# Contents:

**Backend**

Day 1 : Creating Server in Backend in Express.js

Day 2 : Creating and Organizing Routes - Routes in Express.js

Day 3 : Creating Controllers in Express.js - Managing Application Logic

Day 4 : User Registration Using Postman - Getting User Data as Response

Day 5 : Connection Backend with MongoDB Database - Node.js and Mongoose

Day 6 : Securing your sensitive data with dotenv in our Backend

Day 7 : Creating Schema and Model for User Registration in MERN App

Day 8 : Storing Registered User Data in Online Database - MongoDB Atlas

Day 9 : Secure User Password Using bcryptjs in MERN App

Day 10 : Secure User Authentication with JWT in MERN App(JSON Web Token)

Day 11 : Building a Complete User Login Route with Functionality in MERN App

Day 12 : Building a Custom Compare Password Check Function

Day 13 : Registration and Login Validation Using ZOD in MERN App

Day 14 : A Deep Dive into Error Middleware Implementation in our MERN App

Day 15 : Building Complete Contact Form(Schema, Route, and Logic)

**Frontend**

Day 16 : Creating React App with Vite in Frontend

Day 17 : React Page Navigation with react-router-dom

Day 18 : Building Navbar with React || Open Page without Reloading

Day 19 : Registration Page and Store Form Data in React State Variables

Day 20 : Login Page and Store Login Data in React State Variable

Day 21 : CSS Styling in Login and Register Page with Tailwind CSS

Day 22 : Homepage and About Page with Tailwind CSS

Day 23 : Contact Form Data in React State Variable with Tailwind CSS

Day 24 : 404 Page - Page Not Found with Tailwind CSS

**Merger Frontend with Backend**

Day 25 : Store Registration Form Data in Database (MERN App)

Day 26 : Store Login Form Data in Database (MERN App)

Day 27 : Store JWT Token in Local Storage using Context API for Authentication

Day 28 : Toogle Login and Logout in our MERN Stack App

Day 29 : JWT Token Verification Middleware and Create Route to get User Data from Database in our MERN Stack App (Backend)

Day 30 : Authentication: Auto-fill Contact Fields with User Data from the database

Day 31 : Get the username from the database of the logged-in User on the About Page

Day 32 : Store the Contact Form Data in MERN Stack

Day 33 : Create a Service Scheme, Model, Route, and Controller in the MERN Stack

Day 34 : Create Dynamic Cards by Fetching Backend Services Data with React

Day 35 : Handling Login and Registration For Validation with React

Day 36 : Integrating React Toastify for Dynamic Alerts in React App

Day 37 : Fix the Problem of Needing Refreshing for Login and Logout

**Continuing with our Admin Panel**

Day 38 : Retrieve the user data from the database

Day 39 : Retreive the contact data from the database

Day 40 : Secure our Admin Route with JWT Verification and Middleware

Day 41 : Verifying whether user is admin or not

Day 42 : Building an Admin Page Layout - Nested Admin Routes

Day 43 : Get all the users data and show in table

Day 44 : Delete the users details on clicking Delete Button

Day 45 : Create backend by get user by id

Day 46 : Creating update page in frontend

Day 47 : Update user details in the backend and database with frontend

Day 48 : Update user in the frontend

Day 49 : Get contact form every data (frontend+backend)

Day 50 : Delete contact form data from frontend and backend

# Everything that we need in backend as simple as possible

# Day 1 : Creating a Backend Server Using Express.js

Express.js is a Node.js framework that helps to make use of Node.js to make servers easily. Creating a server with Node.js really sucks. So, here I am creating a backend server using Express.js.

### How to get started with the project?

First, you need to create a server in the backend using Express. Here I have given the code to get started with the backend server.

const express = require("express");

const app = express();

//Route for Homepage

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

res.status(200).send("Hello World");

})

const port = 3000;

app.listen(port,()=>{

console.log(`Server is running on port ${port}`);

})

Here, after I have installed the express package. I required the express package in my project and assigned it to the express variable.

Let me say this, the first line gives every power of express package to express variable. Doesn't that sound cool?

Secondly, I assigned the express() function to the variable app`.

Now, we can use the app variable to create routes and listen to the route using the server and port.

By default, the backend uses the [localhost](http://localhost/) server and we are using the port 3000.

Now, you are good to run your backend at [localhost:3000](http://localhost:3000/).

I am using the nodemon package for smooth scrolling. If you want to know about nodemon then you can [click here...](https://www.npmjs.com/package/nodemon)

### Conclusion

You will get to see a white screen with the message `Hello World` in the browser after creating a backend server using Express.js as given in the above code.

# Day 2 : Creating and Organizing Routes - Routes in Express.js

Express Routes are used for handling different kinds of tasks in the backend. These routes are called from the front end and execute the code as written inside these routers. Creating and Organizing the routes can be done with the help of the Express.Router() method.

### Creating the routes using express.Router()

First, let me tell you this is done only to make the server.js i.e. your main file clean and healthy.

Let us go with the process now.

First, you need to create a folder named router. It's not mandatory but it's a kind of convention followed by every developer to understand each other perfectly.

If it is your personal project, then you can use another name too. But let me warn you that later you are going to suffer a lot.

Here's the code to be written inside the auth-route.js

//Register and Login routes - Authentication Router

const express = require("express");

const router = express.Router();

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

res.status(200).send("Hello from auth-router.js");

});

// You can follow the either way to write the routes

router.route("/register").get((req,res)=>{

res.status(200).send("Hello from auth-router.js - Register");

})

module.exports = router;

You need to import express and assign the express.Router(). This will eventually give all the power to router variables and help in creating routes.

After that, you need to export the router and import it into the server.js file.

const express = require("express");

const app = express();

const router = require("./router/auth-router");

// Mouth Routes : To use the Express router , you need to mount it a specific path like /api/auth

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

const port = 3000;

app.listen(port,()=>{

console.log(`Server is running on port ${port}`);

})

After that, you need to mount the router into a specific url. You can keep it as you desire as far as it is understandable.

I have used /api/auth as it is a well-understood url by the developer community. And of course, these routes are API(Application Programming Interface).

Have you ever heard about REST API? If not, you need to check out by [clicking here...](https://www.redhat.com/en/topics/api/what-is-a-rest-api)

### Conclusion

Here, you have finished creating some simple routes. Creating and Organizing routes with Express.js is easy, isn't it?

# Day 3 : Creating Controllers in Express.js - Managing Application Logic

Controllers are the functions where you will write code to handle the data or let me say the incoming requests and responses for the client.

Yesterday, we learned how to create routes when starting a project. Today, will learn about creating controllers in express.js and managing application logic.

### How to create controllers?

First, let me tell you it's not a big topic to talk about. Here we are just going to manipulate yesterday's code about creating routes a little bit.

If you have not viewed that article, then you need to see it by [clicking here....](https://www.linkedin.com/pulse/creating-organizing-routes-expressjs-manoj-shrestha-5fbvc?trackingId=e2epK1cQQImNg0qrR1b5MQ%3D%3D/?trackingId=e2epK1cQQImNg0qrR1b5MQ==)

First, we will start by creating a `controller` folder inside the server folder.

Then create a file for the controller with extension \*.js and you can see the code as follows:

// Home logic

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

try{

res.status(200).send("Hello from auth-controller.js")

}catch(error){

res.status(404).send({msg:"Page not found"});

}

}

//Registration Logic

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

try{

res.status(200).send("Hello from auth-controller.js - Register");

}catch(error){

res.status(404).send({msg:"Page not found"});

}

}

module.exports = {home,register};

We have done the same thing with routes too, haven't we?

Ok! What will happen with the routes then?

Yeah, we are going to change that too.

First, we will import these functions in the routes file. and you can see the code as:

//Register and Login routes - Authentication Router

const express = require("express");

const router = express.Router();

const authControllers = require("../controllers/auth-controller");

router.route("/").get(authControllers.home);

router.route("/register").get(authControllers.register);

module.exports = router;

Now, our routes file also looks quite good and clean. And you are all good to go for the next step.

### Conclusion

Thank you for giving your time. I hope you have learned about creating controllers in Express.js and managing your application logic. And yeah, we havenot written the logic yet.

# Day 4 : User Registration Using Postman - Getting User Data as Response

Postman is a tool that helps us in testing our API and implements methods like GET, POST, PATCH, DELETE, and so on. Today, we are going to learn about user registration using Postman for getting user data as a response to the console and the user side in the form of JSON.

### How to register a user using Postman?

Remember, first you need to have Postman. You can install the application or alternatively use the Postman extension in the VS Code. I am using the Postman extension in the VS Code.

Then you need to create a collection, let us say I named it as mernadmin.

Then inside the collection, you can create various kinds of requests as mentioned above i.e. GET, POST, and others.

If we need to fetch data, then we need to use the GET method.

But today, we are going for user registration using Postman. So, we are going to use the POST method.

After creating a new request, we need to set the API for the POST method in the input box above.

Then we need to set the Headers to Content-Type: application/json.

Then we need to insert the data inside the body in the JSON format.

let us say:

{

"name" : "manoj",

"email" : "manoj@test.com"

}

After that, just click on send.

You will see that we will get empty curly braces as output. That's because we are handling the JSON data but our application doesn't know about it. We need to tell the application about it.

For that, just go to your server.js file and above your specified routes, add a line:

app.use(express.json());

This will tell the express application that we are handling the JSON data. It's called a middleware.

The main function of middleware is to modify the req and res objects, and then compile and execute any code if required.

Like here, it is modified into the JSON format by the middleware express.json.

### Conclusion

In conclusion, you have successfully learned about user registration using Postman and get user data as a response. This will be beneficial when you are inserting data into the database from the client side or front end.

# Day 5 : Connection Backend with MongoDB Database - Node.js and Mongoose

After, you create the routes and controllers for your project. Another task is to connect your backend with any database. For our MERN Stack Project, we will be using the MongoDB Database.

In this article, we will have a successful connection of the backend with MongoDB Database using Node.js and Mongoose.

Mongoose is a package npm that helps us to connect with the database easily and manage our code.

### How to connect the backend with the database?

* First, let us create a folder named **utils**and inside it a file named **db.js**. You need to write the following code:

const mongoose = require('mongoose');

const URI = 'mongodb+srv://<username>:<password>@cluster0.f2lowrm.mongodb.net/';

const connectDB = async() =>{

try{

await mongoose.connect(URI);

console.log('Database connected');

}catch(err){

console.log(err.message);

}

}

mongoose.connect(URI);

module.exports = connectDB;

For connecting with the database, you can use the local server or MongoDB Atlas.

