>>> mkdir server //All the backend code will be put into the server directory

>>> cd server >>>npm init -y

>>> npm i express

>>> cd ../

>>> cat > server.js [inside server folder]

SERVER.js

const express = require("express");

const app = express();

// route the person visiting which page

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

    // 200 is for success

    res.status(200).send("Hello Omar to the Mern Stack Journey");

})

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

    // 200 is for success

    res.status(200).send("Hello Omar Welcome to Registeration page");

})

// Run the server on the specific ports

const PORT = 5000;

app.listen(PORT,()=>{

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

})

Run the application:

>>> npm run start

>>> npm install -g nodemon [Automatic Restart. NO need to start server again and again]  
>>> nodemon server.js [If does not work go to powershell and >>>SetExecutionPolicy Unrestricted]

-> Now on run from nodemon

Router:

Every route will be kept in separate file such as register, about and we will pass reference to the server file.

Inside Server file create router folder

>>> cd server

>>> mkdir router

Inside router file create auth-router.js

>>> cd router; cat > auth-router.js

In the router file you write all the routes and export it and other file which wants to use the route import your file with the help `use`

- auth-router.js

const express = require("express");

const router = express.Router();

// route the person visiting which page

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

    // 200 is for success

    res.status(200).send("Hello Omar to the Mern Stack Journey. I am from router");

})

// to use routes which make or includes this file

module.exports = router;

- server.js

const express = require("express");

const app = express();

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

// The nodemon module is constantly using server.js to make it use other file include `use`

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

* app.use('/api/auth', router);

The line app.use('/api/auth', router); in an Express.js application sets up middleware to handle requests to a specific route. Here’s a breakdown of what it does:

app.use: This function is used to mount middleware functions at a specified path. Middleware functions are executed when the path matches the request's path.

'/api/auth': This is the base path for which the middleware will be executed. Any request that starts with /api/auth will be passed to the middleware.

router: This is an instance of an Express router. A router is a mini Express application that can handle a subset of the app’s routes.

When you combine these together, app.use('/api/auth', router); means that any request starting with /api/auth will be handled by the middleware functions defined in the router.

Example:

- A request to http://localhost:3000/api/auth/login will be handled by the router.post('/login', ...) route handler.

- A request to http://localhost:3000/api/auth/register will be handled by the router.post('/register', ...) route handler.

- homepage

url: localhost:5000/api/auth/

auth-router.js:

const express = require("express");

const router = express.Router();

// route the person visiting which page

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

//     // 200 is for success

//     res.status(200).send("Hello Omar to the Mern Stack Journey. I am from router");

// });

// You can use this as well

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

    res

    .status(200)

    .send("Hello Omar I am from router as well!!")

});

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

    res

    .status(200)

    .send("Hello Omar I am from router register as well!!")

});

// to use routes which make or includes this file

module.exports = router;

Controllers:

>>> cd server

>>> mkdir controllers; cat > auth-controller.js

Dispatch and Action:

Dispatch: Function that carries out that action.

Action: Description of what you want to do.

Just for layman:

Dispatch:

router.route('/')

Action:

.get((req,res)=>{

    res

    .status(200)

    .send("Hello Omar I am from router as well!!")

});

In MERN or Express it is known as CONTROLLERS

auth-controller.js

// Home logic start

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

    try{

        res

        .status(200)

        .send("Hello Omar I am from router as well!! Home page")

    }catch(error){

        console.log(error)

    }

}

// Home logic end

// Register Logic Start

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

    try{

        res

        .status(200)

        .send("Hello Omar I am from router as well!! Register Page")

    }catch(error){

        console.log(error)

    }

}

// Register Logic End

module.exports = {home, register}

auth-router.js:

const express = require("express");

const router = express.Router();

// const {home,register} = require('../controllers/auth-controller')\

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

// route the person visiting which page

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

//     // 200 is for success

//     res.status(200).send("Hello Omar to the Mern Stack Journey. I am from router");

// });

// You can use this as well

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

//     res

//     .status(200)

//     .send("Hello Omar I am from router as well!!")

// });

// Since you added home in your contorller and 'require' here now just pass home

// router.route('/').get(home);

router.route('/').get(authControllers.home)

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

// to use routes which make or includes this file

module.exports = router;

Install Postman or use extension in vs code:

After installing and logging in:

1. Create a Collection [name='mernAdmin']
2. Add a request [name='register']
3. Paste your url ‘http://localhost:5000/api/auth/register’

// get => read

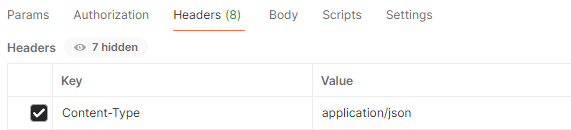
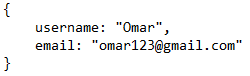
// post => send to database

// put or patch => update or insert new row

// delete => delete

Post:

router.route('/register').post(authControllers.register);

1. Make post request in postman
2. Write your post route {url}
3. Go to headers
4. Set key = Content-Type
5. Value = application/json
6. 
7. Go to body
8. Select raw
9. 
10. And hit send

Include json in app [Server.js]

>>> app.use(express.json())

Parses JSON: The express.json() middleware is used to parse the body of incoming HTTP requests that have a Content-Type of application/json.

Adds Parsed Data to req.body: Once the JSON is parsed, it is stored in req.body, making it easily accessible in your route handlers.

1. After including again hit the send in postman api
2. 

MongoDB

Log into MongoDb:

<https://www.mongodb.com/>

In the application install >>> npm i mongoose

In the server folder create a utils filder and inside create a db.js

>>> mkdir utils; cat > db.js

**Why Use Mongoose?**

1. **Schema Definition**: Mongoose allows you to define schemas for your data models, ensuring consistency and validation.
2. **Data Modeling**: It provides a powerful API to create, read, update, and delete records in a MongoDB database.
3. **Validation**: Mongoose supports built-in and custom validation, ensuring data integrity.
4. **Middleware**: You can define middleware (hooks) that are executed before or after certain operations, such as saving or removing documents.
5. **Queries**: Mongoose provides a rich query API to perform complex queries on your data.

