## Page:1

# Authorization

# Content.contoller.js

const express = require("express");

const jwt = require("jsonwebtoken");

const ContentRouter = express.Router();

const dotenv = require("dotenv");

dotenv.config();

ContentRouter.get("/home", (req, res) => {

  res.send("Welcome to home page");

});

//Private

ContentRouter.get("/movie", (req, res) => {

  //   const { token } = req.query;

  //   console.log(token);

  const { token } = req.headers;

  console.log(req.headers.token);

  //   res.send("ok");

  jwt.verify(token, process.env.SECRET\_KEY, function (err, decoded) {

    if (err) {

      res.status(401).send({ message: "Invalid token" });

    } else {

      //   console.log(decoded);

      if (decoded) {

        res.send("You can Striming Movies");

      } else {

        res.send("You cant Striming Movies");

## Page:2

    }

    }

  });

});

ContentRouter.get("/series", (req, res) => {

  const { token } = req.query;

  jwt.verify(token, process.env.SECRET\_KEY, function (err, decoded) {

    if (err) {

      res.status(401).send({ message: "Invalid token" });

    } else {

      //   console.log(decoded);/

      if (decoded) {

        res.send("You can Striming Movies");

      } else {

        res.send("You cant Striming Movies");

      }

    }

  });

});

module.exports = ContentRouter;

# Explanation

This code is a basic example of a Node.js application using the Express framework for building web servers and jsonwebtoken (jwt) for token-based authentication. Let’s break down the code and the concepts in a simple, beginner-friendly way.

**Key Concepts:**

## Page:3

1. **Express.js**: A popular framework for creating web servers in Node.js. It helps you easily define routes (URLs) and how the server should respond to requests.
2. **JWT (JSON Web Tokens)**: A method for securely transmitting information between a client (like your browser) and a server (like the one you're building). JWTs are often used for authentication (making sure the user is who they claim to be).
3. **dotenv**: A package to load environment variables from a .env file into your Node.js project. These variables are used to store sensitive information like your SECRET\_KEY for signing JWTs.

**Code Explanation:**

1. **Importing Dependencies**:

js

Copy code

const express = require("express");

const jwt = require("jsonwebtoken");

const dotenv = require("dotenv");

dotenv.config();

* + You are importing express to build the server and jwt to handle the token authentication.
  + dotenv is used to load environment variables from the .env file, which helps keep sensitive information like your SECRET\_KEY private.

1. **Setting Up the Router**:

js

Copy code

const ContentRouter = express.Router();

* + You are creating a Router using express. This will handle specific routes or paths (like /home, /movie, and /series).

1. **Public Route (/home)**:

js

Copy code

ContentRouter.get("/home", (req, res) => {

res.send("Welcome to home page");

});

## Page:4

* + This is a public route that anyone can access. When someone visits /home, the server responds with the message: "Welcome to home page".

1. **Private Route (/movie) with Authentication**:

js

Copy code

ContentRouter.get("/movie", (req, res) => {

const { token } = req.headers; // Extract token from request headers

jwt.verify(token, process.env.SECRET\_KEY, function (err, decoded) {

if (err) {

res.status(401).send({ message: "Invalid token" });

} else {

res.send("You can Striming Movies");

}

});

});

* + **Token Verification**: This route is protected and requires a valid JWT token to access.
  + The token is expected to be provided in the request headers (e.g., Authorization or token header).
  + The jwt.verify() function checks if the token is valid. If the token is invalid or missing, it sends back a 401 Unauthorized response with the message "Invalid token".
  + If the token is valid, it sends back the message: "You can Striming Movies".

1. **Private Route (/series) with Query Token**:

js

Copy code

ContentRouter.get("/series", (req, res) => {

const { token } = req.query; // Extract token from URL query parameters

jwt.verify(token, process.env.SECRET\_KEY, function (err, decoded) {

if (err) {

res.status(401).send({ message: "Invalid token" });

## Page:5

} else {

res.send("You can Striming Movies");

}

});

});

* + This is similar to the /movie route, but instead of getting the token from the headers, it gets the token from the query parameters of the URL.
  + The process of verifying the token is the same as in the /movie route.

1. **Exporting the Router**:

js

Copy code

module.exports = ContentRouter;

* + You are exporting ContentRouter so that it can be used in other parts of your application, usually in the main server file (app.js or server.js).

**Summary:**

* **Public route**: /home is available for everyone without any authentication.
* **Protected routes**: /movie and /series require a valid JWT token.
  + The /movie route expects the token in the request headers.
  + The /series route expects the token in the query parameters.
* **Token verification**: The jwt.verify() function checks if the provided token is valid using the SECRET\_KEY from your .env file. If the token is invalid, a 401 Unauthorized error is sent.

This structure allows you to secure certain parts of your application (like /movie and /series) and ensure only users with valid tokens can access those routes.