In the above code, you can see that the MongoDB Atlas is being used to connect with the database.

That is, our database is running on the cloud in MongoDB Atlas rather than the local computer.

We can access the database from anywhere.

* Then we need to check whether the database has been connected or not. For this, we need to import the **connectDB**inside the server.js. Because our server.js runs on our backend.

const express = require("express");

const app = express();

const connectDB = require("./utils/db");

const port = 3000;

connectDB().then(()=>{

app.listen(port,()=>{

console.log(`Server is running on port ${port}`);

})

})

In the above code, the server will listen and run only after getting connected with the database.

You can see that the server is listening only after the connection successfully executes.

connectDB.then();

Now, you can see a message in your console **Database connected**if you have the correct URI used in the above code.

### Conclusion

So, you have successfully connected your backend with the database using Node.js and Mongoose.

# Day 6 : Securing your sensitive data with dotenv in our Backend

The dotenv package is used in the backend while dealing with Node.js so that you can secure your important data here.

As you might while entering the MONGODB\_URL directly into the project, creates a security problem especially while pushing the project into GitHub.

So, the dotenv is used for securing your sensitive data with dotenv in our backend.

### How to use dotenv in our backend?

* First, you need to install the dotenv package by running the following npm command :

npm i dotenv

* Then you need to create a file named .env and the name is mandatory inside the server folder. Then you can keep your sensitive and private data inside the file like:

MONGODB\_URI='mongodb+srv://<username>:<password>@cluster0.f2lowrm.mongodb.net/<database\_name>'

* Before using this data in your project, you need to import dotenv and configure your project with dotenv. Let me do it with the database connection as follows:

const mongoose = require('mongoose');

const dotenv = require('dotenv');

dotenv.config();

// using data from .env file

const URI = process.env.MONGODB\_URI;

const connectDB = async() =>{

try{

await mongoose.connect(URI);

console.log('Database connected');

}catch(err){

console.log(err.message);

process.exit(0);

}

}

mongoose.connect(URI);

module.exports = connectDB;

Now, your data is secure. You can use .gitignore for more security while pushing into GitHub or other version control platforms.

If you don't know, what GitHub is then you can [click here...](https://www.linkedin.com/pulse/what-gitignore-file-manoj-shrestha/?trackingId=JtOa9TbjRu2tRqRBYA7mYg==)

### Conclusion

Congratulations, your sensitive data is secured now.

# Day 7 : Creating Schema and Model for User Registration in MERN App

While we are using Mongoose, our first work is creating a schema and model for user registration in the MERN app.

Basically, Schemas are the blueprint of the model which includes document properties, values, defaults, validators, and so on.

Similarly, Models are the interfaces to the database that help you to create, query, update, delete records, and so on.

### How to create a Schema and Model?

* Creating a Schema and Model is very easy. First, create a folder named **models**. We are going to create both schema and model inside this folder. Then create a file named **user-model.js.**Import or require the mongoose into your file.

const mongoose = require('mongoose');

* Then you create the schema for user registration like the below:

const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({

username: {

type: String,

required: true,

},

email: {

type: String,

required: trusted,

},

phone:{

type: String,

required: true,

},

password: {

type: String,

required: true,

},

isAdmin: {

type: Boolean,

default: false,

},

});

You can see above that we have created a schema of the model. It signifies how our database would look like.

We need a new mongoose.Schema({<properties>}) is assigned to a variable so that we can use this schema for the model.

You can see we have named all the properties with their type.

Here **required**tells that those fields are mandatory.

Similarly, **default**tells that when the user won't tell about the admin then by default that value will be stored in the database.

* At last, you need to create a model with this Schema.

//define the model or collection name

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

module.exports = User;

You need to do it just below the Schema section.

Remember, we have been told that we are going to create a Schema and Model inside the same file.

Here, we use a new mongoose.model() assigned to a variable for exporting it to the main server file.

The first parameter is the collection name here inside the "" and the second parameter is the name of the variable assigned to the Schema.

### Conclusion

Now, you are ready for the registration process.

# Day 8 : Storing Registered User Data in Online Database - MongoDB Atlas

Here, we will be storing registered user data in an online database called MongoDB Atlas.

I know that we haven't made a front-end form or anything like that. So, we will register user data with the help of a tool called Postman.

### How to store registered user data in the online database?

* The data sent to the server from the form lies inside the req.body.First, let us destructure the data.

const {username, email, phone, password} = req.body;

* Let us check whether the email is already registered or not.

const userExist = await User.findOne({email : req.body.email});

Show some message if the user already exists with the email.

if(userExist){

return res.status(400).json({msg:"Email already exists"});

}

* Let us create a new user now.

await User.create({username, email, phone, password});

As a whole our code inside the register method of auth-controller.js looks like this.

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

try {

// console.log(req.body);

//Check whether the email exists or not in the database already

const {username, email, phone, password} = req.body;

const userExist = await User.findOne({email:req.body.email});

if(userExist){

return res.status(400).json({msg:"Email already exists"});

}

const user = await User.create({username,email,phone,password});

res.status(200).json({ user});

} catch (error) {

res.status(404).json({ msg: "Page not found" });

}

};

### Conclusion

Congrats, you have successfully created the logic for storing registered user data in the online database called MongoDB Atlas.

# Day 9 : Secure User Password Using bcryptjs in MERN App

The Bcrypt is a package that helps to hash the password to make it more secure.

### How to secure user passwords using bcryptjs in the MERN App?

Here, you have two ways to implement the method to secure the password.

1. First, you can directly implement it inside the controller as logic.

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

try {

const { username, email, phone, password } = req.body;

const userExist = await User.findOne({ email: req.body.email });

if (userExist) {

return res.status(400).json({ msg: "Email already exist" });

}

const saltRounds = 10;

const hashed\_password = await bcrypt(req.body.password,saltRounds);

const user = await User.create({

username,

email,

phone,

password : hashed\_password,

});

res.status(201).json({ user });

} catch (err) {

res.status(400).json({ msg: "Page not found" });

}

};

Here, you create the logic inside the controllers which makes our code look bad. So, there's another method too.

2. Second, you can implement just before the model using pre() method.

//secure the password using bcrypt.js

userSchema.pre("save", async function(next){

const user = this;

if (!user.isModified("password")) {

return next();

}

try {

const hashedPassword = await bcrypt.genSalt(10).then((salt) => {

return bcrypt.hash(user.password, salt);

});

user.password = hashedPassword;

next();

} catch (err) {

next(err);

}

});

//define the model or collection name

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

module.exports = User;

Below the schema and above the model, you can hash the password.

Basically, it will hash the password before the password is saved into the database. So, we have done it before the model.

So that it will execute first with a pre-method. The first parameter is mandatory i.e save method.

The second method is always a normal async function.

### Conclusion

Congrats, you have secured the user password using bcrytpjs in MERN app.

# Day 10 : Secure User Authentication with JWT in MERN App(JSON Web Token)

JWT (JSON Web Token) is a npm package that helps us in the authentication and authorization of the user. Here, we will see how to secure user authentication with JWT in the MERN app.

### How to generate JWT (JSON Web Token) for a user?

* First, you need to install the jsonwebtoken package inside the server folder.

npm i jsonwebtoken

* Then you need to create a token while registering for the user.

userSchema.methods.generateToken = async function () {

try {

return jwt.sign(

{

userId: this.\_id.toString(),

email: this.email,

isAdmin: this.isAdmin,

},

process.env.JWT\_SECRET\_KEY,

{

expiresIn: "30d",

}

);

} catch (err) {

console.error(err);

}

};

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

module.exports = User;

Just above the model, we have added the token into the userSchema by the use of jwt.sign() method.

Remember jwt.sign() methods have two mandatory fields Payload and secret key while the other is optional.

**Payload** contains the data which needs to be stored in the token.

**Secret key** is used to sign the token. It is stored in the environment variables.

**Settings**, the third parameter which is optional lets you keep different settings to the token. Here, we have used expiresIn:"30d" which sets the token to expire in 30 days.

Now, when the user is created automatically it will create a token for the user which can be seen by the use of a controller.

When returning the message with registration successful or user data in the controller during registration, you can return the token also.

res

.status(201)

.json({

user,

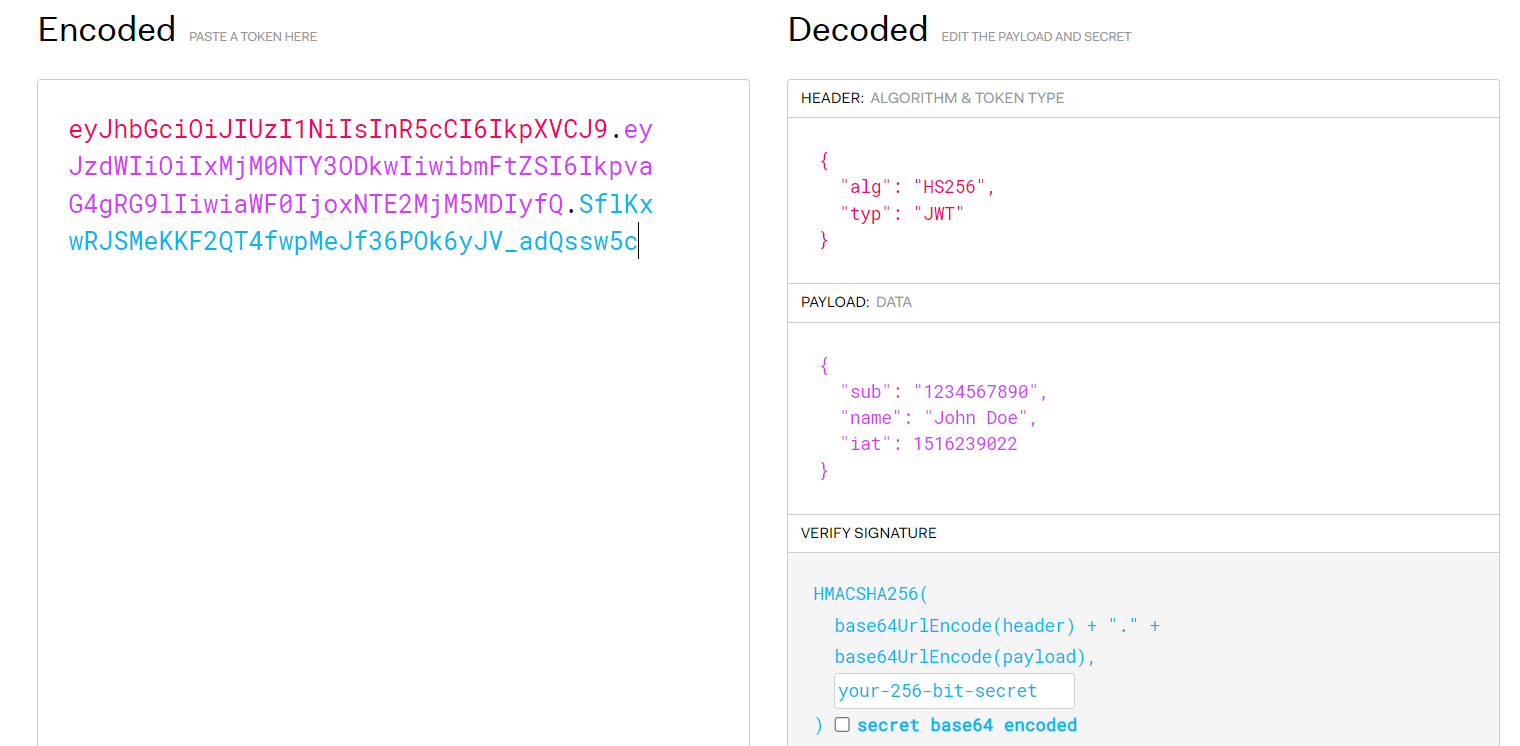
token: await user.generateToken(),

userId: user.\_id.toString(),

});

Remember, the jsonwebtoken deals with the string. So, you must have to convert the id to a string while creating a token in the payload.

