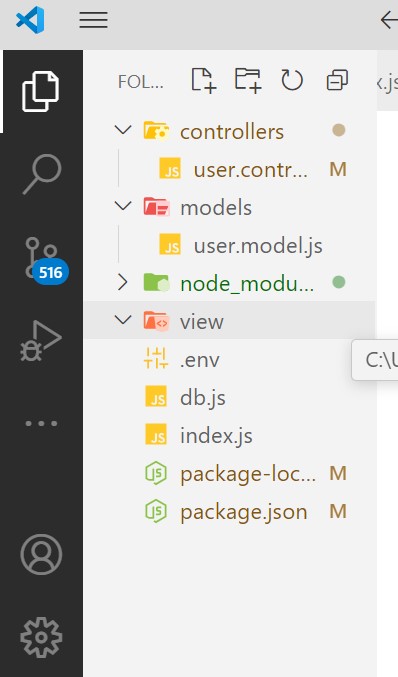
## Page:1

# Authentication Project



## Index.js

const express = require("express");

const connection = require("./db");

const UserRouter = require("./controllers/user.controller");

const dotenv = require("dotenv");

dotenv.config();

const app = express();

app.use(express.json());

## Page:2

app.use("/user", UserRouter);

console.log(process.env.PORT);

app.listen(process.env.PORT, async () => {

  try {

    await connection;

    console.log(`server is running at PORT ${process.env.PORT}`);

    console.log("connected to the db");

  } catch (error) {

    console.log("error");

  }});

## db.js

const mongoose = require("mongoose");

const connection = mongoose.connect("mongodb://127.0.0.1:27017/nupurDB");

module.exports = connection;

## .env

PORT =8080

KEY = ""

# User.model.js

const mongoose = require("mongoose");

//### UserSchema ###

const UserSchema = new mongoose.Schema({

## Page:3

  name: String,

  email: String,

  password: String,

});

// #### Model ###

const UserModel = mongoose.model("user", UserSchema);

module.exports = UserModel;

# usermodel.contoller.js

const express = require("express");

const UserModel = require("../models/user.model");

const UserRouter = express.Router();

const bycrypt = require("bcrypt");

const jwt = require("jsonwebtoken");

const dotenv = require("dotenv");

// ###Registration Routes ###

UserRouter.get("/alluserdata", async (req, res) => {

  try {

    const alluserdata = await UserModel.find();

    res.status(200).json({ userdata: alluserdata });

  } catch (error) {

    res.status(400).json({ message: "Error fetching users" });

  }

});

## Page:4

UserRouter.post("/add", (req, res) => {

  //   console.log(req.body);

  const { name, email, password } = req.body;

  console.log(name, email, password);

  try {

    bycrypt.hash(password, 5, async (err, hash) => {

      if (err) {

        return res.status(500).json({ message: "Error in hashing password" });

      } else {

        const singleuser = new UserModel({ name, email, password: hash });

        await singleuser.save();

        res.status(200).json({ message: "user created successfully" });

      }

    });

  } catch (error) {

    res.status(400).json({ message: error });

  }

});

UserRouter.post("/login", async (req, res) => {

  const { email, password } = req.body;

  try {

    const matchuser = await UserModel.findOne({ email });

    console.log(matchuser);

    const hashpassword = matchuser.password;

    bycrypt.compare(password, hashpassword, function (err, result) {

      if (result) {

        res

## Page:5

.status(200)

          .json({ message: "Login Successfully", user: matchuser });

      } else {

        res.status(400).json({ message: "Invalid Email or Password" });

      }

    });

  } catch (error) {

    res.status(400).json({ message: error });