In the homepage of MongoDb:

- Create a new cluster with M0.

- Set your password in deployment for creating a user.

- Click on compass and install it in the next step and setup compass.

- Open compass and compy the uri

In db.js:

const mongoose = require('mongoose')

mongoose.connect('mongodb://localhost:27017/project1DB')

mongoose.connection.on('connected',()=>{

    console.log("Connected to mongodb")

})

mongoose.connection.on('error',(err)=>{

    console.log("Error: ", err)

})

module.exports = mongoose;

and include in the server.js

>>> const db = require('./utils/db.js');

To setup MongoDB for the first time.

<https://www.youtube.com/watch?v=dEm2AS5amyA&ab_channel=EducationAnalysist>

You can also install mongodb shell to write your commands on shell.

<https://www.mongodb.com/try/download/shell>

To set up shell:

<https://www.youtube.com/watch?v=RzhCWeOJvyA&ab_channel=GameTrick>

db.js:

const mongoose = require('mongoose')

// mongoose.connect('mongodb://localhost:27017/project1DB')

const URI = 'mongodb://localhost:27017/project1DB'

// mongoose.connection.on('connected',()=>{

//     console.log("Connected to mongodb")

// })

// mongoose.connection.on('error',(err)=>{

//     console.log("Error: ", err)

// })

const connectDb = async () => {

    try {

        await mongoose.connect(URI)

        console.log("Connection Successfully")

    } catch (error) {

        console.log("Error: ", error)

    }

}

module.exports = connectDb;

server.js:

const connectDb = require('./utils/db.js');

// Run the server on the specific ports

const PORT = 5000;

connectDb().then(()=>{

    app.listen(PORT,()=>{

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

    });

})

Dotenv (Secure your private data with dotenv):

>>> npm i dotenv

- dotenv is a zero-dependency module that loads environment variables from a .env file into process.env in Node.js projects. This module is commonly used to manage configuration settings and secrets, such as database connection strings, API keys, and other sensitive data, without hardcoding them into your application's source code.

-> Connect MongoDB with backend:

Step1: Create a cluster

Step2: Click on connect cluster.

Step3: Click on Drivers.

Step4: Copy the connection string and add your password and Database name:

mongodb+srv://email:<password>@omarcluster.im5ijmw.mongodb.net/db\_name?retryWrites=true&w=majority&appName=clusterName

Step5: add into your db.js URI.

To make it Secure create a ‘.env’ file and add this code:

MONGODB\_URI = "mongodb+srv://forchegg80:omar123@omarcluster.im5ijmw.mongodb.net/project1DB?retryWrites=true&w=majority&appName=omarcluster"

And in your db.js in URI variable:

const URI = process.env.MONGODB\_URI

And in server.js import your .env file as:

require("dotenv"). config();

Creating model schema=>

Schema: Defines the structure of the documents within the collection. It specifies the fields, their types, and any additional constraints or validation.

>>> Create models folder

>>> Create “user-model.js”

>>> Define user schema.

const mongoose = require("mongoose");

// Define the User schema

const userSchema = new mongoose.Schema({

  username: {

    type: String,

    required: true,

  },

  email: {

    type: String,

    required: true,

  },

  phone: {

    type: String,

    required: true,

  },

  password: {

    type: String,

    required: true,

  },

  isAdmin: {

    type: Boolean,

    default: false,

  },

});

// define the model or the collection name

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

module.exports = User

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

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

    try {

        console.log(req.body);

        const { username, email, phone, password } = req.body; // Fixed 'passwword' typo

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

        if (userExist) {

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

        }

        // Create a new user instance

        const newUser = new User({ username, email, phone, password });

        // Save the new user to the database

        await newUser.save();

        // Send a success response

        res.status(200).json({ message: "User registered successfully" });

    } catch (error) {

        console.error(error);

        res.status(500).json({ msg: "Server error" });

    }

};

