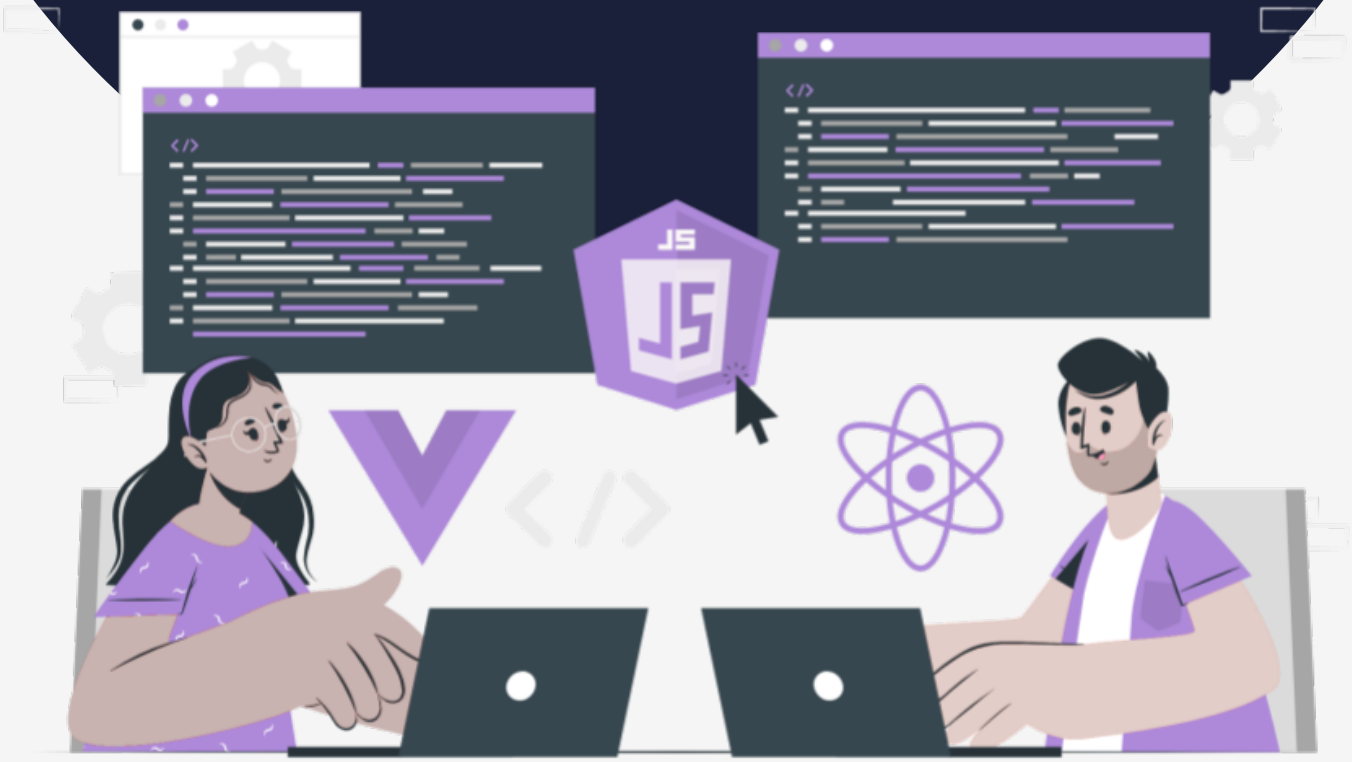
Lesson:

Create a simple login API for SignUp and SignIn



PW SKILLS

Topics

- Setup node.js server
- Connect the application to the MongoDb
- Create user schema
- JWT (router level Middleware)
- create Controller
- create Routes

The following are some requirements for developing these APIs. First of all, you need to have Node installed on your system, as well as MongoDB since we are using MongoDB as our database. So first of all we will learn what is basically node, express, and MongoDB.

Node.js: Node.js is a platform built on **Chrome's JavaScript runtime** for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.
Express: Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node-based Web applications.
MongoDB: MongoDB is a cross-platform, document-oriented database that provides high performance, high availability, and easy scalability. MongoDB works on the concept of collection and documentation.