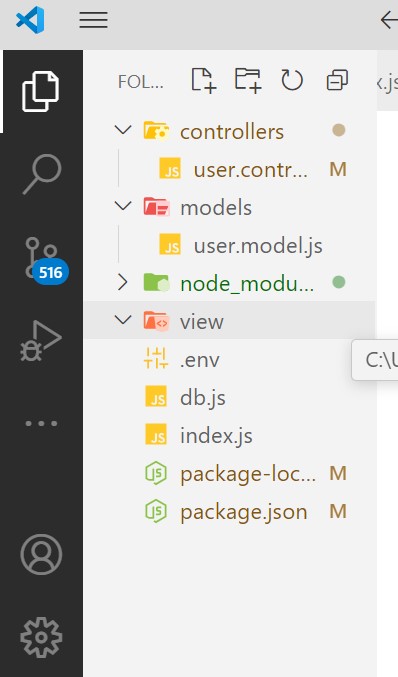
  }

});

module.exports = UserRouter;

## Page:1

# Explanation



## Page:2

This code sets up a simple web server using **Express.js** (a framework in **Node.js**) that can handle user-related actions like registration and login. It also connects to a **database** (most likely **MongoDB**) to store user data.

Let’s break this down step by step in a simple way:

**1. Setting Up the Tools (Modules):**

* **express**: This is the main library that helps us create a web server, which listens for and responds to requests.
* **connection**: This represents the connection to the database (MongoDB). It’s defined elsewhere, probably in a db.js file.
* **UserRouter**: This is where all the user-related routes (like registration or login) are defined.
* **dotenv**: This is a tool that allows us to load environment variables from a .env file, like the port number where the server will run.

js

Copy code

const express = require("express"); // Load the Express framework

const connection = require("./db"); // Load the database connection

const UserRouter = require("./controllers/user.controller"); // Load user-related routes

const dotenv = require("dotenv"); // Load dotenv to use environment variables

dotenv.config(); // Load environment variables from the .env file

**2. Creating the Express App:**

* This step initializes the **Express app**. Think of it as starting up the web server so it can respond to requests.

js

Copy code

const app = express(); // Create an instance of the Express app

**3. Handling JSON Data:**

* app.use(express.json()): This line ensures that if we receive data in JSON format (like {"name": "John"}), it will be automatically converted into a JavaScript object so we can work with it easily.

js

Copy code

## Page:3

app.use(express.json()); // Allows the server to automatically handle JSON data

**4. User Routes:**

* app.use("/user", UserRouter): This line tells the server, "Hey, if someone goes to any URL starting with /user, let the UserRouter handle it." For example:
  + /user/register for registering users.
  + /user/login for logging in users.

js

Copy code

app.use("/user", UserRouter); // Mount user-related routes at /user

**5. Logging the Port:**

* process.env.PORT: This reads the **port number** from the .env file, which is usually something like PORT=3000. The port is where your server will listen for requests. The console.log just prints the port number to the console.

js

Copy code

console.log(process.env.PORT); // Print the port number to the console

**6. Starting the Server:**

* app.listen(process.env.PORT): This command tells the app to start listening for requests on the port number defined in the .env file.
* **Async/Await**: Inside this, we wait for the database to connect using await connection. If the connection to the database is successful, it logs the message: "connected to the db". If something goes wrong, it catches the error and prints "error".

js

Copy code

app.listen(process.env.PORT, async () => {

try {

await connection; // Wait for the database connection

console.log(`server is running at PORT ${process.env.PORT}`); // Server started

console.log("connected to the db"); // Successfully connected to the database

} catch (error) {

console.log("error"); // Log an error if the connection fails

## Page:4

}

});

**7. Environment Variables:**

* The dotenv.config() and process.env.PORT refer to the .env file. This file holds sensitive information like the port number or API keys that we don't want to hardcode in the app.
* Example of what the .env file might look like:

bash

Copy code

PORT=3000

DB\_URL=mongodb://localhost:27017/mydatabase

**Basic Concepts:**

* **Express.js**: A framework that helps you build web servers easily.
* **Database (MongoDB)**: Stores data like user information. The connection object is what connects your server to the database.
* **Routes**: Specific paths where requests are handled. Here, /user routes handle user-related operations.
* **Middleware**: app.use(express.json()) is a middleware that processes incoming JSON data.
* **Port**: This is the "door" your server listens to. It's defined in the .env file.