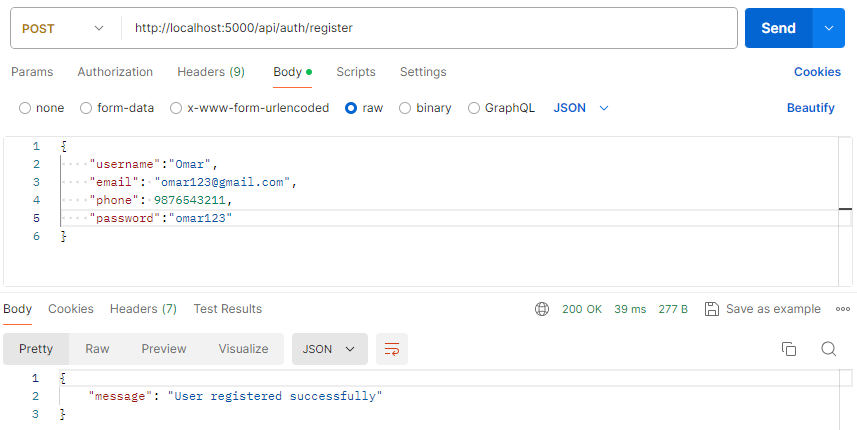
Go to clusters in Database tab,

Click on collection you will see your collection name

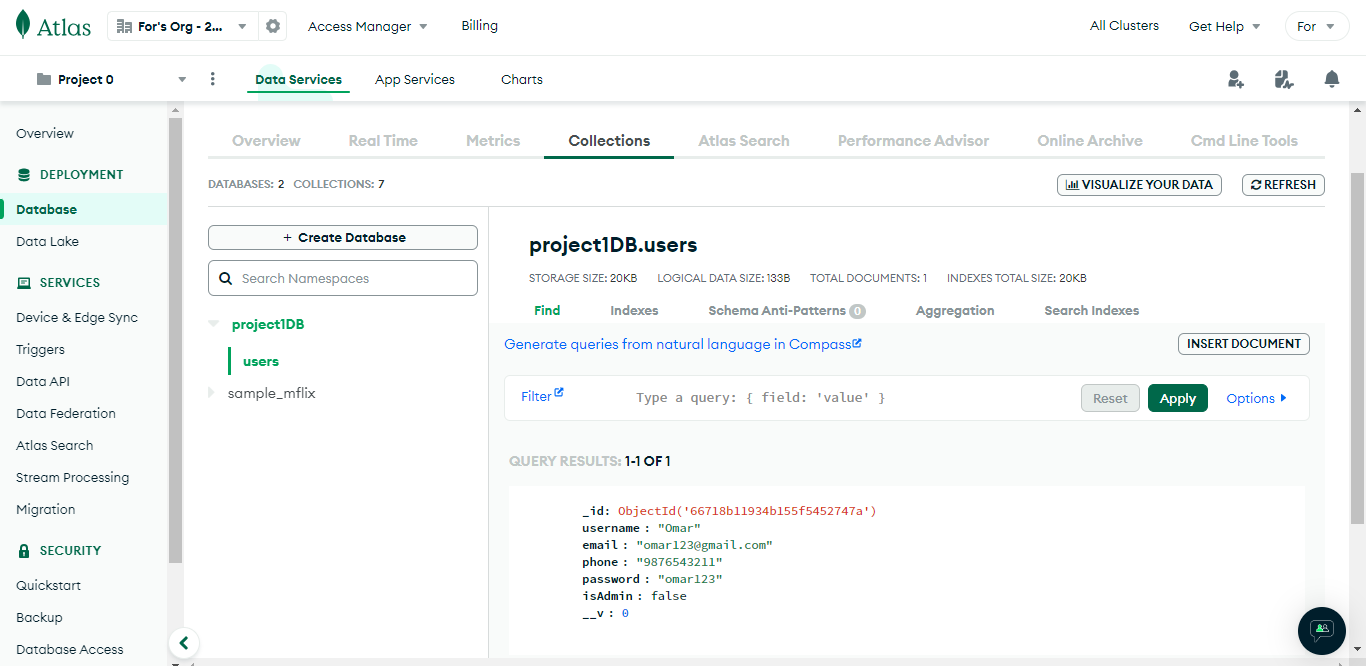
Go to postman and check for your post request.

Write a code for registration in your auth-controllers.js

Check in postman:



>>> Go to Database in the MongoDB web -> Click on your cluster -> click on Collections -> click on your database name and click on user schema and hit apply.



Bcrypt

>>> npm i bcrypt

Hash and compare are the two function that is used while using bcrypt.

const hash = bcrypt.hashSync(myPlaintextPassword, saltRounds);

bcrypt.compareSync(myPlaintextPassword, hash); // true

More salt more level of hashing.

In user\_model.js:

const bcrypt = require("bcrypt");

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

  const user = this;

  console.log("actual data ", this);

  if (!user.isModified) {

    return next();

  }

  try {

    const saltRound = await bcrypt.genSalt(10);

    const hashedPassword = await bcrypt.hash(user.password, saltRound);

    user.password = hashedPassword;

  } catch (error) {

    return next(error);

  }

});

Explanation:

The userSchema.pre("save", async function () { ... }) code defines a middleware function in Mongoose that runs automatically before a user document is saved to the database. This function is typically used to perform some processing on the user document, such as hashing the user's password for security purposes.

if (!user.isModified("password")) { return next(); } checks if the password field has been modified. If it hasn't, the function calls next() to proceed with saving the document without re-hashing the password.

next() is a callback function that passes control to the next middleware or to the save operation itself.

Salt

Salt is a random string added to a password before hashing it. The purpose of a salt is to ensure that even if two users have the same password, their hashed passwords will be different. This adds an extra layer of security and helps to protect against certain types of attacks, such as rainbow table attacks (precomputed hash attacks).

bcrypt.genSalt(10) generates a salt with a complexity of 10 rounds. The number of rounds determines how complex and time-consuming it is to generate the salt, making it harder for attackers to precompute hash values.

Hashing is a process that transforms the original password into a fixed-length string of characters, which is typically a digest that appears random. The hash function is designed to be a one-way function, meaning it's computationally infeasible to reverse the process and retrieve the original password from the hash.

bcrypt.hash(user.password, saltRound) hashes the user's password using the generated salt. The resulting hashed password is a unique string that corresponds to the input password combined with the salt.

This hashed password is what gets stored in the database instead of the plain text password.

Putting It Together

Combining salt and hash provides robust security for storing passwords:

Salting and Hashing:

First, a unique salt is generated for each password.

The password is then combined with this salt and hashed.

The resulting hashed password is stored in the database.

Secure Storage:

When the user logs in, their input password is combined with the same salt and hashed again.

If this new hash matches the stored hash, the password is correct.

In the controller.js

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

    try {

        console.log(req.body);

        const { username, email, phone, password } = req.body; // Destructure the request body

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

        if (userExist) {

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

        }

        // Create the new user instance and save it to the database

        const userCreated = await User.create({

            username,

            email,

            phone,

            password

        });

        // Send a success response

        res.status(200).json({ message: "User registered successfully", user: userCreated });

    } catch (error) {

        console.error(error);

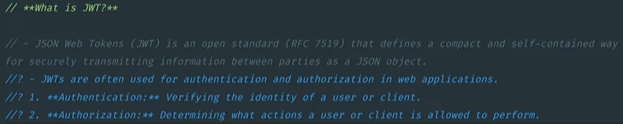
        res.status(500).json({ msg: "Server error" });

