Backend Development [ using MERN stack ]

2 major component :-

* A Programig language
* A Database

1 Programing Language :- JAVA , JS , PHP , Golang , C++ ( Any One )

2 Database :- Mongo , My SQL , Postgrace , SQLite ( Any One )

What is Backend ??

:- The backend is the non-visible part of a website, application, or software system that handles all the behind-the-scenes operations that make it work

What is work of Backend Developers ??

:- Back-end developers, also known as back-end engineers, build and maintain the server-side of websites and applications. Their work includes:

* Data storage: Organizing and storing data
* Security: Ensuring the security of data
* Application logic: Creating and maintaining the core logic of the application
* Databases: Interacting with and maintaining databases
* API: Creating and maintaining APIs
* Integration: Integrating data and applications

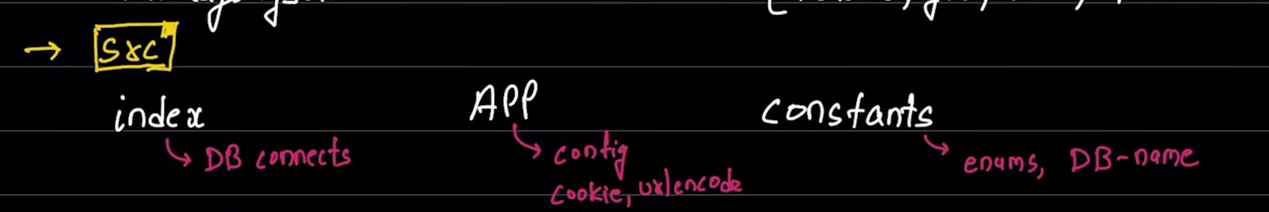
In Lamen terms backend developer handles 3 major things :

* Data
* API ( majorly third party api ) create API and handle API
* Files

File Structure For Backend Application –

First .env contains variabls and other information which is sensitive and these are not pushed in to github or are not shared.

Src folder :---



Index is majorly the entry point for Backend Application

Directory Structure ( File Structure )

* DB :- Database code is written ( connection of mongo DB )
* Modal :- Schem of DataBase is stored in Modal Folder
* Controlers :- Functions are stored in this folder ( Working of Application is defined )
* Routes :- Routes are defined
* Middleware :- Middleware are written in this folder
* Utils :- repetitive things are defined in this folder ( ex talk to database , send mail ,file upload )

This was File Structure for Backend Application Now Creation of the Server

EXPRESS JS:-

The following is the basic code example of setting up server with help of Expressjs

require('dotenv').config();

**const** express **=** require('express');

**const** app **=** express();

**const** port **=** process.env.PORT **||** 2000;  // Use port from .env or default to 2000

**const** arr **=** [{

  name: "Shashank",

  id: 1,

  age: 21

},

{

  name: "Sharvary",

  id: 2,

  age: 17

},

{

  name: "Nita",

  id: 3,

  age: 45

},

{

  name: "Narayan",

  id: 4,

  age: 55

}];

app.get('/', (req, res) **=>** {

  res.send('Hello World!');

});

app.get( '/api/arr' , ( req , res ) **=>** {

  res.send(arr);

} )

app.get('/login', (req, res) **=>** {

  res.send('<h1>Welcome to Login page</h1>');

});

app.listen(port, () **=>** {

  console.log(`Example app listening on port ${port}`);

});

Now this is just the basic code to create server using express js now this code is also providing API response on route ( /api/arr )

EXPLAINING NEW TERMS OF THIS CODE

* require('dotenv').config();: This loads environment variables from a .env file into process.env. This is useful for storing configuration values like the port number.

--NOW THE EXAMPLE FRONT END CODE TO ACCESS DATA FROM OUR CREATED SERVER—

**import** { useEffect, useState } **from** 'react'

**import** reactLogo **from** './assets/react.svg'

**import** viteLogo **from** '/vite.svg'

**import** './App.css'

**import** axios **from** 'axios'

**function** App() {

**const** [arr, setArr] **=** useState([]);

  useEffect(() **=>** {

    axios.get('/api/arr')

    .then((res) **=>** {

      setArr(res.data);

    })

    .catch((er) **=>** {

      console.log(er);

    })

  });

**return** (

    <>

      <**h1**>Hello world</**h1**>

      <**p**>Array :- {arr.length}</**p**>

      {

        arr.map((ar) **=>** (

          <**div** key**=**{ar.id}>

            <**h2**>{ar.name}</**h2**>

            <**p**>{ar.age}</**p**>

          </**div**>

         ) )

      }

    </>

  )

}

**export** **default** App

This Is the react code to access info from our server ( basically accessing api data ) this is very simple code no need to explain it in detail

In this code there are major concepts used named AXIOS , PROXY , CORS

* Axios is a popular JavaScript library used to make HTTP requests from Node.js or XMLHttpRequests from the browser. It supports the Promise API, which is native to JavaScript ES6, making it easy to handle asynchronous operations.