Now, let's set-up our application

Open any of your editor like visual studio code |sublime text or any other.

In the terminal write

```
Unset
npm init
```

Then the app will be initialized, and all basic dependencies will be installed. Here, you need to pass a few details if you wish, otherwise, there is no need to do so. The command prompt will look like the following.

```
Unset
package name: (jwtauth)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to F:\jwtAuth\package.json:
{
  "name": "jwtauth",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
Is this OK? (yes)
```

Now install some dependencies that will require in developing our api's like express, mongoose, cookie-parser, bcrypt, dotenv. So in the terminal write

```
Unset
npm install express cookie-parser bcrypt mongoose jsonwebtoken
cors email-validator dotenv
```

Use of each module.

1. **express :** Allows you to define routes of your application based on HTTP methods and URLs.
2. **cookie-parser:** It is used for parsing the cookies
3. **bcrypt:** It is used for hashing and comparing the passwords.
4. **mongoose:** It is used to connect to our MongoDB database.
5. **jwtwebtoken:** JSON Web Token (JWT) is an open standard that defines a compact and self-contained way of securely transmitting information between parties as a JSON object.
6. **cors:** It is a middleware that enables Cross-Origin Resource Sharing (CORS) in a Node.js application.
7. **email-validator:** It is a lightweight library that validates email addresses
8. **dotenv:** it's loads environment variables from a .env file in a Node.js application.

Let's now define our app directory, i.e. the name of all the folders and files within the directory. Our main file will be index.js, which contains information about how the app is being listened to.

```
Unset
backend
     config
       - databaseConfig.js
     controller
       - authController.js
     middleware
       - jwtAuth.js
     model
       - userSchema.js
     node_modules
     router
       authRouth.js
     .env
     app.js
     index.js
     package-lock.json
     package.json
```

Here, config/databaseConfig.js will contain the database connection, middleware/jwtAuth.js will contain code for finding the user token, userSchema.js will have user schema, controller/authController contains all logic part, router/authRoute.js have all our routes and finally, our index.js will have imported dependencies

Lets first mention all the required environment variables in .env file

```
Unset
PORT= 8081
MONGODB_URL= mongodb://127.0.0.1:27017/<database
name>
SECRET=SECRET
CLIENT_URL = http://localhost:3000
```

Now Create one file in the root directory called app.js and set up the basic express server with middleware.

```JavaScript
const express = require("express");
const app = express();

const cookieParser = require("cookie-parser");
const cors = require("cors");

//middleware
app.use(express.json()); //built-in middleware
app.use(cookieParser()); //Third-party middleware
app.use(cors({ origin: process.env.CLIENT_URL, credentials:
true })); // Third-party middleware

app.use("/", (req, res) => {
  res.status(200).json({ data: "JWTauth server ;)" });
});

module.exports = app;
```

With the help of **app.use()** we can mount middleware functions.
app.use(express.json()) is used to parse incoming JSON requests,
app.use(cookieParser()) is used to parse incoming cookies, and
app.use(cors()) allows the application to receive requests from a different domain or port. It takes an options object as an argument, in this case, we are passing **origin** and **credentials.**

- **credentials:** Takes a boolean value indicating whether to include cookies and authorization headers in cross-origin requests.
- **Origin:** Pass the single origin or multiple origins in the array, so requests from the specified origin will be allowed. The origin option can also be set to **'true'**, which will allow requests from any origin, or **false**, which will disallow all requests from different origin.

app.use("/" , ()=>{}) defines a middleware that responds to all incoming requests at the root URL ("/")

Now create an index.js file in the root directory

```JavaScript
require("dotenv").config();
const PORT = process.env.PORT;

const app = require("./app.js");

app.listen(PORT, () => {
  console.log(`
server is listening at http://localhost:${PORT}`);
});
```

In this file,
First, we load environment variables, retrieve a specified **PORT** value from the .env, import an Express application from app.js file, and start the server listening on the specified **PORT.**

Connect the application to the MongoDb

In the root directory, create a folder called config, and create a file databaseConfig.js within that folder.

```JavaScript
const mongoose = require("mongoose");
const MONGODB_URL = process.env.MONGODB_URL;

const connectDatabase= () => {
  mongoose
    .connect(MONGODB_URL)
    .then(() => console.log("connected to DB"))
    .catch((err) => console.log(err.message));
};

module.exports = connectDatabase;
```

The above code exports a function that establishes a connection to a MongoDB database using Mongoose, with the database URL retrieved from environment variables.