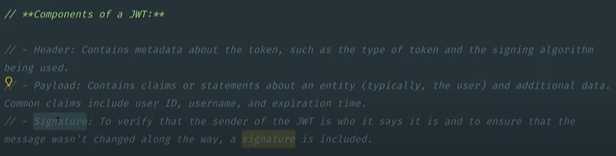
    }

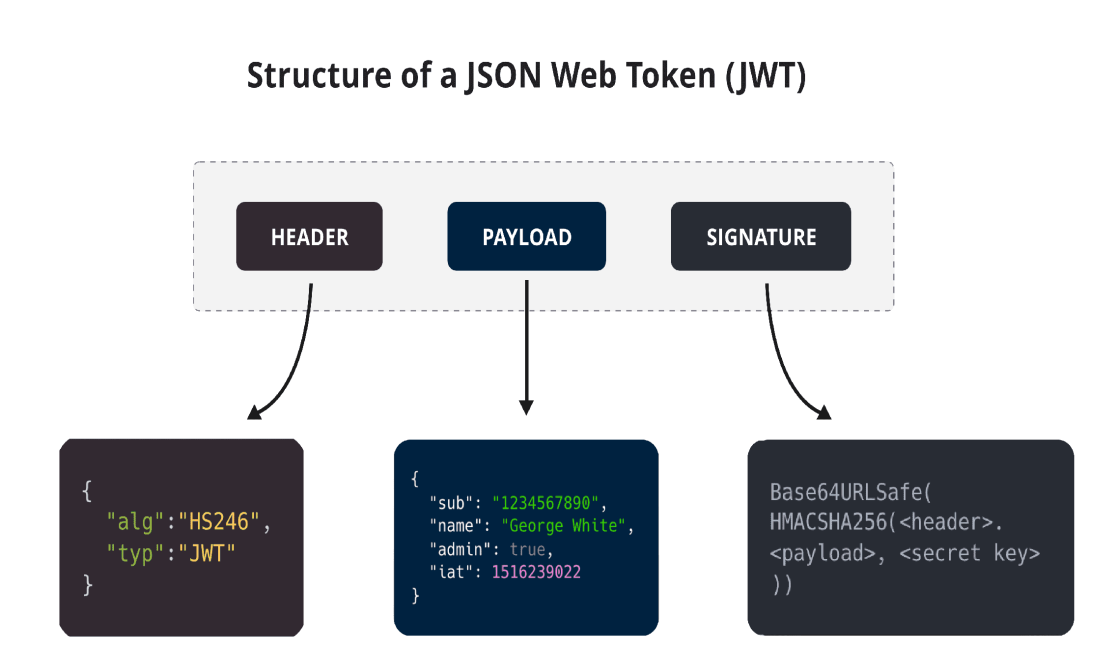
};

JWT: JSON Web Token

JWT (JSON Web Token) is a npm package that helps us in the authentication and authorization of the user.







JSON Web Token:

>>> npm i jsonwebtoken

User-model.js:

// json web token

userSchema.methods.generateToken = function() {

  const token = jwt.sign(

      {

          userId: this.\_id.toString(),

          email: this.email,

          isAdmin: this.isAdmin

      },

      process.env.JWT\_SECRET\_KEY,

      { expiresIn: "30d" }

  );

  return token;

};

Explanation:

jwt.sign(): This function from the jsonwebtoken library creates a signed JWT.

Payload: The first argument to jwt.sign() is the payload, which contains the data that will be encoded into the token.

userId: The unique identifier of the user (this.\_id.toString()). this refers to the current user document instance.

email: The user's email address (this.email).

isAdmin: A boolean indicating if the user has administrative privileges (this.isAdmin).

Secret Key: The second argument is the secret key used to sign the token. It's typically stored in an environment variable (process.env.JWT\_SECRET\_KEY) to keep it secure and configurable.

Options: The third argument is an options object. Here, expiresIn: "30d" sets the token to expire in 30 days.

Benefits of Using JWT

Stateless Authentication: JWTs allow for stateless authentication, meaning that the server does not need to store session information between requests.

Security: JWTs are signed, which means their integrity and authenticity can be verified by the server.

Scalability: Stateless authentication mechanisms like JWTs make it easier to scale applications since you don’t need to maintain session state across servers.

Conclusion

The generateToken method creates a JWT for a user, which can be used for authentication purposes in your application. It securely encodes user information and signs it with a secret key, making it a reliable way to implement stateless authentication.

User-model.js

const jwt = require("jsonwebtoken")

.env file:

JWT\_SECRET\_KEY = "CLOCKTICKLES"

auth-controller.js :

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

    try {

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

        // Check if all required fields are present

        if (!username || !email || !phone || !password) {

            return res.status(400).json({ msg: "All fields are required" });

        }

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

        if (userExist) {

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

        }

        // Create and save the new user

        const userCreated = await User.create({

            username,

            email,

            phone,

            password

        });

        // Generate a JWT

        const token = await userCreated.generateToken();

        // Send a success response

        res.status(200).json({

            message: "User registered successfully",

            user: userCreated,

            token: token,

            userId: userCreated.\_id.toString()

        });

    } catch (error) {

        console.error(error);

        res.status(500).json({ msg: "Server error" });

    }

};

After successfully creating the user, the code generates a JWT for the new user by calling the generateToken method on the userCreated instance.

The response includes the newly created user information, the generated JWT, and the user ID.

Benefits of Using JWT in Registration

Stateless Authentication

Explanation: After a user registers, they receive a JWT, which they can use for subsequent requests to authenticate themselves without the server needing to store session information.

Example: When the user logs in or accesses protected routes, they send the JWT in the Authorization header. The server verifies the token to authenticate the user.

Security

Explanation: The JWT is signed using a secret key (typically stored in an environment variable), ensuring that the token cannot be tampered with. The server can verify the token's integrity and authenticity.

Example: Even if a user intercepts the token, they cannot modify the payload (such as changing the user ID or privileges) without invalidating the signature.

Why Use JWT (JSON Web Token)

JWT (JSON Web Token) is used for securely transmitting information between parties as a JSON object. It is widely used for authentication and information exchange due to its compact size, self-contained nature, and security features. Here’s a detailed explanation of why JWT is used, along with an example to illustrate its usage.

Structure of JWT

A JWT consists of three parts:

Header: Contains metadata about the token, such as the type of token (JWT) and the signing algorithm being used.