CORS :-

CORS (Cross-Origin Resource Sharing) errors occur when a web application attempts to make requests to a server that is on a different domain, protocol, or port than its own. By default, web browsers block such cross-origin requests for security reasons.

How to Fix CORS Errors

1. Server-Side Solution

To allow cross-origin requests, the server must include specific headers in its responses. This can be done by setting the Access-Control-Allow-Origin header.

Node.js and Express:

You can use the cors middleware to handle CORS in your Express application.

First, install the cors package:

npm install cors

Then, use it in your Express application:

**const** express **=** require('express');

**const** cors **=** require('cors');

**const** app **=** express();

app.use(cors());

**const** port **=** process.env.PORT **||** 2000;

app.get('/', (req, res) **=>** {

  res.send('Hello World!');

});

app.get('/api/arr', (req, res) **=>** {

**const** arr **=** [

    { name: "Shashank", id: 1, age: 21 },

    { name: "Sharvary", id: 2, age: 17 },

    { name: "Nita", id: 3, age: 45 },

    { name: "Narayan", id: 4, age: 55 }

  ];

  res.send(arr);

});

app.listen(port, () **=>** {

  console.log(`Example app listening on port ${port}`);

});

The cors middleware automatically sets the Access-Control-Allow-Origin header to allow requests from any origin. You can also configure it to allow only specific origins:

app.use(cors({

  origin: 'http://your-frontend-domain.com'

}));

**2. Client-Side Solution**

While the best practice is to handle CORS on the server side, sometimes you might need to work around CORS issues during development.

**Using a Proxy in Development:**

If you're using a development server like Vite, you can configure a proxy to bypass CORS issues.

For example, in a Vite project, you can configure the proxy in vite.config.js:

**import** { defineConfig } **from** 'vite'

**import** react **from** '@vitejs/plugin-react'

// https://vitejs.dev/config/

**export** **default** defineConfig({

  server: {

    proxy: {

      '/api': 'http://localhost:3000'

    }

  },

  plugins: [react()],

})

This setup will proxy requests from http://localhost:3000/api (Vite's default port) to <http://localhost:2000/api>.

**3. Browser Extensions**

During development, you can also use browser extensions that temporarily disable CORS checks. This is not recommended for production but can be useful for quick testing.

* **Chrome**: Allow CORS: Access-Control-Allow-Origin
* **Firefox**: [CORS Everywhere](https://addons.mozilla.org/en-US/firefox/addon/cors-everywhere/)

HOW TO SETUP A PROFESSIONAL BACKEND PROJECT

First run npm init command on terminal and follow the steps and when our packge.json file is created after do the following steps.

1. Make .gitignore file and use gitignore generator to generate the gitignore content
2. Create .env file
3. Create file Structure defined in the beginning of this pdf in your project
4. Config your package.json file

Configuration of Package,json –

  "name": "project1",

  "version": "1.0.0",

  "description": "first project",

  "type": "module",

  "main": "index.js",

Note here type is written as module which tell our application that we are using modulejs and not commonjs

1. Install dependency --

Install Nodemon

  "scripts": {

    "dev": "nodemon src/index.js"

  },

Do this modification In package.json

1. Use extensions such as Prettier and dotenv in your projects

PRETTIER :- Good Practice for Production grade code ( Suggested By Profectionals )

Usecase : - when we write code in production we write code in the team and there are some standards defined within the team which we have to follow

for ex – tab space should 4 or 2 space

Setup of Prettier

First remember it is dev dependency so install it as dev dependencies

npm i -D prettier

then make .prettierc mean prettier config file which store configuration of prettier

ex

{

    "singleQuote": **false**,

    "bracketSpacing": **true**,

    "tabWidth": 2,

    "trailingComma": "es5"

}

Make .prettierignore file which define which things the prettier should ignore

For example

/.vscode

/node\_modules

./dist

\*.env

.env

.env.\*

These are some file prettier ignore

---Connect To MongoDB---

Don’t use MongoDB locally weather as use MongoDB atlas which is online MongoDB

Cause there are some advantages of using MongoDB online

MongoDB Atlas may be preferred over MongoDB Compass for several reasons, including:

Scalability

MongoDB Atlas allows databases to scale up or down based on workload requirements, while MongoDB Compass does not support database scaling.

Cloud-based

MongoDB Atlas is a cloud-based database service that allows developers to get started right away in any major public cloud.