You can see the structure of json web token as follows:



Structure of JSON Web Token (JWT)

**Header** consists of a type of token which is JWT and the signing algorithm to be used.

**Payload** consists of claims which are generally the user data and additional data.

There are three types of claims: registered, public and private claims.

**Signature** is used to verify the sender of the JWT. It is created by encoded headers, the encoded payload, a secret key, and using the algorithm specified in the header to sign these.

### Conclusion

Congratulations, you have created JSON web token for the user registered. Now, you can see the JSON web token while registering the user.

# Day 11 : Building a Complete User Login Route with Functionality in MERN App

Today, we will create a login route for our project in the backend.

### How to build a complete user login route with functionality in MERN app?

* First, we need to create routes for the login in the auth-router.js file inside the routes folder.

router.route("/login").post(authControllers.login);

Add this route to your auth-router.js file at the top just before you export the router.

* Now, create a method register inside the auth-controller.js file. Implement the following logic:

//Login Logic

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

try {

const { email, password } = req.body;

const userExist = await User.findOne({ email : req.body.email });

if (!userExist) {

return res.status(400).json({ message: "Invalid Credentials" });

}

const isMatch = await bcrypt.compare(password, userExist.password);

if (isMatch) {

res.status(200).json({

msg: "Login Successful",

token: await userExist.generateToken(),

userId: userExist.\_id.toString(),

});

} else {

res.status(401).json({ message: "Invalid email or password" });

}

} catch (err) {

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

}

};

You can see an asynchronous function login used as a route handler for the login endpoint.

The function begins by destructuring the email and password from the request body (req.body).

Next, it has used user.findOne({email : [req.body.email](http://req.body.email/" \t "https://www.linkedin.com/pulse/building-complete-user-login-route-functionality-mern-manoj-shrestha-efkbc%3FtrackingId=2YxxbyjcuixDVguaN28GzA%253D%253D/_self)}) or user.findOne({email}) to find whether any user exists or not with the same email name.

If the user doesn't exist, then the method will return with the message Invalid Credentials.

If an email is found or a user is found, then it compares the user's entered password with the database hashed password.

If the password doesn't match, then it displays a message with "Invalid email or password".

Otherwise, it will display the message "Login Successful" with the token and id.

### Conclusion

Congrats, you have successfully made user login route for logging into the system.

# Day 12 : Building a Custom Compare Password Check Function

Yesterday, we saw that we used the [bcrypt.compare](http://bcrypt.compare/" \t "https://www.linkedin.com/pulse/building-custom-compare-password-check-function-manoj-shrestha-6xf2c%3FtrackingId=yBsckdSMR82Pe2NMh9KXrA%253D%253D/_self)() inside the controller which made our code look dirty.

So, today we are going to compare the password inside our schema creating a custom method.

### How to build a custom compare password check function?

* You need to create an async function in your userSchema

userSchema.methods.isValidPassword = async function(password){

try{

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

}catch(err){

throw err;

}

}

This will take the user-entered password as a password and compare the password with the password inside the userExist variable with the help of this.password.

* Now, we will call this inside our controller instead of [bcrypt.compare](http://bcrypt.compare/" \t "https://www.linkedin.com/pulse/building-custom-compare-password-check-function-manoj-shrestha-6xf2c%3FtrackingId=yBsckdSMR82Pe2NMh9KXrA%253D%253D/_self) by:

const isMatch = await userExist.isValidPassword(password);

Where the password in the parameter is the password entered by the user present inside the req.body.

### Conclusion

Congrats, you have made a custom method for password check or password validation.

# Day 13 : Registration and Login Validation Using ZOD in MERN App

ZOD is a schema declaration and validation library for typescript. Here, we going to do registration and login validation using ZOD in the MERN app.

### How to implement ZOD Validation?

* Inside validators, First, create a validation schema for the form you want. Here, I am creating for signup form as signupSchema.

const { z } = require("zod");

//Creating an object schema

const signupSchema = z.object({

username: z

.string({ required\_error: "Username is required" })

.trim()

.min(3, {

message: "Username must be at least 3 chars",

})

.max(255, { message: "Username must not be more than 255 chars long" }),

email: z

.string({ required\_error: "Email is required" })

.trim()

.min(3, {

message: "Email must be at least 3 chars",

})

.max(255, { message: "Email must not be more than 255 chars long" }),

phone: z

.string({ required\_error: "Phone number is required" })

.trim()

.min(3, {

message: "Phone number must be at least 10 chars",

})

.max(255, { message: "Phone number must not be more than 20 chars long" }),

password: z

.string({ required\_error: "Password is required" })

.trim()

.min(3, {

message: "Password must be at least 6 chars",

})

.max(255, { message: "Password must not be more than 1024 chars long" }),

});

module.exports = signupSchema;

Let me break the code:

1. First, we import zod and assign it to the z variable. Here, {z} means that it is extracting the zod object from the zod library.
2. Then we create a schema using z.object({}).
3. Similar to the actual schema, we are using the same key like username, email, and others.
4. z.string() defines that the value must be a string.
5. {require\_error : "Username is required"} checks whether the field is empty or not.
6. .trim() will trim the field to convert it into the array.
7. .min(value, {msg:""}) checks the minimum limit.
8. .max(value,{msg:""}) checks the maximum limit.

* Inside the middleware, now we need to create an error middleware for validation.

const validate = (schema) => async (req, res, next) => {

try {

const parseBody = await schema.parseAsync(req.body);

req.body = parseBody;

next();

} catch (err) {

res.status(400).json({ msg: err.issues[0].message });

}

};

module.exports = validate;

Here, we have created a validation function that takes a zod schema for validation.

Then we pass an async function with req for the request object, and res for the response object, next to move to the next middleware.

await schema.parseAsync(req.body) checks for the validation.

And after validating it will again assign the value to req.body.

But our work doesn't end here. We have to implement it in our project.

* Inside auth-router.js we are going to import the auth-validator and validator-middleware.

const validate = require("../middlewares/validate-middleware");

const signupSchema = require("../validators/auth-validator");

Then we need to pass it as a parameter to register the route so that it will be validated before implementing the auth-contollers.register controller logic.

router.route("/register").post(validate(signupSchema),authControllers.register);

### Conclusion

Now, your register form or signup form contains validation. Tomorrow, we will deeply dive into error validation.

# Day 14 : A Deep Dive into Error Middleware Implementation in our MERN App

Error middleware helps us to handle the error from a single file.

### How to set up the error middleware function?

* First, we create a error-middleware.js file inside the middleware folder.

const errorMiddleware = (err,req,res,next) =>{

const status = err.status || 500;

const message = err.message || "BACKEND ERROR";

const extraDetails = err.extraDetails || "Error from backend";

return res.status(status).json({message,extraDetails});

}

module.exports = errorMiddleware;

Here, we require an extra parameter **error** for catching errors. You can see it store status, messages, and extra details obtained from different locations inside the backend.

* In every function inside the backend while catching the error, we must have to include next(err) where err is a parameter that contains err status, message, and extra details (optional).

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

try {

res.status(200).json({ msg: "Hello from auth-controller.js" });

} catch (error) {

// res.status(404).json({ msg: "Page not found" });

next(err);

}

};

* Don't forget to mention about this middleware in the server.js file.

const errorMiddleware = require("./middlewares/error-middleware");

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

app.use(errorMiddleware);

const port = 3000;

connectDB().then(()=>{

app.listen(port,()=>{

console.log(`Server is running on port ${port}`);

})

})

### Conclusion

After including err in every location, you are ready to handle errors by using error middleware. This is given itself by express js.

# Day 15 : Building Complete Contact Form(Schema, Route, and Logic)

Now, here we will build a complete contact form i.e. Schema, Route, and Logic.

### How to build a complete contact form?

* First, let us build a Schema and model for the **contact form.**

const { Schema, model, default: mongoose } = require("mongoose");

const contactSchema = new Schema({

username: { type: String, required: true },

email: { type: String, required: true },

message: { type: String, required: true },

});

//create a model or collection

const Contact = new model("contact",contactSchema);

module.exports = Contact;

We have made this schema inside models the folder with the name contact-model.js.

Now, we need to create logic or controller using this schema to insert the data into the database.

const Contact = require('../models/contact-model');

//Contact Logic

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

try{

const response = req.body;

await Contact.create(response);

return res.status(200).json({message : "message send successfully"});

}catch(err){

res.status(400).json({message : "Cannot submit the contact form"});

}

}

module.exports = contactForm;

Here, we have created a method contactForm for writing logic for inserting data into the document.

* Then let us create a route inside the router folder for handling this logic.

const express =require('express');

const router = express.Router();

const contactForm = require("../controllers/contact-controller");

router.route("/contact").post(contactForm);

module.exports = router;

Here, we have created a route called /contact for handling logic.

* Lastly, we need to set the server.js file.

const contactRoute = require('./router/contact-router');

app.use("/api/form",contactRoute);

Now, you are all set up and ready for testing through the backend.

{

"username" : "Manoj Shrestha",

"email" : "admin@test.com",

"message" : "I am fine"

}

Suppose you post this data to /api/form/contact , then you are going to get the result:

{

"message": "message send successfully"

}

If you check your database, then you are going to get the data inside your database.