Payload: Contains the claims. This is the data you want to transmit securely. Claims are statements about an entity (typically, the user) and additional data.

Signature: This is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

* Login Logic

auth-controller.js:

// Login Start

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

    try {

        const { email, password } = req.body;

        // Check if the user exists

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

        if (!userExist) {

            return res.status(400).json({ msg: "Invalid credentials" });

        }

        // Compare the provided password with the stored hashed password

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

        if (!isPasswordCorrect) {

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

        }

        // Generate a token (assuming you have a method on the user schema for this)

        const token = await userExist.generateToken();

        // Respond with the token and user ID

        res.status(200).json({

            msg: "Login successful",

            token,

            userId: userExist.\_id.toString()

        });

    } catch (error) {

        console.error(error);

        res.status(500).json({ msg: "Server error" });

    }

};

// Login End

module.exports = { home, register,login}

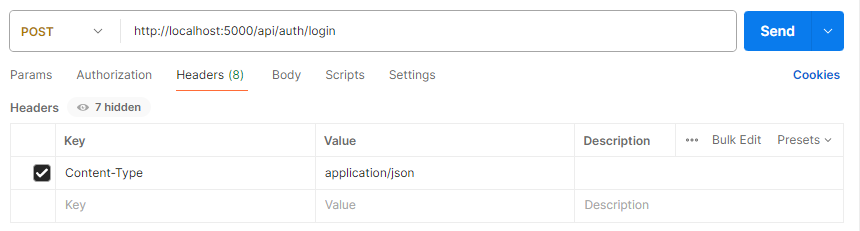
auth-router.js:

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

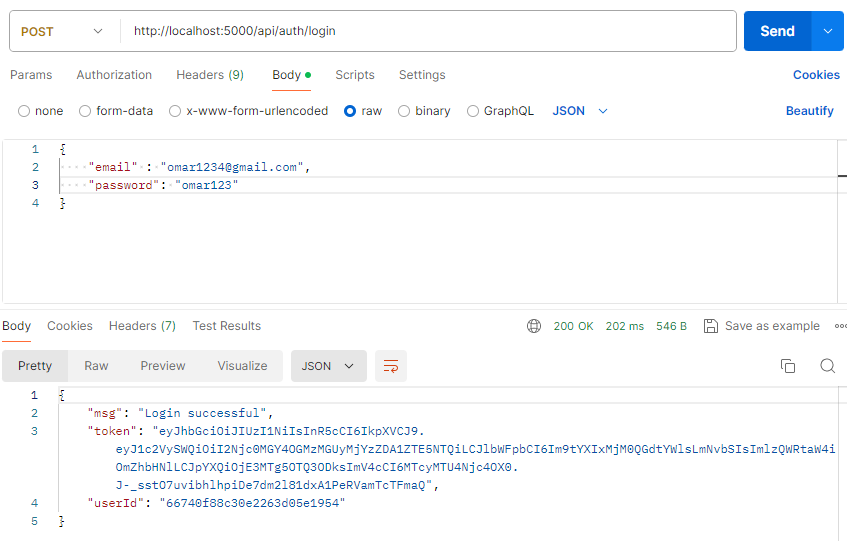
Check your url/ post request with postman:

>>> Set the url with the post request and headers.

>>> add your code/object to check if it works.



Add your email and password object:



Move user compare password to different location:

user-models.js:

// compare password:

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

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

}

auth-controllers.js:

const isPasswordCorrect = await userExist.comparePassword(password);

By moving the password comparison logic to a method on the user schema, you encapsulate the functionality related to the user model. This makes your code more modular, reusable, and easier to maintain. It follows the object-oriented principle of placing methods related to a specific object (in this case, a user) within the class (or schema) representing that object.

userSchema.methods.comparePassword: This method is added to the user schema. It takes a password as an argument and compares it with the hashed password stored in the database using bcrypt.compare.

this.password: Refers to the user's hashed password stored in the database.

By defining comparePassword as a method on the schema, you can now call it on any user instance.

* Encapsulation:

By encapsulating the password comparison logic within the user model, you ensure that this logic is consistently used whenever needed. This prevents code duplication and reduces the likelihood of errors.

* Reusability:

If you need to compare passwords in other parts of your application, you can simply call user.comparePassword(password) without rewriting the comparison logic.

* Clean Code:

This approach keeps your controller code cleaner and more focused on handling HTTP requests and responses, while the user model handles user-specific logic.

FORM VALIDATION: [Zod]

**Zod** is a TypeScript-first schema declaration and validation library. It provides a simple, expressive syntax to define schemas and perform validation, making it a powerful tool for ensuring data integrity in your applications. In a MERN (MongoDB, Express, React, Node.js) stack, Zod can be used to validate incoming data, such as request bodies, query parameters, and even responses, both on the server-side (Node.js/Express) and client-side (React).

**Key Features of Zod**

1. **Type Inference:** Zod can infer TypeScript types from schemas.
2. **Flexible Validation:** Zod provides a wide range of validation methods and allows for custom validation logic.
3. **Composition:** Schemas can be composed and reused.
4. **Error Messages:** Zod provides detailed error messages out of the box.

>>> mkdir validator; cat > auth-validator.js [inside server folder]

>>> npm i zod

auth-validator.js:

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

const signupSchema = z.object({

    username: z

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

        .trim()

        .min(3, { message: "Name must be atleast 3 characters" })

        .max(255, { message: "Name must not be more than 255 characters" }),

    email: z

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

        .trim()

        .email({message:"Invalid email address"}),

    phone: z

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

        .trim()

        .min(10,{message:"Phone number must be atleast 10 characters"})

        .max(15,{message:"Phone number must not be more than 15 characters"}),

    password: z

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

    .min(7,{message:"Password must be atleast 7 characters"}),

})

module.exports = signupSchema;

z.object(): Creates a new Zod object schema.

username, email, phone, password: Defines the fields for the signup data with their respective validation rules.