Now in the app.js file, connect the MongoDB database by calling the **connectDatabase()** function imported from a config/databaseConfig.js file.

The app.js file will look like this.

```JavaScript
const express = require("express");
const app = express();

const cookieParser = require("cookie-parser");
const cors = require("cors");
//imported the connectDatabase form ./config/databaseConfig.js
const connectDatabase = require("./config/databaseConfig.js");


// connect to db
connectDatabase();

app.use(express.json());
app.use(cookieParser());
app.use(cors({ origin: [process.env.CLIENT_URL], credentials:
true }));

app.use("/", (req, res) => {
  res.status(200).json({ data: "JWTauth server ;)" });
});

module.exports = app;
```

Create user schema

Create a folder called model in the root directory and in the model folder create a file called userSchema.js.

Let's define userSchema which can be used to create, update, and query user documents in a MongoDB database.

```JavaScript
const mongoose = require("mongoose");
const { Schema } = mongoose;
const userSchema = new Schema(
  {
    name: {
      type: String,
      require: [true, "user name is Required"],
      minLength: [5, "Name must be at least 5 characters"],
      maxLength: [50, "Name must be less than 50 characters"],


      trim: true
    },
    email: {
      type: String,
      required: [true, "user email is required"],
      unique: true,
      lowercase: true,
      unique: [true, "already registered"]
},
password: {
      type: String,
      select: false
    },
forgotPasswordToken: { type: String },
forgotPasswordExpiryDate: { type: Date }
 },
```

```
  { timestamps: true }
);

// Hashes password before saving to the database
userSchema.pre("save", async function (next) {
  // If password is not modified then do not hash it
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  return next();
});

userSchema.methods = {
jwtToken() {
    return JWT.sign(
      { id: this._id, email: this.email },
      process.env.SECRET,
      { expiresIn: "24h" } // 24 hours

    );
  },
 //userSchema method for generating and return forgotPassword
token
 getForgotPasswordToken() {
```

```
    const forgotToken =
crypto.randomBytes(20).toString("hex");
    //step 1 - save to DB
    this.forgotPasswordToken = crypto
       .createHash("sha256")
       .update(forgotToken)
       .digest("hex");

    /// forgot password expiry date
    this.forgotPasswordExpiryDate = Date.now() + 20 * 60 *
1000; // 20min

    //step 2 - return values to user
    return forgotToken;
 }
}
```

- **name:** A required string field that must be at least 5 characters long and no longer than 50 characters. The "trim" option removes any whitespace from the beginning and end of the string.
- **email:** A required unique string field that must be in lowercase. If a user with the same email already exists, an error will be thrown.
- **password:** A string field that is not selectable (i.e. it will not be returned in queries by default).
- **forgotPasswordToken:** A string field that will be used to store a token for resetting the user's password.
- **forgotPasswordExpiryDate:** A date field that will be used to store the expiry date of the reset password token.

The **"timestamps"** option is set to true, which means that Mongoose will automatically add **"createdAt"** and **"updatedAt"** fields to the documents in this collection.

In user Schema we also defined **pre-middleware function** (executed before saving a user document to the database) and **schema methods**

JavaScript

```
// Hashes password before saving to the database
userSchema.pre("save", async function (next) {
  // If password is not modified then do not hash it
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  return next();
});
```

The **pre-middleware function** checks whether the user's password has been modified or not. If the password has not been modified, then the function returns and does nothing. If the password has been modified, then the function uses the npm package **bcrypt** to hash the password before saving it to the database.

```JavaScript
userSchema.methods = {
jwtToken() {
    return JWT.sign(
      { id: this._id, email: this.email },
      process.env.SECRET,
      { expiresIn: 24 * 60 * 60 * 1000 } //24
    );
  },

getForgotPasswordToken() {
    const forgotToken =
crypto.randomBytes(20).toString("hex");
    //step 1 - save to DB
    this.forgotPasswordToken = crypto
       .createHash("sha256")
       .update(forgotToken)
       .digest("hex");

    /// forgot password expiry date
    this.forgotPasswordExpiryDate = Date.now() + 20 * 60 *
1000; // 20min

    //step 2 - return values to user
    return forgotToken;
 }
}
```