### Conclusion

Congrats, you have successfully created functionality for the contact form in the backend.

# Everything that we need in frontend as simple as possible

# Day 16 : Creating React App with Vite in Frontend

After the successful completion of our backend codes and testing, now we are moving a step forward with creating a React App with Vite in the front end.

### How to get started with our React App?

* Here, we are going to use the Vite environment for creating our React application.

npm create vite@latest

Enter the following command in your main project and hit Enter.

* Enter the name of the front-end project or folder. let's say **client.**
* Then give the name of the package. let us say, **client.**
* Then with the help of the arrow key select **React** as a framework.
* Similarly, select **JavaScript** as a variant.
* After that, type the following command.

cd client

This lets you enter into the frontend project folder.

* Type the following command to install all packages from package.json.

npm install

* In order to run the project, enter the following command after the packages have been installed.

npm run dev

### More about the frontend default folder setup

**node\_modules** contains the codes of all the packages.

**public** contains just vite.svg file only.

**src** contains the components.

All the work is done in the src/App.jsx.

const App = () =>{

return (<h1>Our Default HomePage</h1>)

}

export default App;

Do this as your first code in React.

### Conclusion

Congratulations, you have successfully created a React App with Vite in the front end.

# Day 17 : React Page Navigation with react-router-dom

Before going further into our topics, we have to deal with the react page navigation with react-router-dom.

After this setup is completed we are all free to create designs.

### How to create React Page Navigation?

* First, we need to install a package called react-router-dom.

npm i react-router-dom

* In order to create page navigation we need to export {BrowserRouter, Routes, Route} from the react-router-dom.

import {BrowserRouter, Routes, Route} from 'react-router-dom';

* Then we need to create paths for different pages.

const App = () =>{

return (<>

<BrowserRouter>

<Routes>

<Route exact path="/" element={<Home />} />

<Route path='/about' element={<About />}/>

<Route path='/contact' element={<Contact />}/>

<Route path='/service' element={<Service />}/>

<Route path='/register' element={<Register />}/>

<Route path='/login' element={<Login />}/>

<Route path='/\*' element={<NotFound />} />

</Routes>

</BrowserRouter>

</>)

}

Here, BrowserRouter is the main heading component with Routes for setting up different routes.

Route helps to set individual routes for different pages.

Here, at last, we have created a 404 page also for handling errors.

You can check this by going to different paths.

localhost:5173/login

* Make the pages inside the src/pages

pages

-Home.jsx

-Contact.jsx

and other pages like this.

### Conclusion

Now, you have successfully created react page navigation to navigate to different pages.

# Day 18 : Building Navbar with React || Open Page without Reloading

Today, we are going to build a navbar with React where we will try to open the page without reloading.

### How to build a navbar with React that doesn't reload the page?

* First, let us create a navbar component:

import React from "react";

import { NavLink } from "react-router-dom";

const Navbar = () => {

return (

<>

<header>

<div className="container">

<div className="logo-brand">

<a to="/">Manoj</a>

</div>

</div>

<nav>

<ul>

<li>

<NavLink to="/">Home</NavLink>

</li>

<li>

<NavLink to="/about">About</NavLink>

</li>

<li>

<NavLink to="/service">Services</NavLink>

</li>

<li>

<NavLink to="/contact">Contact</NavLink>

</li>

<li>

<NavLink to="/register">Register</NavLink>

</li>

<li>

<NavLink to="/login">Log In</NavLink>

</li>

</ul>

</nav>

</header>

</>

);

};

export default Navbar;

In order to make components, make src/components and create a file named as Navbar.jsx inside it.

The above code helps to create navbar components.

In order to create a link which doesn't reload the page, you need to use <NavLink to="/path">name</NavLink>.

* After that, all the work is done on the App.jsx.

You can call the navbar on each of the pages individually which takes a lot of time. So for such a reusable component, you can directly call it inside App.jsx.

import Navbar from './components/Navbar';

* After this, you can call this component inside BrowserRouter before handling the Routes.

<BrowserRouter>

// Navbar is shown before other paths are handled. So it can be seen in every pages.

<Navbar />

<Routes>

<Route exact path="/" element={<Home />} />

<Route path='/about' element={<About />}/>

<Route path='/contact' element={<Contact />}/>

<Route path='/service' element={<Service />}/>

<Route path='/register' element={<Register />}/>

<Route path='/login' element={<Login />}/>

<Route path='/\*' element={<NotFound />} />

</Routes>

</BrowserRouter>

### Conclusion

Congrats, you have successfully created a navbar with React that opens pages without reloading.

Remember if you use <a> tag instead of <NavLink> then the page gets reloaded.

# Day 19 : Registration Page and Store Form Data in React State Variable

Let us create a registration page and try to store form data in the React state variable so that we can send the data in the backend for storing.

### How to store register form data in the React state variable?

* First, we need to create a register form as you all know.

<form action="" onSubmit={handleSubmit}>

<div>

<label htmlFor="username">username</label>

<input

type="text"

name="username"

value={user.username}

placeholder="username"

id="username"

required

autoComplete="off"

onChange={handleInput}

/>

</div>

<div>

<label htmlFor="email">email</label>

<input

type="email"

name="email"

value={user.email}

placeholder="email"

id="email"

required

autoComplete="off"

onChange={handleInput}

/>

</div>

<div>

<label htmlFor="phone">phone</label>

<input

type="number"

name="phone"

value={user.phone}

placeholder="phone"

id="phone"

required

autoComplete="off"

onChange={handleInput}

/>

</div>

<div>

<label htmlFor="password">password</label>

<input

type="password"

name="password"

value={user.password}

placeholder="password"

id="password"

required

autoComplete="off"

onChange={handleInput}

/>

</div>

<button type="submit">Register Now</button>

</form>

Then let us write the logic for storing these data in the React state variable.

* Can you see their name attribute in all the above input boxes? That should be similar to the data you need to catch in the backend. It creates a similarity between the code sent from the front end. We have already seen these while sending data through the Postman API testing tool.
* First, let us handle the form-submit event.

//Handling Form Submission

const handleSubmit = (e) => {

e.preventDefault();

console.log(user);

};

You can see we have prevented the form from reloading on submitting which is quite popular while you work with form in React.

* Now, let us create a useState for handling the form data.

const [user, setUser] = useState({

username: "",

email: "",

phone: "",

password: "",

});

* Now, let us handle the change in the input box created in the form.

//Handling the input value

const handleInput = (e) => {

let name = e.target.name;

let value = e.target.value;

setUser({

...user,

[name]: value,

});

};

We have used ...user spread operator for getting previous data and changing the name dynamically so we can store data in the useState dynamically.

### Conclusion

Congrats, you have successfully created a registration form in React and stored the form data in the React State variable.

# Day 20 : Login Page and Store Login Data in React State Variable

Let us create a login page and try to store form data in the React state variable so that we can send the data to the backend for storage.

### How to store login form data in the React state variable?

* First, we need to create a login form as you all know.

<form action="" onSubmit={handleSubmit}>

'

<div>

<label htmlFor="email">Email</label>

<input

type="email"

name="email"

placeholder="Enter email"

value={user.email}

required

autoComplete="off"

onChange={handleChange}

/>

</div>

<div>

<label htmlFor="password">Password</label>

<input

type="password"

name="password"

placeholder="Enter password"

value={user.password}

required

autoComplete="off"

onChange={handleChange}

/>

</div>

<button type="submit">Log In</button>

</form>

Then let us write the logic for storing these data in the React state variable.

* Can you see their name attribute in all the above input boxes? That should be similar to the data you need to catch in the backend. It creates a similarity between the code sent from the front end. We have already seen these while sending data through the Postman API testing tool.
* First, let us handle the form-submit event.

// Handling the form submission

const handleSubmit = (e) =>{

e.preventDefault();

console.log(user);

}

You can see we have prevented the form from reloading on submitting which is quite popular while you work with form in React.

* Now, let us create a useState for handling the form data.

const [user, setUser] = useState({

email: "",

password: "",

});

* Now, let us handle the change in the input box created in the form.

// Handling the input value

const handleChange = (e) => {

setUser({

...user,

[e.target.name]: e.target.value,

});

};

We have used ...user spread operator for getting previous data and changing the name dynamically so we can store data in the useState dynamically.

### Conclusion

Congrats, you have successfully created a login form in React and stored the form data in the React State variable.

# Day 21 : CSS Styling in Login and Register page with Tailwind CSS

Today, we styled the login and register page with Tailwind CSS. Here, we will learn how to get started:

### How to get started with Tailwind CSS?

* First we need to **install Tailwind CSS.**

npm install -D tailwindcss@latest postcss@latest autoprefixer@latest

* Then **configure the Tailwind CSS.**

npx tailwindcss init -p

* Then import tailwind CSS into the React project.In your index.css file (or whichever global CSS file you're using), import Tailwind's styles:

@import 'tailwindcss/base';

@import 'tailwindcss/components';

@import 'tailwindcss/utilities';

* After that include CSS in React. Make sure you're importing the CSS file in your main index.jsx (or main.jsx) file:

import './index.css';

* Then purge the configuration in tailwind.config.js

/\*\* @type {import('tailwindcss').Config} \*/

export default {

content: [

"./index.html",

"./src/\*\*/\*.{js,ts,jsx,tsx}", ],

// purge: ['./index.html', './src/\*\*/\*.{vue,js,ts,jsx,tsx}'],

theme: {

extend: {},

},

plugins: [],

}

* Don't forget to restart the server.

### Conclusion

Now, you are all ready to use **Tailwind classes** in your project in the front end.

# Day 22 : Homepage and About Page with Tailwind CSS

Here we are going to look after the home page and about page.

### How to make a homepage and an About page?

It is just a frontend part. So, I am going to skip the explanation part.

import React from "react";

const Home = () => {

return (

<>

<section className="flex justify-center items-center p-3">

<div className="flex flex-col justify-center px-10">

<h3 className="text-xl">Welcome Everyone to,</h3>

<h1 className="text-5xl">All Round Campus</h1>

<p className="text-sm py-4">Lorem ipsum dolor sit amet consectetur adipisicing elit. Aperiam perspiciatis eos inventore! Sint, ipsa consequatur.</p>

<div className="flex gap-6">

<button className="bg-blue-400 px-4 py-2 text-white">Connect Now</button>

<button className="bg-blue-400 px-4 py-2 text-white">Register</button>

</div>

</div>

<div className="w-1/2">

<img

src="https://images.unsplash.com/photo-1546198632-9ef6368bef12?q=80&w=2070&auto=format&fit=crop&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D"

alt=""

className="rounded-lg"

/>

</div>

</section>

</>

);

};

export default Home;

### Conclusion

You can practice frontend on your own.

