>
/* General reset and font settings */
* {
margin: 0;
padding: 0;
box-sizing: border-box;
font-family: "Poppins", sans-serif;
}
/* Body styling */
body {
overflow: hidden; /* Prevent scrollbars */
position: relative;
}
```

```css
/* Section styling for the main container */
section {
display: flex;
min-height: 100vh; /* Full viewport height */
align-items: center;
justify-content: center;
background: linear-gradient(to bottom, #f7f7fe, #dff1ff); /* Gradient background */
position: relative;
z-index: 1;
}
/* Colorful blurred circles for background effect */
section .colour {
position: absolute;
filter: blur(150px);
}
section .colour:nth-child(1) {
top: -350px;
width: 600px;
height: 600px;
background: #bf4ad4;
}
section .colour:nth-child(2) {
left: 100px;
width: 500px;
height: 500px;
bottom: -150px;
background: #ffa500;
}
section .colour:nth-child(3) {
right: 100px;
bottom: 50px;
width: 300px;
height: 300px;
background: #2b67f3;
}
/* Styling for animated squares */
.box {
position: relative;
z-index: 2;
}
.box .square {
position: absolute;
border-radius: 10px;
backdrop-filter: blur(5px);
background: rgba(255, 255, 255, 0.1);
animation-delay: calc(-1s * var(--i));
animation: animate 10s linear infinite;
box-shadow: 0 25px 45px rgba(0, 0, 0, 0.1);
border: 1px solid rgba(255, 255, 255, 0.5);
border-right: 1px solid rgba(255, 255, 255, 0.2);
border-bottom: 1px solid rgba(255, 255, 255, 0.2);
}
/* Keyframes for animation of squares */
@keyframes animate {
0%, 100% { transform: translateY(-40px); }
50% { transform: translateY(40px); }
}
/* Positioning and sizing of animated squares */
```

```
.box .square:nth-child(1) {
top: -50px;
left: -60px;
width: 100px;
height: 100px;
}
.box .square:nth-child(2) {
z-index: 2;
top: 150px;
left: -100px;
width: 120px;
height: 120px;
}
.box .square:nth-child(3) {
z-index: 2;
width: 80px;
height: 80px;
right: -50px;
bottom: -60px;
}
.box .square:nth-child(4) {
left: 100px;
width: 50px;
height: 50px;
bottom: -80px;
}
.box .square:nth-child(5) {
top: -80px;
left: 140px;
width: 60px;
height: 60px;
}
/* Container for the registration form */
.container {
width: 400px;
display: flex;
min-height: 400px;
position: relative;
border-radius: 10px;
align-items: center;
justify-content: center;
backdrop-filter: blur(5px);
background: rgba(255, 255, 255, 0.1);
box-shadow: 0 25px 45px rgba(0, 0, 0, 0.1);
border: 1px solid rgba(255, 255, 255, 0.5);
border-right: 1px solid rgba(255, 255, 255, 0.2);
border-bottom: 1px solid rgba(255, 255, 255, 0.2);
z-index: 2;
}
/* Styling for the form */
.form {
width: 100%;
height: 100%;
padding: 40px;
position: relative;
}
/* Header of the form */
.form h2 {
```

```
color: #000000;
font-size: 24px;
font-weight: 600;
position: relative;
letter-spacing: 1px;
margin-bottom: 40px;
}
.form h2::before {
left: 0;
width: 80px;
height: 4px;
content: "";
bottom: -10px;
background: #080808;
position: absolute;
}
/* Styling for input boxes */
.form .input__box {
width: 100%;
margin-top: 20px;
}
.form .input__box input {
width: 100%;
color: #000000;
border: none;
outline: none;
font-size: 16px;
padding: 10px 20px;
letter-spacing: 1px;
border-radius: 35px;
background: rgba(255, 255, 255, 0.2);
border: 1px solid rgba(255, 255, 255, 0.5);
box-shadow: 0 5px 15px rgba(0, 0, 0, 0.05);
border-right: 1px solid rgba(255, 255, 255, 0.2);
border-bottom: 1px solid rgba(255, 255, 255, 0.2);
}
/* Placeholder styling */
.form ::placeholder {
color: #fff;
}
/* Styling for submit button */
.form .input__box input[type="submit"] {
color: #666;
cursor: pointer;
background: #fff;
max-width: 100px;
font-weight: 600;
margin-bottom: 20px;
}
/* Styling for the forgot password link */
.forget {
color: #000000;
margin-top: 5px;
}
.forget a {
color: #070707;
font-weight: 600;
text-decoration: none;
```

