1. [backend](https://github.com/burakorkmez/mern-chat-app/tree/master/backend)
2. /[controllers](https://github.com/burakorkmez/mern-chat-app/tree/master/backend/controllers)

/

**auth.controller.js**

This code is for user authentication in a Node.js application using Express.js, Mongoose, and bcrypt.js. It includes three main functions: signup, login, and logout.

1. **signup**: This function is used to register a new user. It first checks if the password and confirm password fields match. Then, it checks if the username already exists in the database. If the username is unique, it hashes the password using bcrypt.js and creates a new user with the provided details. It also generates a JWT token for the new user and saves the user to the database.
2. **login**: This function is used to authenticate an existing user. It checks if the username exists in the database and if the provided password matches the hashed password stored in the database. If the credentials are valid, it generates a JWT token for the user.
3. **logout**: This function is used to log out a user. It clears the JWT token from the cookies.

The generateTokenAndSetCookie function is used to generate a JWT token for a user and set it as a cookie. This function is not defined in this code, but it’s imported from another module.

Please note that this code should be used as a part of a larger Express.js application, and appropriate error handling and validation should be added for production use. Also, remember to keep your JWT secret key secure and never expose it in your client-side code or version control system. Happy coding! 😊

Signup function

Sure, let’s break down the signup function:

export const signup = async (req, res) => {

try {

const { fullName, username, password, confirmPassword, gender } = req.body;

if (password !== confirmPassword) {

return res.status(400).json({ error: "Passwords don't match" });

}

const user = await User.findOne({ username });

if (user) {

return res.status(400).json({ error: "Username already exists" });

}

const salt = await bcrypt.genSalt(10);

const hashedPassword = await bcrypt.hash(password, salt);

const boyProfilePic = `https://avatar.iran.liara.run/public/boy?username=${username}`;

const girlProfilePic = `https://avatar.iran.liara.run/public/girl?username=${username}`;

const newUser = new User({

fullName,

username,

password: hashedPassword,

gender,

profilePic: gender === "male" ? boyProfilePic : girlProfilePic,

});

if (newUser) {

generateTokenAndSetCookie(newUser.\_id, res);

await newUser.save();

res.status(201).json({

\_id: newUser.\_id,

fullName: newUser.fullName,

username: newUser.username,

profilePic: newUser.profilePic,

});

} else {

res.status(400).json({ error: "Invalid user data" });

}

} catch (error) {

console.log("Error in signup controller", error.message);

res.status(500).json({ error: "Internal Server Error" });

}

};

1. **Extract User Input**: The function begins by extracting the user’s input from the request body. This includes the user’s full name, username, password, confirm password, and gender.
2. **Check Passwords**: It then checks if the password and confirm password fields match. If they don’t, it sends a 400 status code response with an error message.
3. **Check Existing User**: Next, it checks if a user with the provided username already exists in the database. If a user is found, it sends a 400 status code response with an error message.
4. **Hash Password**: If the username is unique, it hashes the password using bcrypt.js. This involves generating a salt and then hashing the password with the salt.
5. **Profile Picture**: It then generates a profile picture URL based on the user’s gender.
6. **Create New User**: It creates a new user with the provided details and the hashed password. The profile picture URL is set based on the user’s gender.
7. **Save User**: If the new user is valid, it generates a JWT token for the user, saves the user to the database, and sends a 201 status code response with the new user’s details. If the new user is not valid, it sends a 400 status code response with an error message.
8. **Error Handling**: If any error occurs during this process, it logs the error message and sends a 500 status code response with an error message.

This function is a part of a larger Express.js application and should be used in conjunction with other middleware and routing logic. It’s also important to note that this function assumes that the User model and generateTokenAndSetCookie function have been defined elsewhere in your application. Happy coding! 😊

What is salt?

[In cryptography, a salt is random data that is used as an additional input to a one-way function that hashes data, a password, or passphrase1](https://en.wikipedia.org/wiki/Salt_%28cryptography%29). [The purpose of a salt is to defend against attacks that use precomputed tables (e.g., rainbow tables) by vastly increasing the size of the table needed for a successful attack1](https://en.wikipedia.org/wiki/Salt_%28cryptography%29). Here are some key reasons why salts are used:

1. [**Unique Hashes**: Salts create unique hashes, even when users choose the same passwords2](https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/). This means that even if two users have the same password, their hashed passwords will be different because of the unique salts.
2. [**Preventing Precomputed Table Attacks**: Salts help mitigate hash table attacks by forcing attackers to re-compute the tables using the salts for each user](https://en.wikipedia.org/wiki/Salt_%28cryptography%29)[2](https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/). [This makes precomputed attacks, such as rainbow table attacks, computationally expensive and less feasible1](https://en.wikipedia.org/wiki/Salt_%28cryptography%29).
3. [**Protecting Identical Passwords**: If a database uses unique salts for every password, then even if two users have the same password, the hashed passwords stored in the database will be different1](https://en.wikipedia.org/wiki/Salt_%28cryptography%29). This adds an extra layer of security.