Security

MongoDB Atlas offers built-in defaults for access and end-to-end encryption to secure data. It also has an alert system to help address issues quickly, and transparent and automated recovery from instance failures.

Now Code to connect MongoDB –

First use dotenv package to use environment variables but there are some problems while using dotenv package and that are solved in package.json file by adding experimental feature of dotenv

 "scripts": {

    "dev": "nodemon -r dotenv/config --experimental-json-modules src/index.js"

  },

This is the code

Now code to connect MongoDB

**import** mongoose  **from** "mongoose";

**import** {DB\_NAME} **from** "../constants.js";

**const** connectDB **=** **async** () **=>** {

**try** {

**const** connectionInstance **=** **await** mongoose.connect(`${process.env.MONGODB\_URL}/${DB\_NAME}`);

        console.log(`\nMONGO\_DB connected !! DB\_HOST: ${connectionInstance.connection.host}`);

    } **catch** (error) {

        console.log("MONGO\_DB connection error ",error);

        process.exit(1);

    }

}

**export** **default** connectDB;

Now what does this code do

This code defines a function to connect to a MongoDB database using Mongoose, a popular MongoDB object modeling tool designed to work in an asynchronous environment.

Let's break it down step-by-step:

**1. Importing Mongoose**

**import** mongoose **from** "mongoose";

This line imports Mongoose, which will be used to establish a connection to the MongoDB database and interact with it.

**2. Importing Constants**

**import** { DB\_NAME } **from** "../constants.js";

This imports a constant named DB\_NAME from a constants.js file located one directory up from the current file. This constant is likely used to specify the name of the database to connect to.

**3. Defining the connectDB Function**

**const** connectDB **=** **async** () **=>** {

Here, an asynchronous function named connectDB is defined. This function will handle the database connection process.

**4. Establishing the Connection**

**try** {

**const** connectionInstance **=** **await** mongoose.connect(`${process.env.MONGODB\_URL}/${DB\_NAME}`);

        console.log(`\nMONGO\_DB connected !! DB\_HOST: ${connectionInstance.connection.host}`);

    } **catch** (error) {

        console.log("MONGO\_DB connection error ",error);

        process.exit(1);

    }

**- try { ... } catch (error) { ... }**

This block is used to handle the asynchronous operation and potential errors.

**- const connectionInstance = await mongoose.connect(...)**

This line attempts to connect to the MongoDB database. The connection string is constructed using the MONGODB\_URL environment variable and the DB\_NAME constant. The await keyword is used because mongoose.connect returns a promise.

**Example**

Suppose process.env.MONGODB\_URL is set to "mongodb://localhost:27017" and DB\_NAME is "myDatabase". The connection string will be:

**await** mongoose.connect("mongodb://localhost:27017/myDatabase");

**- console.log(...)**

If the connection is successful, this line logs a message indicating that the MongoDB connection is established, including the host of the connected database.

Now till this point connection is done but server is not created yet now its time to create the server

**import** dotenv **from** "dotenv"

**import** connectDB **from** "./db/index.js";

**import** express **from** "express"

**const** app **=** express();

dotenv.config({

    path: './env'

})

connectDB()

.then(

    () **=>** {

        app.listen(process.env.PORT **||** 8000 , () **=>** {

            console.log(`Server is listeningon port ${process.env.PORT}`)

        })

        app.on("error",(Error) **=>** {

            console.log("ERRR",Error);

**throw** Error

        })

    }

)

.catch(

    (err) **=>** {

        console.log("Failed to connect :- ",err)

    }

)

Now here what we do is we started a server after connecting to mongoDB

PORT no of server is 8000 or port from environment variable ( Note for this app only )\

In detail explanation

Starting the Server

connectDB()

.then(

    () **=>** {

        app.listen(process.env.PORT **||** 8000 , () **=>** {

            console.log(`Server is listeningon port ${process.env.PORT}`)

        })

        app.on("error",(Error) **=>** {

            console.log("ERRR",Error);

**throw** Error

        })

    }

)

.catch(

    (err) **=>** {

        console.log("Failed to connect :- ",err)

    }

)

**Connecting to the Database**

* connectDB(): Calls the function to connect to the MongoDB database.
* .then(...): If the connection is successful, the then block is executed.

**Starting the Server**

* app.listen(process.env.PORT || 8000, ...): Starts the Express server on the port specified by the PORT environment variable or defaults to port 8000.
* console.log(...): Logs a message indicating that the server is listening on the specified port.

**Handling Server Errors**

* app.on("error", (error) => { ... }): Listens for error events on the Express application.
* throw error: Throws the error to stop the application.

**Handling Connection Failures**

* .catch(...): If the database connection fails, the catch block is executed.
* console.log("Failed to connect :- ", err): Logs the connection error.