# Day 23 : Contact Form Data in React State Variable with Tailwind CSS

We have dealt with this part earlier too. So, I won't further discuss about it.

### How to keep contact form data in the React state variable?

import React, { useState } from "react";

const Contact = () => {

const [contact, setcontact] = useState({

username: "",

email: "",

message: "",

});

const handleChange = (e) => {

setcontact({ ...contact, [e.target.name]: e.target.value });

};

const handleSubmit = (e) => {

e.preventDefault();

console.log(contact);

};

return (

<>

<section className="flex p-4 gap-10">

<div className="w-1/2">

<iframe

src="https://www.google.com/maps/embed?pb=!1m18!1m12!1m3!1d112754.45609277317!2d84.19801288756317!3d27.976245661001936!2m3!1f0!2f0!3f0!3m2!1i1024!2i768!4f13.1!3m3!1m2!1s0x39950780deaf2177%3A0x33b3696137fa9dfb!2sByas!5e0!3m2!1sen!2snp!4v1701696801582!5m2!1sen!2snp"

width="600"

height="450"

allowFullScreen

loading="lazy"

referrerPolicy="no-referrer-when-downgrade"

></iframe>

</div>

<form onSubmit={handleSubmit} className="flex flex-col gap-3 px-10">

<h1 className="text-4xl">Contact Form</h1>

<div className="flex flex-col gap-2">

<label htmlFor="username">Username</label>

<input

type="text"

name="username"

value={contact.username}

onChange={handleChange}

className="border-2 border-b-blue-400"

required

/>

</div>

<div className="flex flex-col gap-2">

<label htmlFor="email">Email</label>

<input

type="email"

name="email"

value={contact.email}

onChange={handleChange}

className="border-2 border-b-blue-400"

required

/>

</div>

<div className="flex flex-col gap-2">

<label htmlFor="message">Message</label>

<textarea

name="message"

value={contact.message}

onChange={handleChange}

cols="30"

rows="10"

className="border-2 border-b-blue-400"

required

></textarea>

</div>

<button

type="submit"

className="bg-blue-400 text-white px-4 py-2 rounded-lg"

>

Submit

</button>

</form>

</section>

</>

);

};

export default Contact;

### Conclusion

Finally, we have completed the frontend part. Now, let's make interaction between frontend and backend.

# Day 24 : 404 Page - Page Not Found with Tailwind CSS

404 Page is very important when it comes to handling user requests in the front end. We don't want users to have a bad experience.

### How to create a 404 Page in frontend?

Creating a 404 page in the frontend is not a big deal. Let me just use the following code in the PageNotFound.jsx

import React from "react";

const NotFound = () => {

return (

<>

<section className="flex justify-center items-center flex-col h-screen text-gray-400">

<h1 className="text-4xl">Page Not Found</h1>

<h3 className="text-sm font-bold">You have entered invalid URL.</h3>

</section>

</>

);

};

export default NotFound;

And import it in App.jsx.

import React from 'react';

import {BrowserRouter, Routes, Route} from 'react-router-dom';

import Home from './pages/Home';

import About from './pages/About';

import Contact from './pages/Contact';

import Service from './pages/Service';

import Register from './pages/Register';

import Login from './pages/Login';

import NotFound from './pages/NotFound';

import Navbar from './components/Navbar';

import Footer from './components/Footer';

const App = () =>{

return (<>

<BrowserRouter>

<Navbar />

<Routes>

<Route exact path="/" element={<Home />} />

<Route path='/about' element={<About />}/>

<Route path='/contact' element={<Contact />}/>

<Route path='/service' element={<Service />}/>

<Route path='/register' element={<Register />}/>

<Route path='/login' element={<Login />}/>

<Route path='/\*' element={<NotFound />} />

</Routes>

<Footer />

</BrowserRouter>

</>)

}

export default App;

### Conclusion

Now, you are all ready to connect the front end with the back end.

# Everything that we need in merging frontend with backend

# Day 25 : Store Registration Form Data in Database (MERN App)

Now, we are finally going to connect the front end with the back end. let us start by storing registration form data in the database in our MERN App.

### How to store registration form data in the database in the MERN App?

* We are going to store the data which is inside the useState variable i.e. user in the database. For this, we will write the logic inside the handleSubmit() function which is called on submitting the form.

//Handling Form Submission

const handleSubmit = async (e) => {

e.preventDefault();

console.log(user);

try {

const response = await fetch("http://localhost:3000/api/auth/register", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify(user),

});

if (response.ok) {

setUser({

username: "",

email: "",

phone: "",

password: "",

});

navigate('/login');

}

} catch (err) {

console.log("Register: ", err);

}

};

Here, the major part is the fetch() function. It takes two parameters.

1. The URL of the location lies at the backend.
2. The second parameter is an object. i. First, we must specify the method.ii. Then we must specify the headers.iii. we must pass the body in JSON format through JSON.stringify(user).
3. After the response.ok is true. we can give a message or go to a different location as required.**useNavigate is a method inside react-router-dom which helps to navigate from one page to another in ReactJS apart from regular HTML.**

* Due to this, you might get a message about the CORS policy. This will happen because your front end and back end are running on different ports.**Cross-Origin Resource Sharing (**[CORS](https://developer.mozilla.org/en-US/docs/Glossary/CORS" \t "https://www.linkedin.com/pulse/store-registration-form-data-database-mern-app-manoj-shrestha-bhbpc%3FtrackingId=rhQc2buhfg9SXxAVUIwFJw%253D%253D/_self)**) is an HTTP-header-based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.**So you need to install CORS in the backend to specify the origin.

npm i cors

Then you need to set up the cors in your server.js.

const cors = require('cors');

const corsOptions = {

origin : "http://localhost:5173",

methods : "GET, POST, PUT, DELETE, PATCH, HEAD",

credentials : true

}

//should have to put due to CORS policy

app.use(cors(corsOptions));

### Conclusion

Now you are all set up to send data from the front end to the back end for storing in the database.

# Day 26 : Store Login Form Data in Database (MERN App)

Here, you will get to see how to store login form data in the database in the MERN App.

### How to store login form data in the database in the MERN App?

* It is similar to what we have done yesterday. We are going to write the whole logic inside the handleSubmit() function which is called for submitting the form.

import {useNavigate} from 'react-router-dom';

const navigate = useNavigate();

//Handling Form Submission

const handleSubmit = async (e) => {

e.preventDefault();

try {

const response = await fetch("http://localhost:3000/api/auth/register", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify(user),

});

if (response.ok) {

setUser({

username: "",

email: "",

phone: "",

password: "",

});

navigate('/login');

}

} catch (err) {

console.log("Register: ", err);

}

};

### Conclusion

Now, you can successfully log in into the system.

# Day 27 : Store JWT Token in Local Storage using Context API for Authentication

We created JWT Token in the backend using jsonwebtoken npm package. Now we need to get it into the frontend so that we can authenticate the user.

For this, we can store jwt token in the localStorage.

### How to store JWT Token in Local Storage using Context API for authentication?

* First, we need to create a context API that can be used in different routes in our project.

import { createContext, useContext } from "react";

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {

const storeTokenInLS = (server\_token) => {

return localStorage.setItem("token", server\_token);

};

return (

<AuthContext.Provider value={{ storeTokenInLS }}>

{children}

</AuthContext.Provider>

);

};

export const useAuth = () =>{

const authContextValue = useContext(AuthContext);

if(!authContextValue){

throw new Error("useAuth used outside of the Provider");

}

return authContextValue;

}

First, the createContext() function is being used from the react. This is used to create a new context. Then it is assigned to AuthContext.

This context will allow our data to be shared among all the components in the component tree.

The **AuthProvider** component is a context provider for **AuthContext.**

It takes children as a prop and returns the **AuthContext.Provider** with its children while the value prop is an object that contains a **storeTokenInLS** function.

This function takes **server\_token** as an argument and stores it in local storage.

This means all the child components of **AuthProvider** will have access to this function via the context.

The **useAuth** is a custom hook that uses **useContext** hook from react to access the **AuthContext.**

In summary, this code provides a way to share **storeTokenInLS** function among all components.

* Then, to use this context API. We need to wrap our <App /> inside the AuthProvider.

import {AuthProvider} from "./store/auth.jsx";

ReactDOM.createRoot(document.getElementById("root")).render(

<AuthProvider>

<React.StrictMode>

<App />

</React.StrictMode>

</AuthProvider>

);

You need to do this so that you can use value from context API in all you projects components.

Now, finally, you can use it across every component.

import {useAuth} from '../store/auth';

const {storeTokenInLS} = useAuth();

const res\_data = await response.json();

storeTokenInLS(res\_data.token);

### Conclusion

Congrats, your JWT token is stored successfully inside the localStorage. You can check out this by inspecting and going to **Application**.

# Day 28 : Toggle Login and Logout in our MERN Stack App

Here, we will try to toggle the login and logout functionality in our application using the Context API again.

### How do I toggle login and log out into our MERN Stack App?

* Create a Route to /logout in the app.jsx.

<Route path='/logout' element={<Logout />} />

* Create a logout page so that it will navigate to /login the page upon calling it.

import React, { useEffect } from 'react';

import { Navigate } from 'react-router-dom';

import { useAuth } from '../store/auth';

const Logout = () => {

const {LogoutUser} = useAuth();

useEffect(()=>{

LogoutUser();

},[LogoutUser]);

return (

<Navigate to="/login" />

)

}

export default Logout

* Now, you need to create a method named as LogoutUser in the context of API.

import { createContext, useContext, useState } from "react";

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {

const[token, setToken] = useState(localStorage.getItem("token"));

const storeTokenInLS = (server\_token) => {

return localStorage.setItem("token", server\_token);

};

let isLoggedIn = !!token;

const LogoutUser = () =>{

setToken("");

return localStorage.removeItem("token");

}

return (

<AuthContext.Provider value={{ isLoggedIn, storeTokenInLS, LogoutUser }}>

{children}

</AuthContext.Provider>

);

};

To handle the changes in the token, you need to create a useState for the token initialized as,localStorage.getItem("token") which will be null by default.

Then we created a function that will set the token to empty whenever it is called. The function will be called while calling the Logout Page component.

After that we not only set the token empty. We should also remove the token from the localStorage.

Then we will pass every method and value to AuthContext.Provider.