**Summary:**

This code creates a simple server using Express.js, connects to a database, and sets up routes to handle user-related requests. The server listens for incoming requests on a specific port (like localhost:3000), and when it's up and running, it prints messages to let you know the server is working and connected to the database.

# **db.js**

This code connects your application to a **MongoDB database** using **Mongoose**, a popular library in Node.js for working with MongoDB.

Let’s break it down in a simple way:

**1. Importing Mongoose:**

## Page:5

* mongoose: This is the library that helps us connect to and interact with a MongoDB database. Instead of writing raw database queries, Mongoose simplifies working with the data.

js

Copy code

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

**2. Connecting to MongoDB:**

* mongoose.connect("mongodb://127.0.0.1:27017/nupurDB"):
  + This line tells Mongoose to connect to the **MongoDB database** running on your computer (at 127.0.0.1 or localhost).
  + 27017 is the **default port** MongoDB listens on.
  + "nupurDB" is the name of the database. If it doesn’t exist, MongoDB will automatically create it when you add data.

**In simpler terms**: This line is connecting your app to the nupurDB database on your local MongoDB server.

js

Copy code

const connection = mongoose.connect("mongodb://127.0.0.1:27017/nupurDB"); // Connect to MongoDB database

**3. Exporting the Connection:**

* module.exports = connection: This line exports the connection so it can be used in other files. By doing this, other parts of your app (like your server) can use this connection to interact with the database.

**In simpler terms**: This allows you to use the database connection in other parts of your project.

js

Copy code

module.exports = connection; // Export the connection to be used in other files

**Summary:**

* **Mongoose** helps us connect to and work with MongoDB.
* The code connects to a MongoDB database (nupurDB) running locally on your computer.
* The connection is exported so it can be used in other parts of the application.

**Example Flow:**

1. **MongoDB** is a database that stores data.
2. **Mongoose** makes it easier to work with MongoDB.
3. This code connects to a **local MongoDB database** and exports that connection so you can use it in other files (like for saving or retrieving user data).

# **.env**

The .env file is a simple way to store **environment variables** for your application. Environment variables are settings or values that your app can use, like a **port number** or **API keys**, without hardcoding them in the code. This makes your app easier to manage and more secure.

Here’s a breakdown of the .env file you shared:

**1. PORT = 8080:**

* PORT is the environment variable for the **port number**. The port is like the "door" your application uses to communicate over the internet or on your local computer.
* In this case, it is set to 8080. When your app runs, it will listen for incoming requests on localhost:8080 or your-domain.com:8080.

**In simpler terms**: The port is like an address for your app, and 8080 is that address.

**2. KEY = "":**

* KEY is an environment variable used to store some kind of secret or API key, like a password or a key to access another service.
* Right now, it’s an empty string (""), meaning you haven’t set the value yet. When you need to connect to an external service (like a payment gateway, third-party API, or anything that requires a secret), you would store the actual key here.

**In simpler terms**: The KEY would be something sensitive that you don’t want to share publicly, like a password for another service. You keep it here so it’s not hardcoded in your code.

**Why Use a .env File?**

* **Security**: You don’t want sensitive data like API keys or passwords in your code, especially if you share it on GitHub. Storing them in a .env file keeps them safe.
* **Flexibility**: You can change settings (like the port) without changing the actual code.

**Example of How It’s Used in Code:**

When you use the dotenv package in your Node.js app, it loads these variables into process.env, so you can access them in your code like this:

**js**

Copy code

const PORT = process.env.PORT; // Access the PORT value from the .env file

const API\_KEY = process.env.KEY; // Access the KEY value from the .env file

## Page:6

**Summary:**

* **PORT**: The number 8080 tells your app which "door" to use for communication.
* **KEY**: A placeholder for a secret or API key, which is currently empty.