This setup ensures that the server only starts listening for requests if the connection to the MongoDB database is successfully established. If the connection fails, it logs an error message and does not start the server.

NOW SETUP APP.JS file in src

**import** express **from** "express"

**import** cors **from** "cors"

**import** { limit } **from** "./constants";

**import** cookieParser **from** "cookie-parser";

**const** app **=** express();

console.log("corsOrigin :- ",process.env.CORS\_ORIGIN)

//config crossplatform origin

app.use( cors({

    origin:process.env.CORS\_ORIGIN,

    credentials:**true**

}) )

//setting limit to ascept data

app.use( express.json({limit:limit}) );

app.use( express.urlencoded({ extended:**true** , limit:limit }) );

app.use( express.static("public") );

app.use( cookieParser() )

Explanation

1. Importing Modules

**import** express **from** "express";

**import** cors **from** "cors";

**import** { limit } **from** "./constants";

**import** cookieParser **from** "cookie-parser";

express: This is the Express framework for Node.js, used to build web applications and APIs.

cors: This is a middleware to enable Cross-Origin Resource Sharing (CORS) with various options.

limit: This is an imported constant, likely defining the maximum request body size.

cookieParser: This middleware parses cookies attached to the client request object.

2. Creating the Express Application

**const** app **=** express();

This line initializes a new Express application and assigns it to the app variable.

Configuring CORS

app.use(cors({

    origin: process.env.CORS\_ORIGIN,

    credentials: **true**

}));

* app.use(cors({...})): This line sets up the CORS middleware with the following options:
  + origin: process.env.CORS\_ORIGIN: Specifies the allowed origin(s) for cross-origin requests, which is set through an environment variable.
  + credentials: true: Allows cookies to be included in the requests from the client.

**Example**: If process.env.CORS\_ORIGIN is set to "http://example.com", only requests from http://example.com will be allowed.

Setting Request Body Limits

app.use(express.json({ limit: limit }));

app.use(express.urlencoded({ extended: **true**, limit: limit }));

* express.json({ limit: limit }): Parses incoming requests with JSON payloads and limits the size of the request body to the value of limit.
* express.urlencoded({ extended: true, limit: limit }): Parses incoming requests with URL-encoded payloads and limits the size of the request body to the value of limit.

**Example**: If limit is set to "10mb", the request body size is limited to 10 megabytes.

Serving Static Files

app.use(express.static("public"));

* express.static("public"): Serves static files from the public directory.

**Example**: If there's a file public/index.html, it can be accessed via <http://yourserver.com/index.html>.

Parsing Cookies

app.use(cookieParser());

* This middleware parses cookies attached to the client request object and makes them available under req.cookies object.

**Example**: If the client sends a cookie name=John, it can be accessed as req.cookies.name.

**Putting it all together**

Here's an example of how the server might respond to different requests:

1. **CORS Request**:
   * If a client from http://example.com makes a request, it will be allowed if process.env.CORS\_ORIGIN is set to "http://example.com".
2. **JSON Payload**:
   * A client sends a JSON payload {"key": "value"}. The server will parse this JSON if its size is within the limit.
3. **URL-encoded Payload**:
   * A client sends a URL-encoded payload key=value. The server will parse this payload if its size is within the limit.
4. **Static File Request**:
   * A client requests http://yourserver.com/index.html. If this file exists in the public directory, it will be served.
5. **Cookies**:
   * A client sends a cookie session=abc123. This cookie will be parsed and available as req.cookies.session.

This setup is quite common for an Express server, which handles various requests while maintaining security and efficiency through CORS, request body size limits, static file serving, and cookie parsing.

Now after this create a async handler function which we would use further when we talk to our database while performing various operations

**const** asyncHandler **=** () **=>** {}  FIRST FUNCTION

**const** asyncHandler **=** (func) **=>** {  FUNCTION PASSED AS PARAMETER TO THE FUNCTION AND FUNCTION IS EXECUTED INSIDE THIS FUNCTION

**async** () **=>** {}   THE FUNCTION EXECUTED IS A ASYNC FUNCTION

}

**const** asyncHandler **=** (func) **=>** **async** () **=>** {}  ADVANCE SYNTAX OF ABOVE IMPLEMENTATION { THESE ARE CALLED HIGER ORDER FUNCTION }

**const** asyncHandler **=** (fn) **=>** **async** (req, res, next) **=>** {

**try** {

**await** fn(req, res, next)

    } **catch** (error) {

        res.status(err.code **||** 500).json({

            success: **false**,

            message: err.message

        })

    }

}

Now explaining this function

The code you've provided is a higher-order function called asyncHandler. It takes a function fn as an argument and returns a new asynchronous function that handles errors for the given function fn. This is useful in an Express.js application to handle asynchronous operations and ensure that any errors are properly caught and sent as responses.