* At last, we need to render the Navbar according to the value of isLoggedIn in Navbar.jsx.

import { useAuth } from "../store/auth";

const { isLoggedIn } = useAuth();

{isLoggedIn ? (

<li>

<NavLink to="/logout">LogOut</NavLink>

</li>

) : (

<>

<li>

<NavLink to="/register">Register</NavLink>

</li>

<li>

<NavLink to="/login">LogIn</NavLink>

</li>

</>

)}

Now, your login and logout functionality will run properly. There will occur a simple issue about toggling login and logout in the React app.

We will handle the issue later.

### Conclusion

Now, you have successfully created the toggle between login and log-out functionality.

# Day 29 : JWT Token Verification Middleware and Create Route to get User Data from Database in our MERN Stack App (Backend)

Now, again, we will work in the backend and create a route to get user data from the database based on the jwt token verification middleware.

### How to create JWT token verification middleware?

* First, let us create a method named user inside the controllers for writing logic.
* Let us create a route for getting the user data from the backend into the frontend.

router.route('/user').get(authControllers.user);

* Now, let us get the user data from the request and send the user data into the frontend.

// To Send User Data - User Logic

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

try{

const userData = req.user;

console.log(userData);

return res.status(200).json({msg : userData});

}catch(err){

next(err);

}

}

Now, the question comes here. Where is the req.user coming from?

Let me clear you that it will come from the middleware.

So, let us create a middleware named auth-middleware.js inside the middleware folder.

* In middleware, we will get the token through the header and then verify it with jwt to get the data of the user which has been signed to the jwt at the beginning.Then after getting all the data, we will exclude the password so that it won't be sent to the frontend.

const jwt = require('jsonwebtoken');

const User = require('../models/user-model');

const authMiddleware = async(req,res,next) => {

const token = req.header(authorization);

if(!token){

return res.status(401).json({msg: "No token, authorization denied"})

}

// Assuming token is in the format of Bearer <token>

try{

const jwtToken = token.split(" ")[1];

const isVerified = jwt.verify(jwtToken,process.env.JWT\_SECRET\_KEY);

const userData = await User.findOne({email: isVerified.email}).select("-password");

req.user = userData;

req.token = token;

req.userId = userData.\_id.toString();

next();

}catch(err){

return res.status(401).json({msg: "Token is not valid"})

}

}

module.exports = authMiddleware;

### Conclusion

Congratulations, you have successfully verified the jwt token and created a route to get user data from the database in the backend.

# Day 30 : Authentication: Auto-fill Contact Fields with User Data from the database

Yesterday, we created a route in the backend for authentication. Now, we are going to use the authenticated data on the front end.

### How to auto-fill contact fields with user data from the database?

* First, let us create a state variable that will store the user data that has been logged in by us. As more than one user can log in simultaneously to the system, we need to handle the users with the help of state variables.

const [user, setUser] = useState("");

* Let us create a method that will get data from the backend on authentication with the help of bearer token authorization.

// JWT Authentication - To get the currently logged in user data

const userAuthentication = async () => {

try {

const response = await fetch("http://localhost:3000/api/auth/user", {

method: "GET",

headers: {

Authorization: `Bearer ${token}`,

},

});

if (response.ok) {

const data = await response.json();

setUser(data.msg);

}

} catch (err) {

console.error("Error: ", err.message);

}

};

Here, after the response is obtained from the backend, it is stored inside the user state variable.

* Later, it is sent to other components across the components using **AuthContext.Provider.**

<AuthContext.Provider

value={{ isLoggedIn, storeTokenInLS, LogoutUser, user }}

>

{children}

</AuthContext.Provider>

* But before that, we need to render the **userAuthentication** method once the component is called.

useEffect(() => {

userAuthentication();

}, []);

Our work is completed inside auth.jsx**.**

* Now, let us work inside the **Contact.jsx** component.

const [userData, setuserData] = useState(true);

const {user} = useAuth();

if(userData && user){

setContact({

username : user.username,

email : user.email,

message : ""

});

setuserData(false);

}

You need to add this little piece of code. You can see that we have added an extra state variable, **userData.**

If we don't keep it, then, as the user always has data, it will render the setContact again and again, trying to keep the initial value uncontrollably.

So, userData is set to false at the end of the if statement block to prevent future rendering of the same piece of code to **setContact.**

### Conclusion

Congratulations! Now you have made an auto-fill contact form that fills in data by authenticating the user from the database.

# Day 31 : Get the Username from the database of the logged-in User on the About Page

Here, we are going to get the data of logged-in users from the database and show it on the About page. This can be useful for making a profile page.

### How do I get the username from the database of logged-in users on the About page?

* First, we need to get the data of the logged-in user through a token.This has been done already on Day 30 and Day 31.On Day 30, we dealt with the backend, while on Day 31, we dealt with the frontend using ContextAPI.
* Now, we need to use the data in the front end to show it on the About Page.It's not a big deal. Just import the useAuth component and destructure the object.

import { useAuth } from "../store/auth";

const { user } = useAuth();

If the user gets the data, show the name; otherwise, show some random message.

The whole part of the code is:

import React, { useState } from "react";

import { useAuth } from "../store/auth";

const About = () => {

const { user } = useAuth();

return (

<section>

{user.username ? <p>Hello, {user.username}</p> : <p>Loading...</p>}

</section>

);

};

export default About;

### Conclusion

Now, you have to design your About Page as required.

# Day 32 : Store Contact Form Data in MERN Stack

Now, we are going to store the contact form data in the database in MERN Stack.

### How can we store contact form data in the MERN Stack?

* Let us start to work inside the Contact.jsx.We need to store the data in database on submit so let us implement the logic inside the **handleSubmit()** method.

const handleSubmit = async(e) => {

e.preventDefault();

const response = await fetch("http://localhost:3000/api/form/contact",{

method : "POST",

headers : {

"Content-Type" : "application/json"

},

body : JSON.stringify(contact)

})

if(response.ok){

window.alert("Message sent successfully.");

setContact(defaultContactForm);

}

};

Here, we are just using fetch to send the data into the backend API for storing in the database.

Then we showed a alert message and setContact to defaultContactForm on storing the data.

Now, you might be wondering about defaultContactForm as we have never talked about it before.

const defaultContactForm = {

username: "",

email: "",

message: "",

}

const [contact, setContact] = useState(defaultContactForm);

We have used it as the same object with empty data was called several times. So, this trick will ease our code.

### Conclusion

Now, you have successfully finished storing the data in the backend.

# Day 33 : Create a Service Schema, Model, Route, and Controller in the MERN Stack

Today, we will create the backend for the service functionality.

### How do I create a service schema?

To get a service schema, we need to go inside the **models** folder, and we will do work inside a new file named **services-model.js**.

const { Schema, model, default: mongoose } = require("mongoose");

const ServiceSchema = new Schema({

service: {

type: String,

required: true,

},

description: {

type: String,

required: true,

},

price: {

type: String,

required: true,

},

provider: {

type: String,

required: true,

},

});

Here, our schema contains service, description, price, and provider. Now, we need the model to send data into the database.

### How do I create a service model?

Creating a service model is easy. Just below the Schema, you need to write the following code:

const Service = new model("Service", ServiceSchema);

module.exports = Service;

**The model** takes two parameters: the first is the database name, and the second is the schema name.

### How do I create a service route?

Let us create a route for service inside the **router** folder.

Here, we have used the express.Router() method to create the route. **Don't worry about the services.**

That is a controller, and we are going to create that after we finish creating a route.

const express = require('express');

const router = express.Router();

const services= require("../controllers/service-controller");

router.route("/service").get(services);

module.exports = router;

Now, we also need to include this on the **server.js**.

Include this wherever possible, or, let us say, insert this code on the part where routes are being set.

app.use("/api/data",serviceRoute);

### How do I create a service controller?

We need to write the following code:

const Service = require("../models/services-model");

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

try {

const response = await Service.find();

if (!response) {

res.status(404).json({ msg: "No Services found" });

return;

}

res.status(200).json(response);

} catch (err) {

next(err);

}

};

module.exports = services;

As we are getting data from the service model, we need to require the module in our controllers.

Also, we should write this code inside the **controllers** folder by creating a file named **service-controller.js**.

To get data from the database, we first need to have data in the database. You can add the following JSON data:

[

{

"service": "Web Development",

"description": "Building and maintaining websites",

"price": "500",

"provider": "Company XYZ"

},

{

"service": "Graphic Design",

"description": "Creating visual content",

"price": "300",

"provider": "Company ABC"

},

{

"service": "Content Writing",

"description": "Crafting written materials",

"price": "400",

"provider": "Content Co."

},

{

"service": "Digital Marketing",

"description": "Promoting brands online",

"price": "600",

"provider": "Digital Wizards"