```
}

/* Rain animation for background effect */
.rain {
position: absolute;
top: 0;
left: 0;
width: 100vw;
height: 100vh;
overflow: hidden;
z-index: 0;
}
.rain .drop {
position: absolute;
width: 2px;
height: 90px;
background: rgba(255, 255, 255, 0.6);
animation: fall linear infinite;
}
@keyframes fall {
to {
transform: translateY(100vh);
}
}

/* Top corner text styling */
.top-corner-text {
position: absolute;
top: 20px;
left: 20px;
font-family: 'Pacifico', cursive;
font-size: 32px;
color: black;
z-index: 2;
}
/* Error message styling */
.error {
color: #ff512f;
}
</style>
</head>
<body>
<section>
<!-- Background color shapes -->
<div class="colour"></div>
<div class="colour"></div>
<div class="colour"></div>
<!-- Animated squares -->
<div class="box">
<div class="square" style="--i: 0"></div>
<div class="square" style="--i: 1"></div>
<div class="square" style="--i: 2"></div>
<div class="square" style="--i: 3"></div>
<div class="square" style="--i: 4"></div>
<!-- Registration form container -->
<div class="container">
<div class="form">
<h2>REGISTER</h2>
```

```
<!-- Registration form -->
<form id="registerForm" action="/api/auth/register" method="POST">
<div class="input__box">
<input type="text" placeholder="Username" id="username" name="username" required />
</div>
<div class="input__box">
<input type="email" placeholder="Email" id="email" name="email" required />
</div>
<div class="input__box">
<input type="password" placeholder="Password" id="password" name="password" required />
</div>
<div class="input__box">
<input type="password" placeholder="Confirm Password" id="confirmPassword"
name="confirmPassword" required />
</div>
<div class="input__box">
<input type="submit" value="Register" />
</div>
<p class="forget">
Already have an account? <a href="/login">Login</a>
</p>
<p class="error"></p>
</form>
</div>
</div>
</div>
<!-- Branding text in the top corner -->
<div class="top-corner-text">VibeConnect</div>
<!-- Rain animation -->
<div class="rain"></div>
</section>
<script>
// JavaScript for form validation and submission
const form = document.querySelector('form');
const username = document.querySelector('#username');
const email = document.querySelector('#email');
const password = document.querySelector('#password');
const confirmPassword = document.querySelector('#confirmPassword');
const errorDisplay = document.querySelector('.error');

form.addEventListener('submit', async (e) => {
e.preventDefault(); // Prevent the default form submission

errorDisplay.textContent = ''; // Clear previous errors

// Check if passwords match
if (password.value !== confirmPassword.value) {
errorDisplay.textContent = 'Passwords do not match';
return;
}

try {
// Send form data to the server
const res = await fetch('/api/auth/register', {
method: 'POST',
body: JSON.stringify({
username: username.value,
email: email.value,
password: password.value
}),
headers: { 'Content-Type': 'application/json' }
});

const data = await res.json();

// Handle different responses based on status code
if (res.status === 400 || res.status === 401) {
errorDisplay.textContent = `${data.message}. ${data.error ? data.error : ''}`;
return;
}