**Explanation**

1. **Higher-Order Function**: asyncHandler is a higher-order function because it takes a function (fn) as an argument and returns a new function.
2. **Async Wrapper**: The returned function is asynchronous and wraps the original function (fn). It receives the usual Express.js middleware parameters: req, res, and next.
3. **Try-Catch Block**: Inside the returned function, the original function (fn) is executed inside a try block. If fn completes successfully, it proceeds normally. If an error occurs, the catch block handles it.
4. **Error Handling**: If an error occurs, it is caught and an error response is sent. The response includes a status code (defaulting to 500 if not specified in the error) and a JSON object containing a success flag and the error message.

**Example Usage**

Let's see how you might use this asyncHandler in an Express.js route:

**const** express **=** require('express');

**const** app **=** express();

// Example async route handler

**const** getUser **=** **async** (req, res, next) **=>** {

**const** userId **=** req.params.id;

**const** user **=** **await** findUserById(userId); // Assume this is an async function that fetches user data

**if** (**!**user) {

**throw** **new** Error('User not found');

    }

    res.json({

        success: **true**,

        data: user

    });

};

// Using asyncHandler to wrap the getUser function

app.get('/user/:id', asyncHandler(getUser));

app.listen(3000, () **=>** {

    console.log('Server is running on port 3000');

});

**What Happens Here**

1. **Route Definition**: The getUser function is defined as an asynchronous function that retrieves a user by their ID. If the user is not found, it throws an error.
2. **Error Handling with asyncHandler**: When defining the route /user/:id, the asyncHandler is used to wrap the getUser function. This ensures that any errors occurring inside getUser are caught and handled properly.
3. **Error Response**: If getUser throws an error (e.g., user not found), the asyncHandler catches it and sends a response with a 500 status code and the error message.

**Benefits**

* **Clean Code**: Using asyncHandler helps keep your route handlers clean and free of repetitive try-catch blocks.
* **Centralized Error Handling**: It provides a centralized way to handle errors in your asynchronous route handlers, making the error-handling logic consistent across your application.

Second approach to do the same thing

**const** asyncHandler **=** (requestHandler) **=>** {   //requestHandler is a function

**return** (req, res, next) **=>** {

        Promise

        .resolve(requestHandler(req, res, next))

        .catch((err) **=>** next(err))

    }

}

**export** { asyncHandler }

**Explanation of the Second Version**

1. **Higher-Order Function**: asyncHandler is still a higher-order function that takes a requestHandler function as an argument.
2. **Return Function**: It returns a new function that takes the usual Express.js parameters: req, res, and next.
3. **Promise Handling**:
   * Promise.resolve(requestHandler(req, res, next)) ensures that requestHandler is executed and its result is wrapped in a resolved promise.
   * .catch((err) => next(err)) catches any errors that occur during the execution of requestHandler and passes them to the next middleware (which is typically the error-handling middleware).

Now creating class to handle errors

**class** ApiError **extends** Error {

**constructor**(

        statusCode,

        message**=** "Something went wrong",

        errors **=** [],

        stack **=** ""

    ){

**super**(message)

**this**.statusCode **=** statusCode

**this**.data **=** **null**

**this**.message **=** message

**this**.success **=** **false**;

**this**.errors **=** errors

**if** (stack) {

**this**.stack **=** stack

        } **else**{

            Error.captureStackTrace(**this**, **this**.constructor)

        }

    }

}

**export** {ApiError}

This code defines a custom error class ApiError that extends the built-in Error class in JavaScript. This class is designed to provide more structured error information in an API context.

**Explanation**

1. **Class Definition**: ApiError extends the built-in Error class.
2. **Constructor Parameters**:
   * statusCode: The HTTP status code associated with the error.
   * message: A message describing the error (default is "Something went wrong").
   * errors: An array to hold additional error details (default is an empty array).
   * stack: The stack trace (default is an empty string).
3. **Super Call**: super(message) calls the constructor of the parent Error class with the provided message.
4. **Instance Properties**:
   * statusCode: Stores the provided HTTP status code.
   * data: Initialized to null (could be used to store additional data later).
   * message: Stores the error message.
   * success: Set to false, indicating the operation was not successful.
   * errors: Stores the provided array of error details.
5. **Stack Trace Handling**:
   * If a stack is provided, it sets the stack property directly.
   * If no stack is provided, it captures the stack trace for the current instance using Error.captureStackTrace.