},

{

"service": "SEO Optimization",

"description": "Improving website visibility",

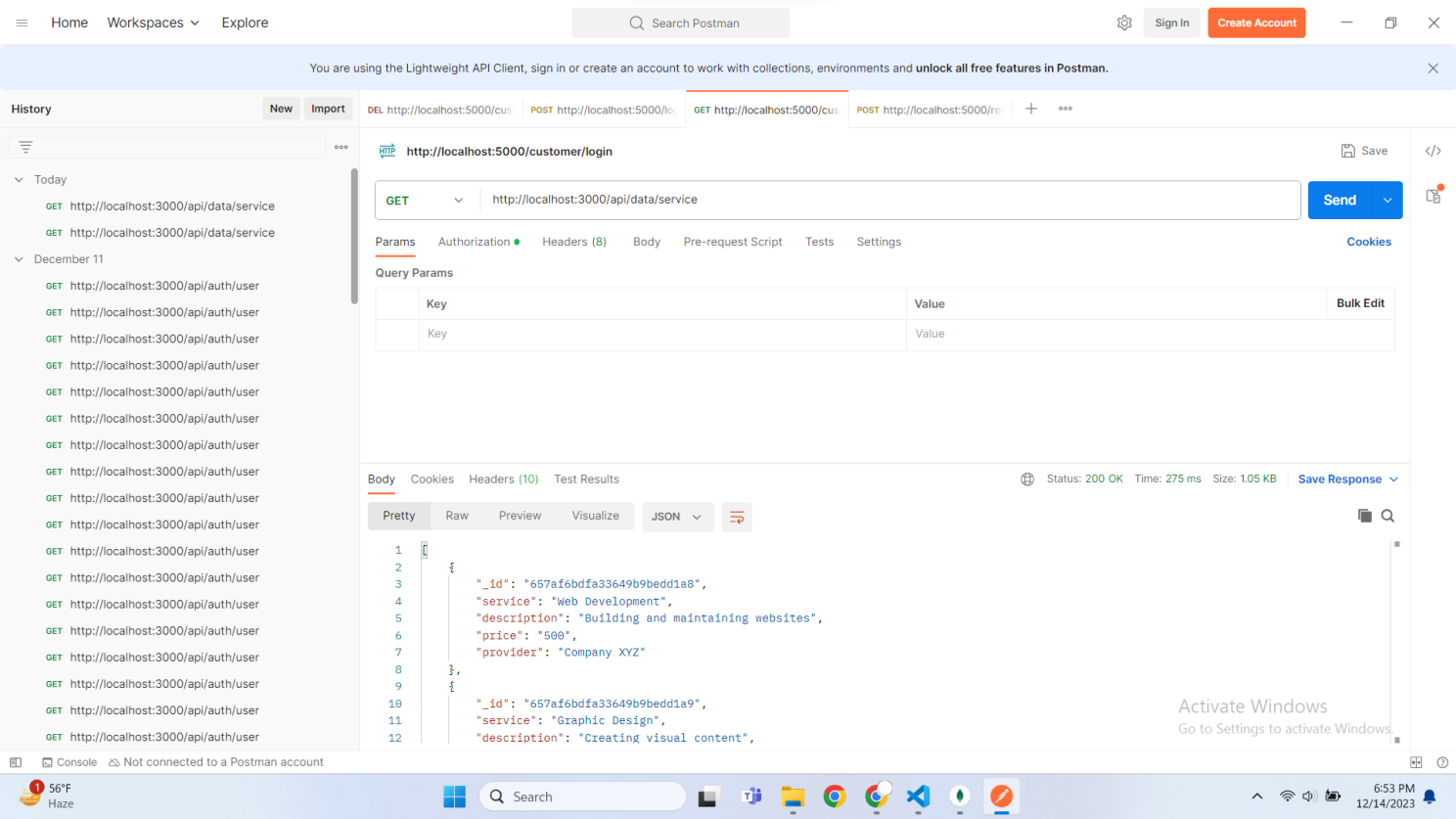
"price": "350",

"provider": "SearchMasters"

}

]

### How do I test the route in the POSTMAN tool?



Testing the route in the POSTMAN tool

Nothing; we need to use the GET method and enter the route of the logic.

You can see that we are getting the data from the database below.

### Conclusion

Now, your backend for the services is all ready.

# Day 34 : Create Dynamic Cards by Fetching Backend Services Data with React

On the previous day, we had already created the setup for the backend to fetch data from the database.

### How do I fetch backend service data with React?

For this, we are going to fetch data from the front end and send it to different parts of our front-end project using the context API.

Here, we are going to create a method that fetches data from the backend API.

// to fetch services data from the database

const getServices = async() =>{

try{

const response = await fetch(`http://localhost:3000/api/data/service`,{

method : "GET"

});

if(response.ok){

const data = await response.json();

setService(data);

}

}catch(err){

console.log(err);

}

}

Now, you might be wondering about setService(data). We have handled a useState in the above lines.

const [service, setService] = useState([]);

We have used an empty array for initializing because we will get an array of objects from the backend.

Now, we need to send a message to call the function. We are going to call the data inside useEffect Hook, which we have stated in earlier articles.

useEffect(() => {

userAuthentication();

getServices();

}, []);

Now, let us send the data to the different components in the front end.

return (

<AuthContext.Provider

value={{ isLoggedIn, storeTokenInLS, LogoutUser, user, service }}

>

{children}

</AuthContext.Provider>

);

After that, let us use the data in the **service** component.

import { useAuth } from "../store/auth";

Then, let us get the **service** that we have sent through the context API.

const { service } = useAuth();

Finally, let us create dynamic cards using JSX.

const Service = () => {

const { service } = useAuth();

return (

<>

<section>

<div>

<h1 className="text-5xl">Services</h1>

</div>

</section>

<div className="flex gap-3 ">

{service.map((item, index) => {

return (

<div className="flex flex-col gap-4 bg-blue-200">

<h2 className="text-3xl">{item.service}</h2>

<p>{item.price}</p>

<p>{item.provider}</p>

<p>{item.description}</p>

</div>

);

})}

</div>

</>

);

};

export default Service;

### Conclusion

Now, you have successfully made the service cards. I will leave styling for you.

# Day 35 : Create Dynamic Cards by Fetching Backend Services Data with React

On the previous day, we had already created the setup for the backend to fetch data from the database.

### How do I fetch backend service data with React?

For this, we are going to fetch data from the front end and send it to different parts of our front-end project using the context API.

Here, we are going to create a method that fetches data from the backend API.

// to fetch services data from the database

const getServices = async() =>{

try{

const response = await fetch(`http://localhost:3000/api/data/service`,{

method : "GET"

});

if(response.ok){

const data = await response.json();

setService(data);

}

}catch(err){

console.log(err);

}

}

Now, you might be wondering about setService(data). We have handled a useState in the above lines.

const [service, setService] = useState([]);

We have used an empty array for initializing because we will get an array of objects from the backend.

Now, we need to send a message to call the function. We are going to call the data inside useEffect Hook, which we have stated in earlier articles.

useEffect(() => {

userAuthentication();

getServices();

}, []);

Now, let us send the data to the different components in the front end.

return (

<AuthContext.Provider

value={{ isLoggedIn, storeTokenInLS, LogoutUser, user, service }}

>

{children}

</AuthContext.Provider>

);

After that, let us use the data in the **service** component.

import { useAuth } from "../store/auth";

Then, let us get the **service** that we have sent through the context API.

const { service } = useAuth();

Finally, let us create dynamic cards using JSX.

const Service = () => {

const { service } = useAuth();

return (

<>

<section>

<div>

<h1 className="text-5xl">Services</h1>

</div>

</section>

<div className="flex gap-3 ">

{service.map((item, index) => {

return (

<div className="flex flex-col gap-4 bg-blue-200">

<h2 className="text-3xl">{item.service}</h2>

<p>{item.price}</p>

<p>{item.provider}</p>

<p>{item.description}</p>

</div>

);

})}

</div>

</>

);

};

export default Service;

### Conclusion

Now, you have successfully made the service cards. I will leave styling for you.

# Day 36 : Integrating React Toastify for Dynamic Alerts in React App

Showing the errors and success messages along with warnings using the alert() method is boring.

### How to integrate React Toastify for dynamic alerts?

First, you need to install **react-toastify** on the front end.

npm install react-toastify

After that, you need to import **ToastContainer** in our main.jsx.

import { ToastContainer} from "react-toastify";

If you want to use it properly, you need to import CSS too.

import "react-toastify/dist/ReactToastify.css";

Now, let us use this in the **main.jsx** like this:

<ToastContainer

position="top-right"

autoClose={3000}

hideProgressBar={false}

newestOnTop={false}

closeOnClick

rtl={false}

pauseOnFocusLoss

draggable

pauseOnHover

theme="light"

/>

Now, wherever you want to display errors, warnings, and success messages. You can use methods like:

toast.info()

toast.success()

toast.warning()

toast.error()

toast()

### Conclusion

Now, you can see the error coming beautifully.

# Day 37 : Fix the Problem of Needing Refreshing for Login and Logout

As you can see, we had problems logging in to the system. The page wasn't refreshed.

How do you fix the problem of needing to refresh for login and logout?

There was a simple error that we couldn't refresh the page as we hadn't stored the token in **auth.jsx** and set it into the setToken() method.

const storeTokenInLS = (server\_token) => {

setToken(server\_token);

return localStorage.setItem("token", server\_token);

};

Now, you can see that your page reloads automatically.

### Conclusion

You have solved the problem of refreshing.

#### Continuing with our admin panel

# Day 38 : Retrieve the user data from the database

First, we need to create a controller to get all the users data. Let us say getAllUsers inside controllers/admin-controller.js

*const User = require("../models/user-model");*

*const getAllUsers = async(req,res) =>{*

*try{*

*// const users = await User.find({}).select("-password");*

*const users = await User.find({}, {password : 0});*

*if(!users && users.length === 0){*

*return res.status(500).json({msg : "No Users Found"})*

*}*

*return res.status(200).json({users});*

*}catch(err){*

*next(err);*

*}*

*}*

*module.exports = {getAllUsers};*

Then we need to create route for this inside the router/admin-router.js

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

*const router = express.Router();*

*const adminControllers = require('../controllers/admin-controller');*

*router.route('/users').get(adminControllers.getAllUsers);*

*module.exports = router;*

Now, you just need to create main route inside server.js

*const adminRoute = require('./router/admin-router');*

*app.use("/api/admin",adminRoute);*

### Conclusion

Now, you have successfully create a route to get all users data from the database.

# Day 39 : Retrieve the contact data from the database

Today, we will learn to get the contacts data succesfully from the database in the backend.

First, we need to create a controller to get all the contacts data. Let us say getAllContactss inside controllers/admin-controller.js

*const Contact = require("../models/contact-model");*

*const getAllContacts = async(req,res) =>{*

*try{*

*const contacts = await Contact.find();*

*if(!contacts && contacts.length === 0){*

*return res.status(500).json({msg : "No Contacts Found"});*

*}*

*return res.status(200).json({contacts})*

*}catch(err){*

*next(err);*

*}*

*}*

*module.exports = {getAllUsers,getAllContacts};*

Then we need to create route for this inside the router/admin-router.js

*router.route('/contacts').get(adminControllers.getAllContacts);*

### Conclusion

Now, you have successfully create a route to get all contacts data from the database.

# Day 40 : Secure our Admin Route with JWT Verification and Middleware

Nothing much we hav already created middlware for verifying JWT Token. We just need to pass that middleware to our routes of admin in admin-router.js

*const authMiddleware = require("../middlewares/auth-middleware");*

*router.route('/users').get(authMiddleware,adminControllers.getAllUsers);*

*router.route('/contacts').get(authMiddleware,adminControllers.getAllContacts);*

### Conclusion

Now, we are all set with the admin route authentication to get access to certain routes.

# Day 41 : Verifying whether user is admin or not

While verifying the user, we have stored all the user data in req.user.

Now, we just need to create a admin middleware to check the boolean value in isAdmin.

For that, let us create a admin-middleware.js

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

*try{*

*console.log(req.user);*

*if(!req.user.isAdmin){*

*throw new Error("You are not admin");*

*}*

*next();*

*}catch(err){*

*next(err);*

*}*

*}*

*module.exports = adminMiddleware;*

Then we need to import in admin-router.js and add there to check whether the user is admin or not.

const adminMiddleware = require("../middlewares/admin-middleware");

router.route('/users').get(authMiddleware,adminMiddleware,adminControllers.getAllUsers);

### Conclusion

Now, the data will be shown only if the user is Admin otherwise not.

