Authentication: When you register and login

Authorization: you can perform certain operations only when user is logged in

In the case of **multer({ storage })**, **storage** is an object that has already been defined elsewhere in the code (in this case, the **diskStorage** configuration). When passed within curly braces as **{ storage }**, it's equivalent to **{ storage: storage }**. This syntax is shorthand for creating an object with a property named **storage** and assigning it the value of the variable **storage**.

CONTROLLERS

In the context of Node.js and web development, a controller is a module or function responsible for handling the logic of a particular route or endpoint in a web application. Controllers are commonly used in MVC (Model-View-Controller) architecture or similar design patterns to separate concerns and organize codebase.

Here's how controllers typically fit into the MVC architecture:

1. **Model**: Represents the data of the application, including data manipulation logic and interaction with the database.
2. **View**: Represents the presentation layer of the application, responsible for rendering the user interface.
3. **Controller**: Acts as an intermediary between the model and view layers. It handles incoming requests from clients, interacts with the model to retrieve or manipulate data, and then passes the processed data to the view for rendering. Controllers also handle responses sent back to clients.

**Module or Function**: A controller can be implemented as a standalone module (a separate file) or as a function within a larger file. It encapsulates the logic related to handling HTTP requests and producing HTTP responses for a specific route.

app.post("/auth/register/", upload.single("picture"), register);

* **app.post("/auth/register/", ...)** defines a route handler for HTTP POST requests to the "/auth/register/" endpoint. This means that when a client makes a POST request to this endpoint, Express will invoke the specified middleware and route handler to process the request.
* **upload.single("picture")**: This is a middleware function that is used to handle file uploads. It's typically provided by a library like Multer in Node.js/Express applications. **upload.single("picture")** specifically configures Multer to handle a single file upload with the field name "picture". When a client sends a multipart/form-data POST request to the "/auth/register/" endpoint, this middleware will intercept the request, parse the uploaded file (if any) from the request body, and store it in a temporary location on the server.
* **register**: This is the route handler function that will be called to handle the incoming request. When the middleware (in this case, **upload.single("picture")**) finishes processing the request, it will pass control to the **register** route handler. This handler is responsible for further processing the request, such as validating form data, performing database operations, and sending a response back to the client.

**bcrypt**:

* **import bcrypt from "bcrypt";**
* This statement imports the bcrypt module, which is a popular library used for hashing passwords securely in Node.js applications.
* By importing bcrypt, you can use its functions to hash passwords, compare hashed passwords, and generate salted hashes, among other cryptographic operations.
* It's common practice to use bcrypt for securely storing user passwords in databases to prevent unauthorized access.

JSON WEBTOKENS

JSON Web Tokens (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JWTs are commonly used for authentication and information exchange in web applications.

Here are the key components of a JWT:

1. **Header**: Contains metadata about the type of token and the cryptographic algorithm used to secure the token.
2. **Payload**: Contains the claims or assertions made by the token, such as user identity, permissions, and additional metadata. Claims are usually classified into three types:
   * Registered claims: Pre-defined claims recommended to be used.
   * Public claims: Custom claims defined by users, but they must be registered in the IANA JSON Web Token Registry or be defined as a URI.
   * Private claims: Custom claims defined by users without registering or URL-related constraints.
3. **Signature**: Used to verify the integrity of the token and ensure it hasn't been tampered with. It's generated by signing the encoded header, encoded payload, and a secret key using a cryptographic algorithm specified in the header.
4. Authentication: JWTs can be used as authentication tokens to verify the identity of users and grant access to protected resources.
5. Authorization: JWTs can carry user permissions and roles, allowing servers to make authorization decisions based on the claims within the token.
6. Information exchange: JWTs can carry any arbitrary data, making them suitable for exchanging information between parties in a secure and tamper-proof manner
7. So, when a user logs in or performs some action that requires authentication, the server creates a JWT containing the user's information. This JWT is then sent to the user's browser or device.
8. Now, whenever the user wants to access a protected area of the website or perform an action, they present their JWT. The server checks the JWT's signature to ensure it's valid and hasn't been tampered with. If everything checks out, the user is granted access.
9. JWTs are handy because they're compact, self-contained, and don't require the server to store session information. They're commonly used for authentication and authorization in web applications.

const UserSchema=new mongoose.Schema({

    firstName: {

        type: String,

        require: true,

        min: 2,

        max: 50,

    }

})

Yes, the properties like **type**, **required**, **min**, and **max** are indeed predefined in Mongoose's **Schema** class. They are part of the Mongoose schema definition syntax and are used to define the structure and validation rules for your MongoDB documents.

CREATIONG OF A MODEL

const User = mongoose.model("User", UserSchema);

* **mongoose.model**: This is a method provided by Mongoose for creating models. It takes two arguments:
  1. The first argument is the name of the model. In this case, it's **"User"**.
  2. The second argument is the schema to be used for the model. Here, it's **UserSchema**.
* **User**: This is the name of the model that you're defining. In Mongoose, models are constructors compiled from schemas. Once you've created a model, you can use it to create, read, update, and delete documents in the MongoDB database.