**Example Usage**

Here's how you might use ApiError in an Express.js application:

**const** { ApiError } **=** require('./path/to/ApiError'); // Adjust the path as necessary

**const** getUser **=** **async** (req, res, next) **=>** {

**try** {

**const** userId **=** req.params.id;

**const** user **=** **await** findUserById(userId); // Assume this is an async function that fetches user data

**if** (**!**user) {

**throw** **new** ApiError(404, "User not found");

        }

        res.json({

            success: **true**,

            data: user

        });

    } **catch** (err) {

        next(err);

    }

};

Error Handling Middleware

**const** errorHandler **=** (err, req, res, next) **=>** {

**if** (err **instanceof** ApiError) {

        res.status(err.statusCode).json({

            success: err.success,

            message: err.message,

            errors: err.errors

        });

    } **else** {

        res.status(500).json({

            success: **false**,

            message: "Internal Server Error"

        });

    }

};

app.use(errorHandler);

**Example Error Object**

Suppose findUserById throws an ApiError

**const** err **=** **new** ApiError(404, "User not found", ["User ID is invalid"]);

This would result in an error response like:

{

"success": false,

"message": "User not found",

"errors": ["User ID is invalid"]

}

**Benefits**

1. **Structured Error Information**: Provides a standardized way to include HTTP status codes, error messages, and additional details.
2. **Ease of Use**: Simplifies error handling in your API by using a consistent error structure.
3. **Stack Trace Management**: Captures stack traces for debugging while allowing customization.

By using the ApiError class, you can ensure that your API's error responses are consistent and informative, making it easier to debug issues and handle errors gracefully.

Above was the code for handling error now this is the code for handling response

**class** ApiResponse {

**constructor**(statusCode, data, message **=** "Success"){

**this**.statusCode **=** statusCode

**this**.data **=** data

**this**.message **=** message

**this**.success **=** statusCode **<** 400

    }

}

**export** { ApiResponse }

Now its time to write aggregation pipeline and use bcrypt to hash password and use JWT ( json web token ) in our application JWT help us to use feature such as Access tokens

NOTE :- An access token is a small piece of code that contains a lot of information about a user, their permissions, groups, and timeframes. It's passed from a server to a user's device and can be used to authenticate a client application to access specific resources on behalf of the resource owner.

First create user model is the easy thing

After that

Pre-Save Hook for Password Hashing

userSchema.pre("save", **async** **function** (next) {  //Arrow function is not used cauze Arrow function dont give us access to this

**if**(**!this**.isModified("password")) **return** next();

**this**.password **=** **await** bcrypt.hash(**this**.password, 10)

    next()

})

* This middleware function hashes the password before saving the user document. It only hashes the password if it has been modified.

**4. Instance Methods**

**isPasswordCorrect**

userSchema.methods.isPasswordCorrect **=** **async** **function**(password){

**return** **await** bcrypt.compare(password, **this**.password)

}

* This method compares a given password with the hashed password stored in the database.

**generateAccessToken**

userSchema.methods.generateAccessToken **=** **function**(){

**return** jwt.sign(

        {

            \_id: **this**.\_id,

            email: **this**.email,

            username: **this**.username,

            fullName: **this**.fullName

        },

        process.env.ACCESS\_TOKEN\_SECRET,

        {

            expiresIn: process.env.ACCESS\_TOKEN\_EXPIRY

        }

    )

}

* This method generates an access token containing the user's ID, email, username, and full name.

**generateRefreshToken**

userSchema.methods.generateRefreshToken **=** **function**(){

**return** jwt.sign(

        {

            \_id: **this**.\_id,

        },

        process.env.REFRESH\_TOKEN\_SECRET,

        {

            expiresIn: process.env.REFRESH\_TOKEN\_EXPIRY

        }

    )

}

* This method generates a refresh token containing only the user's ID.

**5. Exporting the User Model**

**export** **const** User **=** mongoose.model("User",userSchema);

**Example Usage**

Here's how you might use this User model in a typical Express application:

**Registering a New User**

**export** **const** User **=** mongoose.model("User",userSchema);

**import** { User } **from** "./models/User";

app.post("/register", **async** (req, res) **=>** {

**try** {

**const** { username, password, email, fullname, avatar } **=** req.body;

**const** user **=** **new** User({ username, password, email, fullname, avatar });

**await** user.save();

        res.status(201).send({ message: "User registered successfully" });

    } **catch** (error) {

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

    }

});

Authenticating a User

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

**try** {

**const** { username, password } **=** req.body;

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

**if** (**!**user) {

**return** res.status(401).send({ error: "Invalid credentials" });

        }

**const** isPasswordCorrect **=** **await** user.isPasswordCorrect(password);

**if** (**!**isPasswordCorrect) {

**return** res.status(401).send({ error: "Invalid credentials" });

        }

**const** accessToken **=** user.generateAccessToken();

**const** refreshToken **=** user.generateRefreshToken();

        // Optionally save the refresh token in the database

        user.refreshToken **=** refreshToken;