This file helps manage settings and sensitive information in one place, making your app more secure and easier to configure.

# **User.model.js**

This file, user.model.js, is used to define the **structure** of a user in the MongoDB database using **Mongoose**.

Let's break down what each part of this code does:

**1. Importing Mongoose:**

* mongoose: This is the library that helps us define **schemas** (the structure of the data) and interact with MongoDB (the database).

js

Copy code

const mongoose = require("mongoose"); // Load Mongoose to interact with MongoDB

**2. Defining the User Schema:**

* UserSchema: This is where you define how a **user** should look in the database. A **schema** is like a blueprint that describes the shape of the data. In this case, every user will have:
  + name: A string representing the user's name.
  + email: A string representing the user's email.
  + password: A string representing the user's password.

**In simpler terms**: The schema defines what fields (name, email, password) a user will have in the database and the type of each field (string in this case).

js

Copy code

const UserSchema = new mongoose.Schema({

name: String, // The name of the user (String)

email: String, // The email of the user (String)

password: String, // The password of the user (String)

});

## Page:7

**3. Creating the User Model:**

* UserModel: This takes the schema (UserSchema) and turns it into a **model**. A model is what we use to interact with the database.
  + With UserModel, we can do things like **add new users**, **find users**, **update users**, etc.
  + The model is named "user", so MongoDB will create a collection (a table in SQL terms) called users in the database.

**In simpler terms**: The model is like a tool we can use to create, read, update, or delete user data in the database.

js

Copy code

const UserModel = mongoose.model("user", UserSchema); // Create the UserModel to interact with the 'users' collection

**4. Exporting the Model:**

* module.exports = UserModel: This line exports the UserModel so that other parts of the app (like the routes) can use it to interact with the database.

**In simpler terms**: This makes the UserModel available to be used in other parts of the application, like for creating or finding users.

js

Copy code

module.exports = UserModel; // Export the UserModel to use in other files

**Summary:**

1. **Mongoose** is used to define how a user will look in the database.
2. The **schema** defines the structure of a user, with fields like name, email, and password, all of which are strings.
3. The **model** (UserModel) is what allows you to work with the users in the database (adding, reading, updating, or deleting users).
4. **Exporting** the model makes it accessible in other parts of the app, so you can use it in routes and controllers to manage users.

**Example Flow:**

1. You define what a user looks like (name, email, password).
2. You create a model that allows you to interact with the database.
3. You use the model elsewhere to add, update, or find users in the database.

## Page:8

# **User.controller.js**

This code defines routes (endpoints) for user-related actions like **viewing all users**, **registering new users**, and **logging in**. These routes are part of an **Express.js** web server and use **Mongoose** to interact with a MongoDB database.

Let’s break it down step by step in simple terms:

**1. Importing Required Modules:**

* **express**: Helps create routes for handling HTTP requests (e.g., GET, POST).
* **UserModel**: The model that interacts with the MongoDB database (which you defined in user.model.js).
* **bcrypt**: A library used to **hash** passwords (make them secure before saving them in the database).
* **jwt**: Stands for **JSON Web Token**, used to create tokens for authentication (though it's not fully implemented in this example).
* **dotenv**: Loads environment variables from a .env file.

js

Copy code

const express = require("express"); // Load Express framework

const UserModel = require("../models/user.model"); // Load the User model to interact with users in the database

const UserRouter = express.Router(); // Create a router for user-related routes

const bcrypt = require("bcrypt"); // Load bcrypt for password hashing

const jwt = require("jsonwebtoken"); // Load JWT for authentication (not fully used here)

const dotenv = require("dotenv"); // Load dotenv to access environment variables

**2. Creating the Router:**

* **UserRouter** is an instance of an Express **Router**. This allows you to define different routes for user-related actions like registering and logging in.