So, in summary, the line **const User = mongoose.model("User", UserSchema);** creates a Mongoose model named **User**, which is based on the **UserSchema** schema definition. This model can then be used to interact with the MongoDB database collection associated with the **"User"** model, allowing you to perform various operations such as creating new users, querying existing users, updating user data, and so on.

IMP NOTE: WE USE AWAIT IN CASE OF ASYNCHRONOUS FUNCTION CALLS LIKE CALLS TO DATABASE OR OTHER PACKAGES THAT TAKE PLACE ASYNCHRONOUSLY.

REGISTER FUNCTION

User gives us the password, we encrypt and save it. When the user logs in again we hash the password and check if theyre the same then we give them json web token.

1. **User Registration**:
   * When a user registers, they provide a password.
   * This password needs to be securely stored in the database. Instead of storing the password as plain text, it's hashed using a hashing algorithm like bcrypt before storing it in the database.
   * The hashed password is then associated with the user's account in the database.
2. **User Login**:
   * When a user tries to log in, they provide their username/email and password.
   * The server retrieves the hashed password associated with the user's account from the database.
   * The server then hashes the password provided by the user using the same hashing algorithm (bcrypt) and compares it with the hashed password retrieved from the database.
   * If the hashed passwords match, it means the user provided the correct password, and they are granted access.
3. **Issuing JSON Web Token (JWT)**:
   * Once the user's credentials are verified, the server can generate a JSON Web Token (JWT) to provide secure access to certain resources or endpoints.
   * The JWT typically contains claims (such as user ID, username, or other user-related information) and is digitally signed using a secret key known only to the server.
   * This token is then sent back to the client (usually in the response body or headers) and can be included in subsequent requests to access protected resources.
   * On the server side, incoming requests with JWTs are verified to ensure their authenticity and validity before granting access to the requested resources.

STATUS CODES HTTP

In the context of HTTP status codes, the status code **201 Created** indicates that a new resource has been successfully created as a result of the request. It's commonly used in the context of POST requests, where the client sends data to the server to create a new resource, such as adding a new user, posting a new comment, or creating a new record in a database.

When the server receives a POST request and successfully creates the requested resource, it responds with a **201 Created** status code to indicate that the operation was successful. This status code serves as confirmation to the client that their request to create a new resource has been fulfilled.

So, when you see **res.status(201)** in a POST request handler in an Express.js application, it's typically used to send back a response to the client indicating that the creation of a new resource was successful. It's a way of communicating to the client that their request has been processed and the new resource has been successfully added to the system.

PURPOSE OF THE CODE SO FAR:

The provided code is an Express.js route handler function for handling user registration. Let's break down the purpose of each section:

1. **Extracting Data from Request Body**:
   * The function starts by extracting various user registration data from the **req.body**, including **firstName**, **lastName**, **email**, **password**, **picturePath**, **friends**, **location**, and **occupation**.
2. **Generating Salt and Hashing Password**:
   * Next, the code generates a salt using **bcrypt.genSalt()** to hash the password securely. The salt is generated asynchronously using **await** to ensure that the code waits for the salt generation to complete before proceeding.
   * Once the salt is generated, the password provided by the user (**password**) is hashed using **bcrypt.hash()** along with the generated salt.
   * The hashed password is stored in the variable **password\_hash**.
3. **Creating User Object and Saving to Database**:
   * A new instance of the **User** model (presumably defined elsewhere in the code) is created with the extracted user data and the hashed password (**password\_hash**).
   * Additional attributes such as **viewedProfile** and **impressions** are randomly generated and added to the user object.
   * The **newUser** object is then saved to the database using **newUser.save()**, and the result (saved user object) is stored in **saved\_user**.
4. **Sending Response**:
   * Finally, a **201 Created** status response is sent back to the client along with the saved user object in JSON format (**res.status(201).json(saved\_user)**). This informs the client that the user registration was successful and provides them with the details of the newly created user.

Overall, the purpose of this route handler function is to handle user registration by securely hashing the password, saving the user data to the database, and sending a response back to the client indicating the success of the registration process.

ROUTES -> auth.js

**Creating a Router Instance**:

* **const router = express.Router();**: This line creates a new router instance using the **express.Router()** method. This router instance is used to define routes and middleware specific to this router.

Using a router instance provides several benefits over defining routes directly on the **app** object:

1. **Modularity and Organization**: By using routers, you can modularize your route definitions into separate files or modules. This helps organize your codebase, especially as your application grows and you have a large number of routes. Each router can handle routes for a specific feature or area of your application, making it easier to maintain and understand.
2. **Middleware Isolation**: Routers allow you to define middleware specific to a set of routes. Middleware defined on a router will only apply to routes defined on that router, providing better isolation and encapsulation. This helps prevent middleware intended for one set of routes from affecting other routes in your application.
3. **Route Prefixing**: Routers allow you to define a base path for all routes defined on that router. This allows you to group related routes under a common base path, reducing repetition and making your route definitions more concise and readable.
4. **Scalability**: Using routers makes it easier to scale your application by breaking it into smaller, manageable pieces. As your application grows, you can add more routers to handle new features or endpoints without cluttering the main **app** object with too many route definitions.