Inside Server folder:

>>> mkdir middleware; cat > validate-middleware.js

validate-middleware.js

const { parse } = require("../validator/auth-validator");

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

    try {

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

        req.body = parseBody;

        next();

    } catch (error) {

        console.log(error)

        const message = error.errors[0].message;

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

    }

}

module.exports = validate;

auth-router:

const signupSchema = require('../validator/auth-validator')

const validate = require('../middleware/validate-middleware')

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

Explanation:

* Zod Schema (auth-validator.js):

The signupSchema defines the structure and validation rules for the registration data.

It ensures the username, email, phone, and password fields are provided and meet specific criteria (e.g., minimum length, valid email format).

* Validation Middleware (validate-middleware.js):

The validate function is a higher-order function that takes a Zod schema and returns an Express middleware function.

It uses the schema.parseAsync method to validate the request body. If validation passes, it proceeds to the next middleware or route handler. If validation fails, it responds with a 400 status code and an error message.

* Router Setup (auth-router.js):

The validate middleware is applied to the /register route. This ensures that incoming requests to this route are validated against the signupSchema.

If the validation passes, the request proceeds to the register controller method. If validation fails, an error response is sent back to the client.

Flow of the execution:

server.js-> starts port

auth-router-> calls upon the route which calls validator

validator-middleware -> calls upon the auth-validator

auth-validator -> checks for the validation and sends to the controller

Summary of Execution Flow:

Request Received: A client sends a POST request to /api/auth/register.

Router Handling: The request is handled by the authRouter, which matches the /register route.

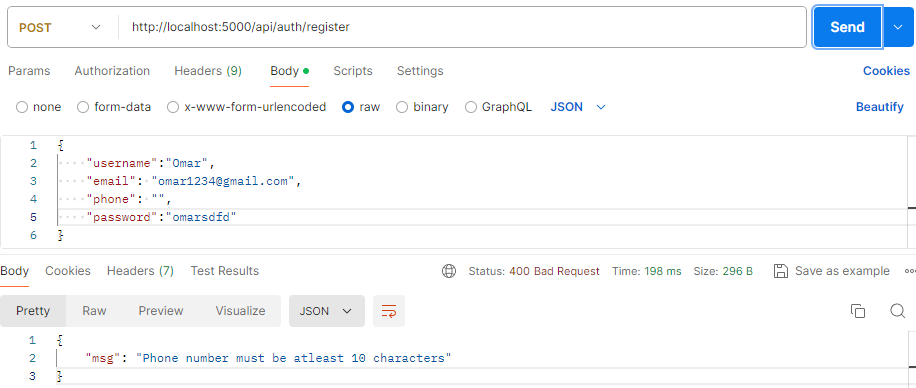
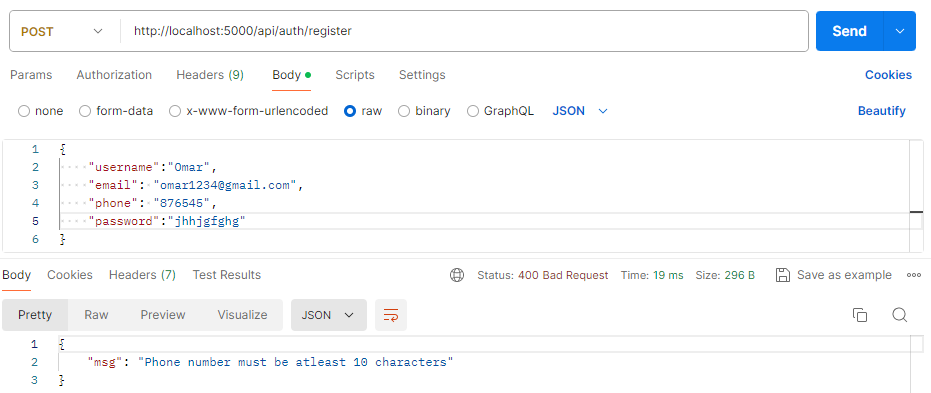
Validation Middleware: The validate middleware validates the request body against the signupSchema.

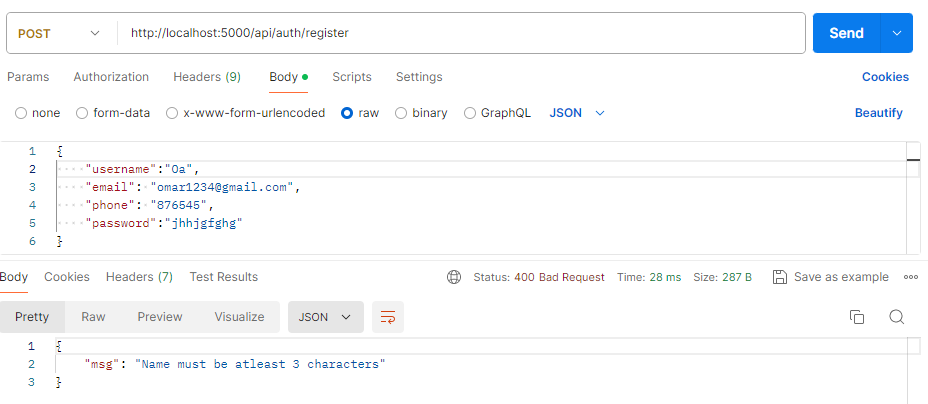
Validation Success: If validation passes, the middleware calls next() to proceed to the controller.

Controller Execution: The register controller handles the request, performs necessary checks, hashes the password, saves the new user, and sends a response.

Validation Failure: If validation fails, the middleware sends a 400 response with the validation error message.