js

Copy code

const UserRouter = express.Router(); // Create a router to define user-related routes

## Page:9

**3. GET Route: View All Users:**

* **GET /alluserdata**: This route retrieves and returns all the users from the database.
* It uses UserModel.find() to get all user records and responds with the data in JSON format.

js

Copy code

UserRouter.get("/alluserdata", async (req, res) => {

try {

const alluserdata = await UserModel.find(); // Fetch all users from the database

res.status(200).json({ userdata: alluserdata }); // Respond with the user data

} catch (error) {

res.status(400).json({ message: "Error fetching users" }); // If there's an error, respond with an error message

}

});

**4. POST Route: Register a New User:**

* **POST /add**: This route is used for **adding a new user** (registering).
* First, the password is hashed using **bcrypt** for security.
* After hashing, a new user is created and saved in the database.

Steps:

1. Extract the user's name, email, and password from the request body.
2. Hash the password using **bcrypt** (with 5 rounds of hashing).
3. Save the user (with the hashed password) to the database.

js

Copy code

UserRouter.post("/add", (req, res) => {

const { name, email, password } = req.body; // Get name, email, and password from request body

console.log(name, email, password);

try {

## Page:10

bcrypt.hash(password, 5, async (err, hash) => { // Hash the password

if (err) {

return res.status(500).json({ message: "Error in hashing password" }); // Handle error in hashing

} else {

const singleuser = new UserModel({ name, email, password: hash }); // Create new user with hashed password

await singleuser.save(); // Save the user in the database

res.status(200).json({ message: "User created successfully" }); // Send success response

}

});

} catch (error) {

res.status(400).json({ message: error }); // If something goes wrong, send error response

}

});

**5. POST Route: Login a User:**

* **POST /login**: This route allows a user to log in.
* First, it looks for a user in the database with the provided email.
* Then, it compares the provided password with the stored (hashed) password using **bcrypt**.
* If the password matches, the user is logged in successfully.

Steps:

1. Extract email and password from the request body.
2. Find the user by their email using UserModel.findOne().
3. Compare the provided password with the hashed password using bcrypt.compare().
4. Respond with success or failure based on the result of the comparison.

js

Copy code

UserRouter.post("/login", async (req, res) => {

const { email, password } = req.body; // Get email and password from request body

try {

## Page:11

const matchuser = await UserModel.findOne({ email }); // Find the user by email in the database

const hashpassword = matchuser.password; // Get the hashed password stored in the database

bcrypt.compare(password, hashpassword, function (err, result) { // Compare the provided password with the hashed password

if (result) {

res.status(200).json({ message: "Login Successfully", user: matchuser }); // If passwords match, login is successful

} else {

res.status(400).json({ message: "Invalid Email or Password" }); // If passwords don't match, send an error message

}

});

} catch (error) {

res.status(400).json({ message: error }); // If there's an error, respond with an error message

}

});

**6. Exporting the Router:**

* Finally, we **export** the UserRouter so that it can be used in other parts of the application (like the main server file where routes are connected).

js

Copy code

module.exports = UserRouter; // Export the router so it can be used in other parts of the app

**Summary:**

1. **Express Router**: UserRouter is used to define routes for user-related actions.
2. **GET /alluserdata**: Retrieves all users from the database.
3. **POST /add**: Registers a new user by hashing their password and saving them to the database.
4. **POST /login**: Logs in a user by comparing their password with the stored hashed password.
5. **bcrypt**: Used to securely hash and compare passwords.
6. **Exporting**: The router is exported so that it can be used in other parts of the application.

## Page:12

**Example Flow:**

* **Registration**: A user sends their name, email, and password. The password is hashed, and the user is saved to the database.
* **Login**: A user provides their email and password. The password is compared with the hashed password in the database, and if it matches, the login is successful.

This controller manages how users interact with your application by creating, retrieving, and authenticating user data.