In practice, a salt is usually generated using existing data such as a user’s ID. [If a completely random salt is created instead, it is stored in the hash so the system can later recover it1](https://en.wikipedia.org/wiki/Salt_%28cryptography%29). [It’s important to note that the salt does not need to be encrypted, because knowing the salt would not help an attacker1](https://en.wikipedia.org/wiki/Salt_%28cryptography%29). Happy coding! 😊

1. [backend](https://github.com/burakorkmez/mern-chat-app/tree/master/backend)
2. /[middleware](https://github.com/burakorkmez/mern-chat-app/tree/master/backend/middleware)

/

**protectRoute.js**

This code is a middleware function for protecting routes in an Express.js application. It’s used to ensure that only authenticated users can access certain routes. Here’s how it works:

1. **Extract Token**: The function begins by extracting the JWT token from the cookies in the request object.
2. **Check Token**: If no token is provided, it sends a 401 status code response with an error message.
3. **Verify Token**: If a token is provided, it verifies the token using the jwt.verify method. This method decodes the token and checks its signature to ensure it’s valid.
4. **Check Decoded Token**: If the token is not valid, it sends a 401 status code response with an error message.
5. **Find User**: If the token is valid, it uses the decoded token to find the corresponding user in the database. It excludes the password field when retrieving the user document.
6. **Check User**: If no user is found, it sends a 404 status code response with an error message.
7. **Attach User to Request**: If a user is found, it attaches the user document to the request object. This allows subsequent middleware functions and route handlers to access the authenticated user’s data.
8. **Call Next Middleware**: Finally, it calls the next function to pass control to the next middleware function in the stack.
9. **Error Handling**: If any error occurs during this process, it logs the error message and sends a 500 status code response with an error message.

This function is a part of a larger Express.js application and should be used in conjunction with other middleware and routing logic. It’s also important to note that this function assumes that the User model has been defined elsewhere in your application. Happy coding! 😊

Sure, let’s delve into point 8:

1. **Call Next Middleware**: Finally, it calls the next function to pass control to the next middleware function in the stack.

In Express.js, middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application’s request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

When a middleware function is done and it needs to pass control to the next middleware function, it calls next(). This is exactly what’s happening in this line of code. After the protectRoute middleware function has completed its task (i.e., verifying the user’s JWT and attaching the user to the request object), it calls next() to pass control to the next middleware function.

This could be another middleware function that performs some operation on the request or response objects, or it could be the route handler itself. The order of middleware functions defined in your Express.js application matters, as they are executed sequentially.

In summary, the next() function is used to pass control from one middleware function to the next. This is a fundamental part of how middleware works in Express.js. Happy coding! 😊

1. [backend](https://github.com/burakorkmez/mern-chat-app/tree/master/backend)
2. /[socket](https://github.com/burakorkmez/mern-chat-app/tree/master/backend/socket)

/

**socket.js**

This code sets up a real-time communication server using Socket.IO, Express.js, and Node.js. Here’s a breakdown of what the code does:

1. **Import Modules**: The necessary modules are imported. This includes socket.io for real-time communication, http for creating the HTTP server, and express for handling HTTP requests.
2. **Create Express App**: An Express application is created.
3. **Create HTTP Server**: An HTTP server is created using the Express application.
4. **Create Socket.IO Server**: A Socket.IO server is created using the HTTP server. CORS (Cross-Origin Resource Sharing) is configured to allow requests from http://localhost:3000.
5. **User Socket Map**: A userSocketMap object is created to map user IDs to their corresponding socket IDs. This allows the server to send messages to specific users.
6. **Socket Connection Event**: An event listener is set up for the connection event. This event is emitted whenever a new client connects to the Socket.IO server.
7. **User Connection**: When a user connects, their user ID is retrieved from the handshake query and their socket ID is stored in the userSocketMap. The server then emits an event to all connected clients with the list of online users.
8. **User Disconnection**: An event listener is set up for the disconnect event. This event is emitted when a client disconnects from the Socket.IO server. When a user disconnects, their entry is removed from the userSocketMap and the server emits an event to all connected clients with the updated list of online users.
9. **Export Modules**: The app, io, and server objects are exported for use in other modules.

This setup allows for real-time communication between the server and clients. Clients can send and receive real-time updates, which is useful for features like chat messaging, live updates, and more. Happy coding! 😊

Sure, let’s delve into each part of the code:

1. **Import Modules**: The necessary modules are imported. This includes socket.io for real-time communication, http for creating the HTTP server, and express for handling HTTP requests.

import { Server } from "socket.io";

import http from "http";

import express from "express";

1. **Create Express App**: An Express application is created. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

const app = express();

1. **Create HTTP Server**: An HTTP server is created using the Express application. This server is required to serve your application and to provide a listener for Socket.IO.

const server = http.createServer(app);

1. **Create Socket.IO Server**: A Socket.IO server is created using the HTTP server. CORS (Cross-Origin Resource Sharing) is configured to allow requests from http://localhost:3000. Socket.IO enables real-time bidirectional event-based communication.

const io = new Server(server, {

cors: {

origin: ["http://localhost:3000"],

methods: ["GET", "POST"],

},

});

1. **User Socket Map**: A userSocketMap object is created to map user IDs to their corresponding socket IDs. This allows the server to send messages to specific users.

const userSocketMap = {}; // {userId: socketId}

1. **Socket Connection Event**: An event listener is set up for the connection event. This event is emitted whenever a new client connects to the Socket.IO server.
2. **User Connection**: When a user connects, their user ID is retrieved from the handshake query and their socket ID is stored in the userSocketMap. The server then emits an event to all connected clients with the list of online users.
3. **User Disconnection**: An event listener is set up for the disconnect event. This event is emitted when a client disconnects from the Socket.IO server. When a user disconnects, their entry is removed from the userSocketMap and the server emits an event to all connected clients with the updated list of online users.
4. **Export Modules**: The app, io, and server objects are exported for use in other modules. This allows other parts of your application to use these instances.

This setup allows for real-time communication between the server and clients. Clients can send and receive real-time updates, which is useful for features like chat messaging, live updates, and more. Happy coding! 😊