Check for the post request:

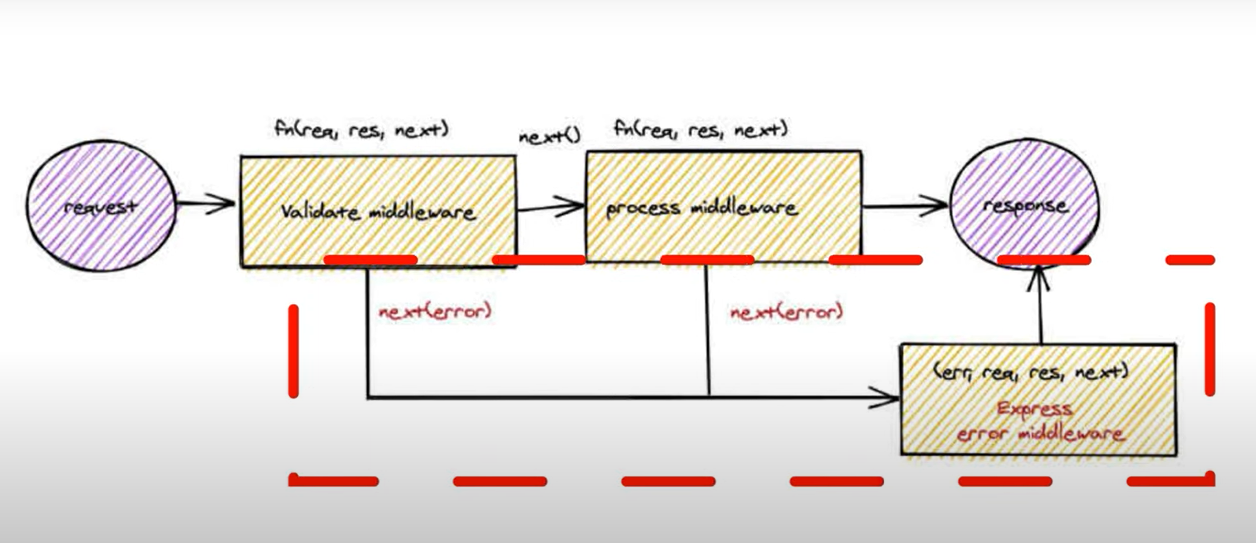


Error Handling:

>>> cat > error-middleware.js

Refrence:

<https://expressjs.com/en/guide/error-handling.html>



auth-controller.js:

register function->

        // res.status(500).json({ msg: "Server error" });

        next(error);

next ensures that the error message will go through the middleware defined.

In the error-middleware.js:

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;

validate-middleware:

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

    try {

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

        req.body = parseBody;

        next();

   catch (error) {

        console.log(error)

        const message = 'Fill the inputs properly'

        const extraDetails = error.errors[0].message;

        const status = 422;

        const err = {

            status,

            message,

extraDetails,

        }

        next(err)

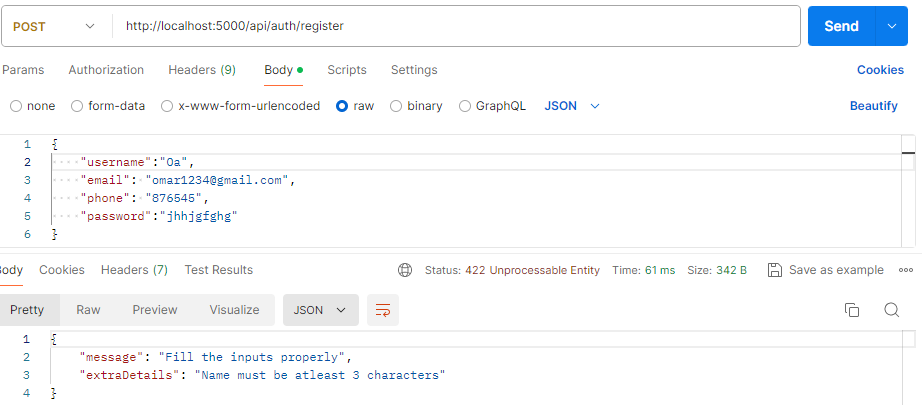
        // res.status(400).json({msg: message})

In the server.js: [To ensure middleware exist]

const errorMiddleware = require('./middleware/error-middleware')

app.use(errorMiddleware)

Postman:



Login Validator:

auth-validator:

const loginSchema = z.object({

    email: z

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

        .trim()

        .email({message:"Invalid email address"}),

    password:z

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

    .min(7,{message:"Password must be atleast 7 characters"}),

})

validate-middleware.js:

const { parse } = require("../validator/auth-validator");

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

    try {

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

        req.body = parseBody;

        next();

    } catch (error) {

        console.log(error)

        const message = 'Fill the inputs properly'

        const extraDetails = error.errors[0].message;

        const status = 422;

        const err = {

            status,

            message,

            extraDetails

        }

        next(err)

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

    }

}

module.exports = validate;

auth-router.js:

router.route('/').get(validate(validatorSchema.loginSchema),authControllers.home)

Contact Form:

>>> cd models

>>> cat > contact-model.js

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

const contactSchema = new Schema({

    username:{

        type: String,

        required:true

    },

    email:{

        type: String,

        required:true

    },

    message:{

        type: String,

        required:true

    }

});

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

module.exports = Contact;

>>> cd router

>>> cat > contact-router.js

const express = require('express')

const router = express.Router();

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

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

module.exports = router;

>>> cd controllers

>>> cat > contact-controller.js

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

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

    try {

        const response = req.body;

        await Contact.create(response);

        return res.status(200).json ({message: "Message Sent Successfully"})

    } catch (error) {

        return res.status(500).json ({message: "Message not Sent"})