BCRYPT SALT

In the context of bcrypt, a salt is a random value that is combined with the password before hashing. Salting is a technique used to strengthen password security by adding randomness to the hashed output, making it more resistant to various cryptographic attacks, such as rainbow table attacks.

Here's how salting works in bcrypt:

1. **Generate a Salt**: When hashing a password using bcrypt, a random salt is generated. This salt is typically a random sequence of bytes.
2. **Combine with Password**: The salt is then combined with the password before hashing. This combination ensures that even if two users have the same password, their hashed passwords will be different because they have different salts.
3. **Hashing**: The combined salt and password are then passed to the bcrypt hashing algorithm, which generates a cryptographic hash of the combined value.
4. **Storing the Hashed Password and Salt**: The resulting hash, along with the salt used, is typically stored in a database. This allows the system to verify the user's password during login by repeating the hashing process with the stored salt and comparing the resulting hash with the stored hash.

By salting passwords before hashing them, bcrypt helps mitigate common password attacks, such as dictionary attacks and rainbow table attacks, making it more difficult for attackers to reverse-engineer passwords from their hashed representations.

JWT VERIFICATION

export const verifyToken=async (req,res,next) =>

{

    try{

        let token=req.header("Authroization");//takes token from this header in frontend?

        if(!token)

            return res.status(403).send("Access denied");//we use send when we return

        //403 means client does not have access rights to the content, 401 means unauthenticated(user must authenticate themselves),400 is client error

        if(token.startsWith("Bearer ")){

            token=token.slice(7,token.length).trimLeft();

        }

        const verified=jwt.verify(token,process.env.JWT\_SECRET);

        req.user=verified;

        next();

    }

    catch(err){

        res.status(500).json({error: err.message});

    }

};

1. **req**: This is the request object. It contains information about the HTTP request that comes from the client. It includes data like headers, parameters, body, etc.
2. **req.header("Authorization")**: This line is trying to retrieve the value of the "Authorization" header from the HTTP request. In many cases, tokens are passed in the "Authorization" header, typically in the format "Bearer [token]".
3. **token**: This variable holds the value of the token extracted from the "Authorization" header.
4. **res**: This is the response object. It's used to send a response back to the client.
5. **res.status(403).send("Access denied")**: If there's no token provided (meaning **token** is falsy), the server responds with a status code of 403 (Forbidden) and sends the message "Access denied" to the client.
6. **jwt.verify(token, process.env.JWT\_SECRET)**: This line attempts to verify the token using a secret key stored in the environment variable **JWT\_SECRET**. It's assumed that **jwt** refers to a library like **jsonwebtoken**.
7. **req.user = verified**: If the token is successfully verified, its decoded payload (usually containing user information) is attached to the **req** object as **req.user**. This allows subsequent middleware or route handlers to access the user information.
8. **next()**: This function is called to pass control to the next middleware function in the stack. It's typically used in middleware to pass control to the next handler after it has completed its work.
9. **res.status(500).json({error: err.message})**: If there's an error during token verification or any other part of the process, the server responds with a status code of 500 (Internal Server Error) and sends a JSON object with the error message.

FOR JWT.VERIFY(DIGITAL SIGNATURE):

Yes, that's essentially how digital signatures work in the context of JWTs. Let's break it down a bit further:

1. **Payload**: The payload of a JWT is typically a JSON object containing some data you want to transmit. This could be user information, permissions, or any other data relevant to your application.(payload is user id)
2. **Header**: Along with the payload, a JWT also contains a header that describes the token and the signing algorithm being used. This header is also encoded in JSON format.
3. **Signature**: To create the signature, the JWT library takes the encoded header, the encoded payload, and a secret key. It then applies a cryptographic hash function (such as HMAC-SHA256) to these inputs, along with the secret key. This produces a unique hash, which is considered the digital signature of the token.
4. **Encoding**: Finally, the encoded header, encoded payload, and digital signature are concatenated together, separated by periods, to form the complete JWT.

So, when you verify a JWT:

* The library decodes the JWT to extract the header, payload, and signature.
* It then recalculates the signature using the same algorithm, the header, the payload, and the secret key.
* If the recalculated signature matches the signature in the JWT, it means the token is valid and hasn't been tampered with.

So, in essence, the secret key is used to generate and verify the digital signature of the JWT, ensuring its integrity and authenticity.

GETTING USER AND USER FRIENDS FROM ROUTE:

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

/\*READ\*/

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

    try{

        const {id}=req.params;

        const user=await User.findByID(id);

        res.status(200).json(user);

    }

    catch(err){

        res.status(404).json({message: err.message});

    }

}

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

        const {id}=req.params;

        const user=await User.findByID(id);

        const friends= await Promise.all(user.friends.map((id)=>User.findByid(id)));

        //Promise.all() takes an array of promises and returns a single promise when all of the constituent promises have resolved

        //friends is an array, Promise.all() returns array of objects

        //map() iterates over allof the ids, user.friends is an array of objects

}

The **.map()** method in JavaScript is used to iterate over an array and apply a function to each element, creating a new array with the results of calling the function on each element.