// Redirect based on user role
if (data.role === "admin") {
location.assign('/admin');
} else {
location.assign('/basic');
}
} catch (err) {
// Handle any errors that occur during fetch
errorDisplay.textContent = 'An error occurred. Please try again later.';
console.log(err.message);
}
});
</script>
</body>
</html>
```

views/user.ejs

Purpose and Overview
Header Section:

Contains the page title ("Contacts") and a logout button.
The header has a background with a semi-transparent black overlay and uses Flexbox for layout.
Contacts List:

Displays a list of users fetched from the backend.
Shows each user's name and online status.
Includes a hover effect to make list items visually appealing.

Background Animation:

Uses a sunset gradient background that animates between two color schemes.
JavaScript:

Fetches user data from the backend and populates the list.
Handles the logout action

code with explanation

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
<title>User page</title>
<!-- Font Awesome for icons -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-
beta3/css/all.min.css" />
<!-- Google Fonts for styling -->
<link rel="stylesheet" href="https://fonts.googleapis.com/css2?
family=Roboto:wght@400;700&family=Open+Sans:wght@300;600&display=swap" />
<style>
/* Basic styling and background settings */
body {
margin: 0;
font-family: 'Roboto', sans-serif;
background: #555;
overflow: hidden;
}

/* Header styling */
#head {
background: rgba(0, 0, 0, 0.15);
color: #fff;
padding: 24px;
padding-top: 12px;
font-size: 24px;
padding-bottom: 12px;
display: flex;
align-items: center;
justify-content: space-between;
}

.title {
display: flex;
align-items: center;
}

.title i {
margin-right: 8px;
}

.logout {
color: #ccc;
font-size: 18px;
text-decoration: none;
display: flex;
align-items: center;
}

.logout i {
margin-right: 8px;
}

/* List styling */
ul {
padding: 0;
list-style-type: none;
margin: 0;
max-height: 400px; /* Set maximum height */
overflow-y: auto; /* Enable vertical scrolling */
}
```

```
li {
margin: 16px;
background: rgba(0, 0, 0, 0.25);
display: flex;
align-items: center;
padding: 10px;
border-radius: 8px;
box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);
transform: perspective(1000px) rotateX(2deg) rotateY(2deg);
transition: transform 0.3s ease-in-out;
}

li:hover {
transform: perspective(1000px) rotateX(0) rotateY(0);
}

li > a {
color: #ccc;
text-decoration: none;
display: flex;
align-items: center;
width: 100%;
justify-content: space-between;
}

.name {
font-size: 18px;
font-family: 'Open Sans', sans-serif;
font-weight: 600;
}

.status {
padding-top: 4px;
font-size: 14px;
font-family: 'Open Sans', sans-serif;
font-weight: 300;
}

/* Background animation */
.sunset-bg {
position: absolute;
top: 0;
left: 0;
width: 100%;
height: 100%;
background: linear-gradient(to bottom, #ff6e7f, #bfe9ff);
animation: sunset 10s infinite alternate;
z-index: -1;
}

@keyframes sunset {
0% {
background: linear-gradient(to bottom, #ff6e7f, #bfe9ff);
}
100% {
background: linear-gradient(to bottom, #ff9966, #ff5e62);
}
```

```
}

/* Customize scrollbar */
ul::-webkit-scrollbar {
width: 8px;
}

ul::-webkit-scrollbar-track {
background: rgba(0, 0, 0, 0.2);
}

ul::-webkit-scrollbar-thumb {
background-color: rgba(0, 0, 0, 0.4);
border-radius: 8px;
}
</style>
</head>
<body>
<!-- Background animation element -->
<div class="sunset-bg"></div>
<!-- Header with title and logout button -->
<div id="head">
<div class="title">
<i class="fa-regular fa-address-book"></i>
<p>Contacts</p>
</div>
<a class="logout" href="/" id="logout-button">
<i class="fa-solid fa-right-from-bracket"></i>
Logout
</a>
</div>
<!-- List to display contacts -->
<ul></ul>
<script>
const ul = document.querySelector("ul");

// Function to get users from the backend
const getUsers = async () => {
const res = await fetch("/api/auth/getUsers");
const data = await res.json();
data.user.forEach(mappedUser => {
if (mappedUser.username !== "admin") {
let li = `<li><a href='/chat/${mappedUser.id}'><p class="name">${mappedUser.username}</p>
<p class="status">${mappedUser.isOnline ? " Online" : " Offline"}</p></a></li>`;
ul.innerHTML += li;
}
});
};

// Call the function to load users
getUsers();

// Handle logout click
document.getElementById('logout-button').addEventListener('click', (event) => {
event.preventDefault(); // Prevent default anchor behavior
// Perform logout logic here (e.g., API call to log out)
// Redirect to home page
window.location.href = '/';
});
</script>
</body>
</html>
```