**await** user.save();

        res.send({ accessToken, refreshToken });

    } **catch** (error) {

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

    }

});

in this example:

1. **Registering a User**:
   * The register endpoint creates a new user and saves it to the database.
   * The password is hashed before saving due to the pre-save hook.
2. **Logging In**:
   * The login endpoint finds a user by username and checks the password using the isPasswordCorrect method.
   * If the credentials are correct, it generates access and refresh tokens using generateAccessToken and generateRefreshToken methods.
   * The refresh token is optionally saved to the database.

This demonstrates how to register and authenticate users using the User model with password hashing and token generation.

NOTE : - JWT is a bearer tocken ( i.e A bearer token is a security token used in web applications and APIs to authenticate and authorize requests and access )

---FILE UPLOAD USING ( MULTER )---

File handling is not done on our server its done on third party app example ( AWS , CLOUDNARY , etc )

Service used : -

First is cloudinary for storage

Second multer package from npm

1 Cloudinary : -

What clodinary do is it takes file and upload it on its server

Our strategy is : -

Use multer take file from user and upload it on our server for temprorary time and then upload file on cloudinary

This is req cause if something happen during upload on cloudinary we can retry upload file from our server and do not tell user to retry

This is 2 step upload and it is done in Production

So now first create file named cloudinary.js in utils

Now to read write and remove file use fileSystem from Nodejs

**import** { v2 **as** cloudinary } **from** "cloudinary"

**import** fs **from** "fs"

cloudinary.config({

    cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,

    api\_key: process.env.CLOUDINARY\_API\_KEY,

    api\_secret: process.env.CLOUDINARY\_API\_SECRET

});

**const** uploadOnCloudinary **=** **async** (localFilePath) **=>** {

**try** {

**if** ( **!**localFilePath ) **return** **null**;

**else** {

**const** response **=** **await** cloudinary.uploader.upload( localFilePath , { resource\_type: "auto" } )

            console.log( "File uploaded successfully" , response.url );

**return** response;

        }

    } **catch** ( err ) {

        fs.unlinkSync(localFilePath)  //if file is not uploaded from local it removes file from locally

**return** **null**

    }

}

**export** { uploadOnCloudinary };

This code configures the Cloudinary service for uploading files from your local server to Cloudinary's cloud storage. It also includes error handling to clean up local files if the upload fails. Here's a detailed breakdown of each part:

cloudinary.config({

    cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,

    api\_key: process.env.CLOUDINARY\_API\_KEY,

    api\_secret: process.env.CLOUDINARY\_API\_SECRET

});

This block sets up the Cloudinary configuration using environment variables. These variables should be set in your environment to keep your credentials secure.

**Defining the Upload Function**

**const** uploadOnCloudinary **=** **async** (localFilePath) **=>** {

**try** {

**if** (**!**localFilePath) **return** **null**;

**else** {

**const** response **=** **await** cloudinary.uploader.upload(localFilePath, { resource\_type: "auto" });

            console.log("File uploaded successfully", response.url);

**return** response;

        }

    } **catch** (err) {

        fs.unlinkSync(localFilePath); // Removes the local file if upload fails

**return** **null**;

    }

}

This function handles the upload process:

1. **Check if localFilePath is provided**:
   * If localFilePath is not provided, the function returns null.
2. **Upload the file to Cloudinary**:
   * **cloudinary.uploader.upload**: This uploads the file at localFilePath to Cloudinary. The { resource\_type: "auto" } option tells Cloudinary to automatically detect the file type (image, video, etc.).
   * **Log success**: If the upload is successful, it logs the URL of the uploaded file and returns the response from Cloudinary.
3. **Error Handling**:
   * If an error occurs during the upload, it catches the error, deletes the local file using fs.unlinkSync(localFilePath), and returns null.

This was config of cloudinary for uploading files from local

Now Configuration of Muter

This code configures Multer to use disk storage for handling file uploads in a Node.js application. Here’s a detailed breakdown of each part of the code:

**import** multer **from** "multer";

**const** storage **=** multer.diskStorage({

    destination: **function** (req, file, cb) {

      cb(**null**, "./public/temp")

    },

    filename: **function** (req, file, cb) {

    //   const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() \* 1E9)

      cb(**null**, file.originalname)   // can be updated

    }

  })

**export** **const** upload **=** multer({

    storage,

})

1. **destination**: This function sets the directory where the files will be saved.
   * **Parameters**: It receives the request object (req), the file object (file), and a callback function (cb).
   * **Implementation**: cb(null, "./public/temp") tells Multer to save files in the ./public/temp directory. The first parameter of cb is null, indicating no error occurred.
2. **filename**: This function determines the name under which the file will be saved.
   * **Parameters**: It receives the request object (req), the file object (file), and a callback function (cb).
   * **Implementation**: cb(null, file.originalname) sets the file name to be the same as the original file name. You can customize this to ensure unique filenames, e.g., by appending a timestamp or a random string to avoid conflicts.