This **schema method JwtToken** generates a JSON Web Token (JWT) by signing an object containing the user's ID and email with a secret key stored in the **process.env.SECRET** environment variable. The token expires after 24 hours (specified in milliseconds) from the time it was created. This method is likely used to authenticate and authorize users in a web application.
The second method **getForgotPasswordToken()** generates a forgot password token by creating a random token, hashing it, and saving the hashed value and expiry date to the database. The unhashed token is returned to the user.

JWT (router level Middleware)

Create a folder called middleware and in the middleware folder create a jwtAuth.js file.

This middleware function called **jwtAuth** is used to authenticate and authorize users in a web application using JSON Web Token (JWT)

```JavaScript
const JWT = require("jsonwebtoken");

const jwtAuth = (req, res, next) => {
  const token = (req.cookies && req.cookies.token) || null;

  if (!token) {
    return res.status(400).json({ success: false, message:
"NOT authorized" });
  }

  try {
// verify JWT token
    const payload = JWT.verify(token, process.env.SECRET);
    req.user = { id: payload.id, email: payload.email };
  } catch (error) {
    return res.status(400).json({ success: false, message:
error.message });
  }

  next();
};

module.exports = jwtAuth;
```

This middleware function first retrieves the JWT token from a cookie in the HTTP request, and if there is no token, it returns an error response indicating that the user is not authorized.

If a token is present, the function then verifies the token using a secret key stored in the **process.env.SECRET** environment variable. If the verification fails, it returns an error response with the error message. Otherwise, the function adds the user's ID and email from the token payload to the request object and passes control to the next middleware in the chain using **next()**. Finally, the function is exported for use in Router-level middleware.

Create Controller

Create a Controller folder in the root directory and in the controller folder create authController.js file

signUp controller function
This function will handle user registration and validation logic.

```javascript
const emailValidator = require("email-validator");
const signUp = async (req, res, next) => {
  const { name, email, password, confirmPassword} = req.body;

  /// every field is required
  if (!name || !email || !password || ! confirmPassword) {
    return res.status(400).json({
      success: false,
      message: "Every field is required"
    });
  }
// validate email using npm package "email-validator"
  const validEmail = emailValidator.validate(email);
  if (!validEmail) {
    return res.status(400).json({
      success: false,
      message: "Please provide a valid email address"
    });
  }

  try {
    /// send password not match err if password !==
confirmPassword
    if (password !== confirmPassword) {
      return res.status(400).json({
        success: false,
        message: "password and confirm should be same"
      });
    }
```

```javascript
// userSchema "pre" middleware function for "save" will hash
the password using "bcrypt" (npm package) before saving the
data into the database
    const userInfo = new userModel(req.body);
    const result = await userInfo.save();
    return res.status(200).json({
      success: true,
      data: result
    });
  } catch (error) {
    console.log(error);
    return res.status(400).json({
      message: error.message
    });
  }
};
```

For signUp, The function first extracts the **name, email, password**, and **confirmPassword** fields from the request body. It then checks if all fields are present, and if any of them are missing, it returns an error response with a message indicating that every field is required.

The function then validates the email address provided by the user using an **email-validator** (npm package) and returns an error response if the email is invalid.

Next, the function checks if the **password** and **confirmPassword** fields match. If they do not match, it returns an error response indicating that the passwords must match.

If all validation checks pass, the function creates a new **userModel** instance with the request body **(req.body)**, saves it to the database using the **save** method, and returns a success response with the saved user data. If any errors occur during the process, the function returns an error response with the error message.