Main File (Server.js)

Purpose of server.js
The server.js file is responsible for:

Setting up the Express server.
Connecting to the database.
Configuring middleware for JSON parsing and cookie handling.
Defining routes for different parts of the application.
Initializing WebSocket connections for real-time communication.
Handling errors and ensuring the server can gracefully shut down if needed.

Imports and Dependencies:

express: Web framework used to build the server.
cookie-parser: Middleware for parsing cookies.
connectDB: Custom function to connect to the database.
adminAuth, userAuth: Middleware functions for authentication.
renderChatWithData, creatSocket: Functions for rendering chat pages and handling sockets.
Server Setup:

connectDB(): Connects to the database when the server starts.
app.use(express.json()): Configures the server to parse JSON request bodies.
app.use(cookieParser()): Parses cookies to handle authentication and other cookie-based functionality.
Routes:

app.use("/api/auth", require("./auth/route")): Mounts authentication-related routes.
app.use('/uploads', express.static('uploads')): Serves static files from the uploads directory.
The .get() methods define routes for rendering EJS templates and handling user interactions.
Error Handling:

process.on("unhandledRejection", err => {...}): Listens for unhandled promise rejections and logs the error.
Closes the server and exits the process if an error occurs.
Socket Initialization:

creatSocket(server): Initializes WebSocket connections.

Package.json

```
{
"name": "javascriptp1", // Name of your project
"version": "1.0.0", // Current version of your project
"main": "index.js", // Entry point file for the project (though it's not used directly in your setup)
"scripts": {
"start": "node server.js", // Command to start the server in production
"dev": "nodemon server.js", // Command to start the server with nodemon for development (automatic
restarts on file changes)
"test": "echo \"Error: no test specified\" && exit 1" // Placeholder test command, which currently just outputs
an error
},
"author": "", // Author of the project (currently empty)
"license": "ISC", // License type (ISC in this case)
"description": "", // Description of the project (currently empty)
"dependencies": {
"auth0": "^4.5.0", // Auth0 authentication library
"bcrypt": "^5.1.1", // Library for hashing passwords
"bcryptjs": "^2.4.3", // Alternative to bcrypt for hashing passwords (included as a backup)
"cookie-parser": "^1.4.6", // Middleware for parsing cookies
"dotenv": "^16.4.5", // Library for loading environment variables from a .env file
"ejs": "^3.1.10", // Embedded JavaScript templating engine for rendering views
"express": "^4.19.2", // Web framework for building server-side applications
"jsonwebtoken": "^9.0.2", // Library for creating and verifying JSON Web Tokens (JWTs)
"mongoose": "^8.6.0", // MongoDB object modeling tool for Node.js
"multer": "^1.4.5-lts.1", // Middleware for handling multipart/form-data (file uploads)
"nodemon": "^3.1.4", // Development tool for automatically restarting the server on code changes
"passport": "^0.7.0", // Middleware for authentication
"passport-facebook": "^3.0.0", // Passport strategy for Facebook authentication
"passport-google-oauth20": "^2.0.0", // Passport strategy for Google OAuth 2.0 authentication
"socket.io": "^4.7.5" // Library for real-time communication using WebSockets
}
}
```