**Creating the Multer Instance**

**export** **const** upload **=** multer({

    storage,

})

This line creates and exports an instance of Multer configured to use the disk storage settings defined earlier.

* **storage**: This key takes the storage configuration created above.

**Usage Example**

To use this upload middleware in a route for handling file uploads, you can do something like this:

**import** express **from** 'express';

**import** { upload } **from** './path-to-your-upload-file';

**const** app **=** express();

app.post('/upload', upload.single('file'), (req, res) **=>** {

    // Handle the uploaded file here

    res.send('File uploaded successfully!');

});

app.listen(3000, () **=>** {

    console.log('Server started on port 3000');

});

In this example, when a POST request is made to /upload with a file in the file field of the form, Multer will store the file in the ./public/temp directory with its original name.

**Summary**

* **Imports Multer** to handle file uploads.
* **Configures disk storage** with a specific destination (./public/temp) and filename strategy (original name).
* **Exports the Multer instance** configured with the storage settings, ready to be used as middleware in routes to handle file uploads.

---WRITING THE FIRST ROUTE---

//Import route

**import** userRouter **from** "./routes/user.routes.js";

//Router Decleration

app.use("/api/v1/users",userRouter);

write this in app.js after config of json cookie parser etc

The route import and route declaration parts of your code are critical for defining and managing how your application handles incoming HTTP requests. Let's break down these parts in detail.

**1. Route Import**

import userRouter from "./routes/user.routes.js";

Purpose: This line imports a router object (userRouter) from the user.routes.js file, which is likely located in a routes directory.

Router Object: In Express.js, a router object is essentially a mini-application that you can attach to the main app object. The router allows you to modularize your application by separating different sections of your API into different files. This is useful for keeping your code organized, especially in larger applications.

Where It’s Defined: The userRouter is defined in the user.routes.js file, where you specify the various routes (endpoints) and the HTTP methods (GET, POST, etc.) that your application will respond to. This file typically also links these routes to their corresponding controller functions, which handle the actual logic for each route.

Example user.routes.js:

**import** { Router } **from** "express";

**import** { registerUser } **from** "../controllers/user.controller.js";

**const** router **=** Router();

router.route("/register").post(registerUser);

**export** **default** router;

* **Explanation:**
  + Router() creates a new router instance.
  + router.route("/register").post(registerUser); defines a POST route at the /register endpoint, which will invoke the registerUser controller function when hit.

**2. Route Declaration**

app.use("/api/v1/users", userRouter);

* **Purpose:** This line tells your main Express application (app) to use the userRouter for any routes that begin with /api/v1/users.
* **Path Prefixing:** The string /api/v1/users is a path prefix. This means that all routes defined in userRouter will be accessible under this path. For example:
  + If userRouter has a route POST /register, it will be available at /api/v1/users/register.
  + If userRouter had a route GET /profile, it would be available at /api/v1/users/profile.
* **Modularity:** This approach keeps your application modular and organized. Instead of defining all routes in the main file, you can define related routes together in separate files and bring them together in the main application using app.use().

**3. How It Works Together**

* When your Express server receives an HTTP request, it checks the route path. If the request path starts with /api/v1/users, the request is forwarded to userRouter.
* userRouter then checks for a matching route within itself. If it finds one, it executes the corresponding controller function.
* If the request path does not start with /api/v1/users, Express continues to check other app.use() declarations or throws a 404 error if no match is found.

**Example Flow:**

1. **Client Request:** A client sends a POST request to http://yourdomain.com/api/v1/users/register.
2. **Express Matching:** Express checks for a route that matches /api/v1/users. It finds the userRouter via app.use("/api/v1/users", userRouter);.
3. **Router Matching:** userRouter checks for a matching route for /register. It finds the POST /register route and executes the registerUser controller function.
4. **Response:** The registerUser function processes the request and sends back a response.

**Benefits:**

* **Organization:** Routes related to users are grouped together in user.routes.js.
* **Modularity:** The code is more modular, making it easier to maintain and extend.
* **Scalability:** As your application grows, you can add more routers (e.g., productRouter, orderRouter) and connect them to your main app with app.use().

This modular approach makes your application easier to develop, debug, and scale as it grows in complexity.