# Day 42 : Building an Admin Page Layout - Nested Admin Routes

For creating admin pages layouts with nested admin routes. First let us create nested admin routes inside the App.jsx

*import Admin\_Layout from "./components/layouts/Admin\_Layout";*

*import Admin\_Contacts from "./pages/admin/Admin\_Contacts";*

*import Admin\_Users from "./pages/admin/Admin\_Users";*

Inside the Routes

*{/\* Create a nested loop \*/}*

*<Route path="/admin" element={<Admin\_Layout />}>*

*<Route path="users" element={<Admin\_Users />} />*

*<Route path="contacts" element={<Admin\_Contacts />} />*

*</Route>*

*import React from "react";*

*import { NavLink } from "react-router-dom";*

*const Admin\_Layout = () => {*

*return (*

*<>*

*<header>*

*<nav>*

*<ul>*

*<li>*

*<NavLink to="/admin/users"> users </NavLink>*

*</li>*

*<li>*

*<NavLink to="/admin/contacts"> contacts </NavLink>*

*</li>*

*<li>services</li>*

*<li>home</li>*

*</ul>*

*</nav>*

*</header>*

*</>*

*);*

*};*

*export default Admin\_Layout;*

But, when you do this you may observe that the Admin\_Users and Admin\_Contacts Pages doesn’t load. For that, we need to use Outlet in the parent component which is one of the feature of react-rotuer-dom.

*import { NavLink, Outlet } from "react-router-dom";*

*import React from "react";*

*import { NavLink, Outlet } from "react-router-dom";*

*const Admin\_Layout = () => {*

*return (*

*<>*

*<Outlet />*

*</>*

*);*

*};*

*export default Admin\_Layout;*

### Conclusion

Now, you are all set with nested routes in the frontend.

# Day 43 : Get all the users data and show in table

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

*import {useAuth} from "../../store/auth"*

*import { toast } from 'react-toastify';*

*const Admin\_Users = () => {*

*const [users, setUsers] = useState([]);*

*const {authorization\_token} = useAuth();*

*const getAllUsers = async() =>{*

*try{*

*const res = await fetch("http://localhost:3000/api/admin/users",{*

*method : "GET",*

*headers : {*

*"Authorization" : authorization\_token*

*}*

*})*

*if(!res.ok){*

*throw new Error("Cannot fetch data");*

*}*

*const data = await res.json();*

*console.log(data.users);*

*setUsers(data.users);*

*}catch(err){*

*toast.error(err.message);*

*}*

*}*

*useEffect(()=>{*

*getAllUsers();*

*},[])*

*return (*

*<>*

*<h1>Admin User Data</h1>*

*<table border="1">*

*<thead>*

*<th>Username</th>*

*<th>Email</th>*

*<th>Phone</th>*

*<th>Functionalities</th>*

*</thead>*

*{*

*users.map((curUser, index)=>{*

*return <tr key={index}>*

*<td>{curUser.username}</td>*

*<td>{curUser.email}</td>*

*<td>{curUser.phone}</td>*

*<td>Edit</td>*

*<td>Delete</td>*

*</tr>*

*})*

*}*

*</table>*

*</>*

*)*

*}*

*export default Admin\_Users*

*const authorization\_token = `Bearer ${token}`;*

*const userAuthentication = async () => {*

*try {*

*const response = await fetch("http://localhost:3000/api/auth/user", {*

*method: "GET",*

*headers: {*

*Authorization: authorization\_token,*

*},*

*});*

*if (response.ok) {*

*const data = await response.json();*

*setUser(data.msg);*

*}*

*} catch (err) {*

*console.error("Error: ", err.message);*

*}*

*};*

*return (*

*<AuthContext.Provider*

*value={{ isLoggedIn, storeTokenInLS, LogoutUser, user, service, authorization\_token }}*

*>*

*{children}*

*</AuthContext.Provider>*

*);*

# Day 44 : Delete the users details on clicking Delete Button

First create a onClick event listener on delete button.

*<td*

*className="bg-red-700 text-white rounded-lg"*

*onClick={() => {*

*deleteUser(curUser.\_id);*

*}}*

*>*

*Delete*

*</td>*

Now, implement the work inside the deleteUser Button.

*const deleteUser = async (id) => {*

*try {*

*const res = await fetch(*

*`http://localhost:3000/api/admin/users/delete/${id}`,*

*{*

*method: "DELETE",*

*headers: {*

*Authorization: authorization\_token,*

*},*

*}*

*);*

*// const data = await res.json();*

*if(res.ok){*

*getAllUsers();*

*}*

*} catch (err) {*

*toast.error(err.message);*

*}*

*};*

In the backend, you need to implement the real logic now.

In admin-router.js

*router.route('/users/delete/:id').delete(authMiddleware, adminMiddleware, adminControllers.deleteUserById);*

Then in admin-controller.js

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

*try{*

*const id = req.params.id;*

*await User.deleteOne({\_id : id});*

*return res.status(200).json({msg : "User Deleted Successfully"});*

*}catch(err){*

*next(err);*

*}*

*}*

*module.exports = {getAllUsers,getAllContacts,deleteUserById, getUserById, updateUserById};*

# Day 45 : Create backend by get user by id

Before updating, we need to get user by id. Let’s setup the backend. In the admin-router.js

*router.route('/user/:id').get(authMiddleware, adminMiddleware, adminControllers.getUserById);*

Then in admin-controller.js

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

*try{*

*const id = req.params.id;*

*const user = await User.findById(id).select('-password');*

*if(!user){*

*return res.status(404).json({msg : 'User not found'});*

*}*

*console.log(req.params.id);*

*return res.status(200).json(user);*

*}catch(err){*

*next(err);*

*}*

*}*

Now, we are all set. We need to work in the frontend.

# Day 46 : Create update page in frontend

First let us create a route form the update page.

*import Admin\_Update from "./pages/admin/Admin\_Update";*

*{/\* Create a nested loop \*/}*

*<Route path="/admin" element={<Admin\_Layout />}>*

*<Route path="users" element={<Admin\_Users />} />*

*<Route path="contacts" element={<Admin\_Contacts />} />*

*<Route path="users/:id/edit" element={<Admin\_Update />} />*

*</Route>*

Now inside pages>admin, Create Admin\_Update.jsx with the following JSX.

*import React, { useEffect, useState } from "react";*

*const Admin\_Update = () => {*

*return (*

*<>*

*<section className="flex p-4 gap-10">*

*<form className="flex flex-col gap-3 px-10">*

*<h1 className="text-4xl">Update User Data</h1>*

*<div className="flex flex-col gap-2">*

*<label htmlFor="username">Username</label>*

*<input*

*type="text"*

*name="username"*

*value={profile.username}*

*onChange={handleChange}*

*className="border-2 border-b-blue-400"*

*required*

*/>*

*</div>*

*<div className="flex flex-col gap-2">*

*<label htmlFor="email">Email</label>*

*<input*

*type="email"*

*name="email"*

*value={profile.email}*

*onChange={handleChange}*

*className="border-2 border-b-blue-400"*

*required*

*/>*

*</div>*

*<div className="flex flex-col gap-2">*

*<label htmlFor="phone">Mobile</label>*

*<input*

*type="phone"*

*name="phone"*

*value={profile.phone}*

*onChange={handleChange}*

*className="border-2 border-b-blue-400"*

*required*

*/>*

*</div>*

*<button*

*type="submit"*

*className="bg-blue-400 text-white px-4 py-2 rounded-lg"*

*>*

*Update*

*</button>*

*</form>*

*</section>*

*</>*

*);*

*};*

*export default Admin\_Update;*

You might notice about lots of errors. let us clear that by creating a state variable profile and a method called handleChange.

*const [profile, setProfile] = useState({*

*username: "",*

*email: "",*

*phone: "",*

*});*

*const handleChange = (e) => {*

*setProfile({ ...profile, [e.target.name]: e.target.value });*

*};*

Then, we need to get the user data from the URL. For that we have useParams in react-router-dom

*const { authorization\_token } = useAuth();*

*const params = useParams();*

*//get the user on edit button*

*const getSingleUserData = async (id) => {*

*try {*

*console.log(params);*

*const res = await fetch(*

*`http://localhost:3000/api/admin/user/${params.id}`,*

*{*

*method: "GET",*

*headers: {*

*Authorization: authorization\_token,*

*},*

*}*

*);*

*const data = await res.json();*

*console.log("users after edit : ", data);*

*setProfile(data);*

*} catch (err) {*

*toast.error("User Update : ", err.message);*

*}*

*};*

*useEffect(()=>{*

*getSingleUserData();*

*},[])*

Don’t forget to call the method using useEffect Hook as I did.

You might see we have used toast.error() method to show error. Don’t forget to impor them.

*import { toast } from "react-toastify";*

*import { useAuth } from "../../store/auth";*

*import { useParams } from "react-router-dom";*

# Day 47 : Update user details in the backend and database with frontend

Nothing much first let us start with creating routes.

*router.route('/users/edit/:id').put(authMiddleware, adminMiddleware, adminControllers.updateUserById);*

Then, let us start with the method:

*const updateUserById = async (req, res) => {*

*try {*

*const id = req.params.id;*

*const updatedUserData = req.body;*

*const userExist = await User.findOne({email : updatedUserData.email});*

*if(userExist){*

*return res.status(400).json({msg: "Email already exists."});*

*}*

*const updatedUser = await User.updateOne(*

*{ \_id: id },*

*{*

*$set: updatedUserData,*

*}*

*);*

*return res.status(200).json(updatedUser);*

*} catch (err) {*

*next(err);*

*}*

*};*

At last, don’t forget to export it.

*module.exports = {*

*getAllUsers,*

*getAllContacts,*

*deleteUserById,*

*getUserById,*

*updateUserById,*

*};*

# Day 48 : Update user in the frontend

*// on submitting the form*

*const handleSubmit = async(e) =>{*

*e.preventDefault();*

*try{*

*const res = await fetch(`http://localhost:3000/api/admin/users/edit/${params.id}`,{*

*method : "PUT",*

*headers : {*

*"Content-Type" : "application/json",*

*Authorization : authorization\_token*

*},*

*body : JSON.stringify(profile)*

*})*

*if(!res.ok){*

*throw new Error("Cannot update error.Go back and try again");*

*}else{*

*const data = await res.json();*

*console.log(res)*

*console.log(data)*

*setProfile({*

*username : data.username,*

*email : data.email,*

*phone : data.phone*

*});*

*toast.success("User updated successfully");*

*}*

*}catch(err){*

*toast.error("Update error: ", err.message)*

*}*

*}*

# Day 49 : Get contact form every data (frontend+backend)

# Day 50 : Delete contact form data from frontend and backend