# User.controller.js

const express = require("express");

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

const bycrypt = require("bcrypt");

const jwt = require("jsonwebtoken");

## Page:6

const dotenv = require("dotenv");

dotenv.config();

console.log(process.env.SECRET\_KEY);

//###   Get All User Contoller

const getalluserdata = 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" });

  }

};

// ###Registration Routes ###

const registrationcontroller = (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" });

      }

## Page:7

    });

  } catch (error) {

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

  }

};

const logincontoller = 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

        //   .status(200)

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

        jwt.sign(

          { matchuser },

          process.env.SECRET\_KEY,

          { expiresIn: 20 },

          function (err, token) {

            if (err) {

              res.status(500).json({ message: "Error in generating token" });

            } else {

              res

                .status(200)

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

            }

## Page:8

  }

        );

      } else {

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

      }

    });

  } catch (error) {

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

  }

};

module.exports = { getalluserdata, registrationcontroller, logincontoller };

# Explanation

This code demonstrates a simple user authentication system using **Node.js**, **Express**, **bcrypt** for password hashing, and **jsonwebtoken (JWT)** for handling token-based authentication. Let's break down the code and concepts in an easy-to-understand way.

**Key Concepts:**

1. **Express.js**: A web framework for Node.js to handle routes and requests.
2. **bcrypt**: A library used to hash (encrypt) passwords, making them secure.
3. **JWT (JSON Web Tokens)**: Tokens used to authenticate users without storing session data on the server. The client (browser) holds the token, and the server validates it when needed.
4. **dotenv**: A module to manage environment variables, such as sensitive data like your SECRET\_KEY.

**Code Explanation:**

**1. Dependencies and Setup:**

js

Copy code

## Page:9

const express = require("express");

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

const bcrypt = require("bcrypt");

const jwt = require("jsonwebtoken");

const dotenv = require("dotenv");

dotenv.config(); // Load environment variables

console.log(process.env.SECRET\_KEY); // Check if SECRET\_KEY is loaded

* express: Used for setting up routes and handling requests.
* UserModel: Represents a MongoDB user model, handling user data.
* bcrypt: Used for hashing passwords before storing them.
* jwt: Used to create and verify tokens for authentication.
* dotenv: Loads environment variables from the .env file.

**2. Get All Users (getalluserdata):**

js

Copy code

const getalluserdata = async (req, res) => {

try {

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

res.status(200).json({ userdata: alluserdata }); // Send back user data

} catch (error) {

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

}

};

* This function fetches and returns all users from the database.
* UserModel.find() is used to get all user data.
* If successful, it sends a 200 OK response with the user data.
* If there's an error, it sends a 400 Bad Request response with an error message.

**3. Registration (registrationcontroller):**

js

## Page:10

Copy code

const registrationcontroller = (req, res) => {

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

try {

bcrypt.hash(password, 5, async (err, hash) => { // Hashing the password with a salt factor of 5

if (err) {

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

} else {

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

await singleuser.save(); // Saving the user to the database

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

}

});

} catch (error) {

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

}

};

* **Registration Process**:
  + This function allows users to register by submitting their name, email, and password.
  + The bcrypt.hash() function hashes the password with a salt (a random value to make the hash more secure).
  + After hashing, the user is saved to the database using UserModel.
  + If successful, it sends a response indicating the user was created.
  + If there's an error (e.g., in password hashing), it sends an error message.

**4. Login (logincontoller):**

js

Copy code

## Page:11

const logincontoller = async (req, res) => {

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

try {

const matchuser = await UserModel.findOne({ email }); // Find the user with the given email

const hashpassword = matchuser.password; // Get the hashed password from the user data

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

if (result) {

// Generate a JWT token on successful login

jwt.sign(

{ matchuser }, // Payload: user data

process.env.SECRET\_KEY, // Secret key from environment variable

{ expiresIn: 20 }, // Token expires in 20 seconds

function (err, token) {

if (err) {

res.status(500).json({ message: "Error in generating token" });

} else {

res.status(200).json({ message: "Login Successfully", token });

}

}

);

} else {

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

}

});

} catch (error) {

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

## Page:12

}

};

* **Login Process**:
  + The function looks for the user by their email using UserModel.findOne({ email }).
  + If a user is found, it compares the password entered by the user with the hashed password stored in the database using bcrypt.compare().
  + If the passwords match, a JWT token is generated with jwt.sign(). The token includes the user information and is signed with the SECRET\_KEY. The token is set to expire after 20 seconds (you can change this duration).
  + If the token is successfully created, the server responds with a success message and the token. If not, it sends an error message.

**5. Exporting Controllers:**

js

Copy code

module.exports = { getalluserdata, registrationcontroller, logincontoller };

* The three functions (getalluserdata, registrationcontroller, and logincontoller) are exported to be used in other parts of the application, typically in route files where they handle user requests.

**Summary:**

* **Registration Process**:
  + Users submit their name, email, and password.
  + The password is securely hashed using bcrypt before being stored in the database.
  + If successful, the user is registered and saved in the database.
* **Login Process**:
  + Users submit their email and password.
  + The password is compared to the hashed password in the database.
  + If the password matches, a JWT token is generated, which the user can use to authenticate future requests.
* **JWT Token**:
  + JWT tokens are created during login and contain user information.
  + These tokens are signed with a SECRET\_KEY and have an expiration time.
  + Users use this token to access protected routes in your application.