SignIn controller function

This function is used to handle user authentication logic and sets a JWT cookie in the HTTP response to allow the user to access the protected route in the application.

```JavaScript
const signIn = async (req, res, next) => {
  const { email, password } = req.body;

// return response with an error message if the email of
password is missing
  if (!email || !password) {
    return res.status(400).json({
      success: false,
      message: "email and password are required"
    });
  }

  try {
    // check user exist or not
    const user = await userModel
      .findOne({
        email
      })
      .select("+password");

// if the user is null or the password is incorrect return
response with an error message
    if (!user || !(await bcrypt.compare(password,
user.password))) {

      return res.status(400).json({
        success: false,
        message: "invalid credentials"
      });

    }

// create the JWT token using the userSchema method
(jwtToken())
const token = user.jwtToken();
    user.password = undefined;

    const cookieOption = {
      maxAge: 24 * 60 * 60 * 1000, //24hr
      httpOnly: true
    };

// return a response with user object and cookie (contains jwt
Token)
    res.cookie("token", token, cookieOption);
    res.status(200).json({
      success: true,
      data: user
    });
  } catch (error) {
    return res.status(400).json({
      success: false,
      message: error.message
    });
  }
};
```

The signIn function first extracts the **email** and **password** fields from the request body, and checks if both fields are present. If either of them is missing, the function returns an error response indicating that both fields are required.

The function then looks up the user in the database using the **findOne()**, and selects the password field using the **select** method. It then checks if the password provided by the user matches the hashed password in the database using the **bcrypt.compare** method. If either the user is not found or the password is incorrect, the function returns an error response indicating that the credentials are invalid.

If the user is found and the password is correct, the function generates a JSON Web Token (JWT) using the **jwtToken** then we set the **password** field of the user object to **undefined** because a password should not be visible to the user, and set a cookie named "**token**" with the JWT in the HTTP response using the **res.cookie** method. Finally, the function returns a successful response with the user data.

getUser controller function
This function retrieves user information based on the authenticated user ID present in the request(set by router level middleware function - jwtAuth).

```javascript
const getUser = async (req, res, next) => {
  const userId = req.user.id;
  try {
    const user = await userModel.findById(userId);
    return res.status(200).json({ success: true, data: user
});
  } catch (error) {
    return res.status(400).json({ success: false, message:
error.message });
  }
};
```

The getUser function first extracts the user ID from the request which we set during the authentication in the JWT middleware function (middleware/jwtAuth.js) and then attempts to find the corresponding user using **userModel.findById.** If successful, returns a success response with the user data.

Forgot Password controller function

```javascript
const forgotPassword = async (req, res, next) => {
   const email = req.body.email;

   // return response with error message If email is undefined
   if (!email) {
      return res.status(400).json({
         success: false,
         message: "Email is required"
      });
   }

   try {
      // retrieve user using given email.
      const user = await userModel.findOne({
         email
      });

      // return response with error message user not found
      if (!user) {
         return res.status(400).json({
            success: false,
            message: "user not found"
         });
      }

      // Generate the token with the userSchema method
getForgotPasswordToken().
    const forgotPasswordToken = user.getForgotPasswordToken();

      await user.save();
      return res.status(200).json({
         success: true,
         token: forgotPasswordToken
      });
   } catch (error) {
```

```
      return res.status(400).json({
         success: false,
         message: error.message
      });
   }
};
```

The **forgotPassword** function is used to reset a user's password. First, it checks if the email property is defined in the request body. If it is not defined, the function sends back an error message with a status code of 400, indicating that email is required.

If email is defined, the function searches for a user with the given email in the database using the **findOne** method from the **userModel**. If no user is found, the function sends back an error message with a status code of 400, indicating that the user was not found.

If a user is found with the given email, the function generates a password reset token using the **getForgotPasswordToken** method defined on the **userSchema**. The token is then saved in a database and returned in the response along with a 200 status code and a success message.

If an error occurs at any point during the execution of the function, a response with a 400 status code and the error message is returned.

Reset password controller function

This function is used to handle a request to reset a user's password, by finding the user associated with the given token, verifying the token is valid and not expired, updating the user's password, and returning the updated user data in a response.

```JavaScript
const resetPassword = async (req, res, next) => {
  const { token } = req.params;
  const { password, confirmPassword } = req.body;

  // return error message if password or confirmPassword is
missing
  if (!password || !confirmPassword) {
    return res.status(400).json({
      success: false,
      message: "password and confirmPassword is required"
    });
  }

  // return error message if password and confirmPassword are
not same
  if (password !== confirmPassword) {
    return res.status(400).json({
      success: false,
      message: "password and confirmPassword does not match"
    });
  }

  //  Hash the "token" using crypto for retrieving the user
from the database using the this hash token as a
forgotPasswordToken
  const hashToken =
crypto.createHash("sha256").update(token).digest("hex");

  try {
```

```
    const user = await userModel.findOne({
      forgotPasswordToken: hashToken,
      forgotPasswordExpiryDate: {
        $gt: new Date() // forgotPasswordExpiryDate() less
then current date
      }
    });

    // return the message if
    if (!user) {
      return res.status(400).json({
        success: false,
        message: "Invalid Token or token is expired"
      });
```

```
    }

    user.password = password;
    await user.save();

    user.forgotPasswordExpiryDate = undefined;
    user.forgotPasswordToken = undefined;
    user.password = undefined;

    return res.status(200).json({
      success: true,
      data: user
    });
  } catch (error) {
    return res.status(400).json({
      success: false,
      message: error.message
    });
  }
};
```

The **resetPassword** function first extracts the **token, password,** and **confirmPassword** properties from the request object. If **password** or **confirmPassword** is undefined, the function returns a response with a message indicating that both fields are required. If **password** and **confirmPassword** do not match, the function returns a response with a message indicating that they don't match.

Next, the function hashes the **token** using the **sha256** algorithm from the **crypto** (npm package). It then attempts to find a user in the database with a matching **forgotPasswordToken** field and a **forgotPasswordExpiryDate** field greater than the current date.

If no matching user is found, the function returns a response with a 400 status code and an error message indicating that the token is invalid or has expired. If a matching user is found, the function updates the user's password and removes the **forgotPasswordToken** and **forgotPasswordExpiryDate** fields. Finally, the function returns a response with a 200 status code and the updated user data.

Logout controller function

```
JavaScript
const logout = async (req, res, next) => {
  try {
    const cookieOption = {
      expires: new Date(),
      httpOnly: true
    };

    res.cookie("token", null, cookieOption);
    res.status(200).json({
      success: true,
      message: "Logged Out"
    });
  } catch (error) {
    res.stats(400).json({ success: false, message:
error.message });
```

```
  }
};
```

This function is responsible for clearing the **token** cookie stored in the client's browser. Set the cookie's value to **null** and its expiration date to the current date, ensuring that the cookie is deleted.

If the operation is successful, the function returns a JSON object with a logged-out message. If an error occurs, the function returns a JSON object with a **success** property set to **false** and an error message.

Create router

Now let's create routers for signin, signup , getUSer, resetPassword, forgotPassword, and logout. first, we have to define authRouter using express.Router()

```javascript
const express = require("express");
const authRouter = express.Router();
const jwtAuth = require("../middleware/jwtAuth.js");

const {
  signUp,
  signIn,
  logout,
  getUser
} = require("../controller/authController.js");

authRouter.post("/signup", signUp);
authRouter.post("/signin", signIn);
authRouter.get("/logout", jwtAuth, logout);

authRouter.get("/user", jwtAuth, getUser);

module.exports = authRouter;
```

Import JWT auth middleware function and all the controller function from controller/authController.js.

Now define routes for signup, signin, logout, forgotPassword, resetPassword, and user(get user detail) by passing the respective controller function.

Export the authRouter and use in app.js

```javascript
// Auth router
app.use("/api/auth", authRouter);
```

Finally app.js looks like this with a database connection, using built-in middleware (express.json()), Third-party middleware (cors(), cookie-parser()), and route with prefix path ("/api/auth").

```javascript
const express = require("express");
const app = express();

const authRouter = require("./router/authRoute.js");
const connectDatabase = require("./config/databaseConfig.js");
const cookieParser = require("cookie-parser");
const cors = require("cors");
// connect to db
connectDatabase();

//middleware

app.use(express.json()) // Built-in middleware
app.use(cookieParser()); // Third-party middleware
app.use(cors({ origin: [process.env.CLIENT_URL], credentials:
true }));  //Third-party middleware

// Auth router
app.use("/api/auth", authRouter);

app.use("/", (req, res) => {
  res.status(200).json({ data: "JWTauth server ;)" });
});

module.exports = app;
```

Finally run the server using the command given below

```
node index.js
```