    }

}

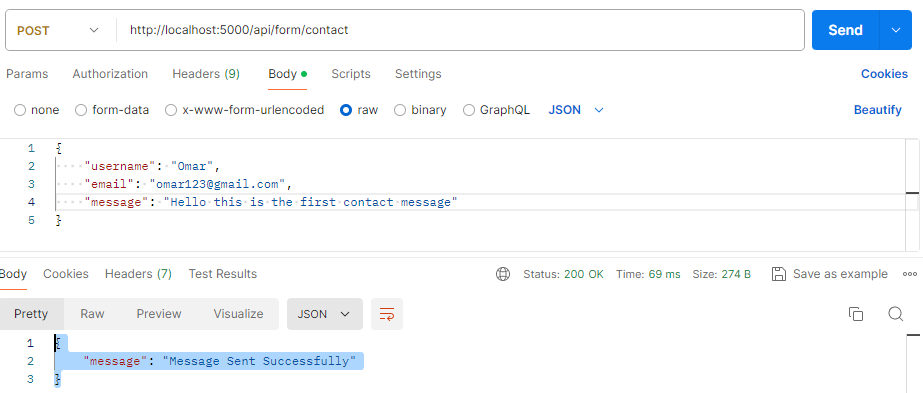
module.exports = contactForm

>>> server.js:

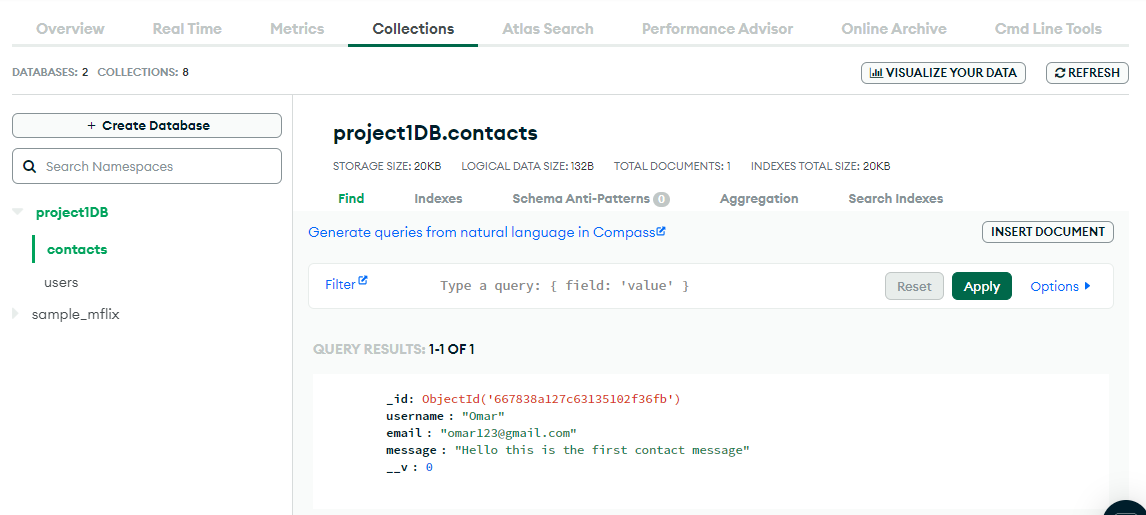
const contactRouter = require('./router/contact-router');

app.use('/api/form',contactRouter);

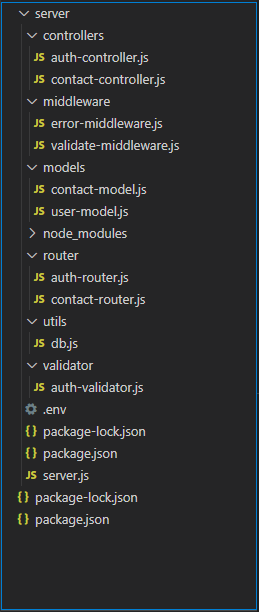
Check the request in postman by creating a new request:



Check it in your collections:



Total Backend till now:



FRONTEND:

React Learning Documentation:

<https://react.dev/learn>

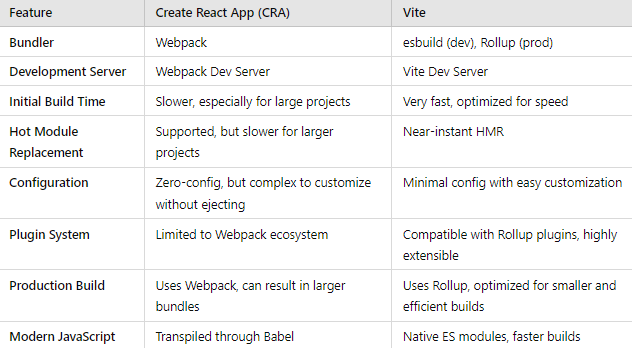
Vite:

<https://vitejs.dev/guide/>

Vite is a build tool and development server that is designed to provide a faster and more efficient development experience for modern web applications, particularly those built with frameworks like React. Here’s a detailed explanation of why you might choose Vite over other tools like Create React App (CRA) or traditional build systems:

**Benefits of Using Vite**

1. **Faster Development Server**:
   * **Instant HMR (Hot Module Replacement)**: Vite provides extremely fast hot module replacement, allowing you to see changes almost instantly without needing to reload the entire page.
   * **Efficient Dependency Handling**: Vite pre-bundles dependencies using esbuild, which is written in Go and is much faster than JavaScript-based bundlers.
2. **Optimized Production Build**:
   * **Smaller and Faster Builds**: Vite uses Rollup under the hood for production builds, which can result in smaller and more optimized output compared to other tools.
   * **Tree Shaking and Code Splitting**: Vite’s build process includes tree shaking and code splitting by default, ensuring only the necessary code is included in the final bundle.
3. **Support for Modern JavaScript Features**:
   * **Native ESM**: Vite uses native ES modules during development, which means you can take advantage of modern JavaScript features and syntax without needing to wait for a long build process.
   * **TypeScript and JSX**: Vite has out-of-the-box support for TypeScript and JSX, making it a great choice for React projects.
4. **Simple and Flexible Configuration**:
   * **Minimal Configuration**: Vite’s configuration is simpler and more straightforward compared to tools like Webpack, allowing you to get started quickly.
   * **Extensible**: Vite is highly extensible with a powerful plugin system that is compatible with Rollup plugins, making it easy to add additional functionality.
5. **Better Support for Large Projects**:
   * **Efficient Cold Starts**: Vite’s architecture is designed to handle large projects more efficiently, resulting in faster cold starts and better performance during development.



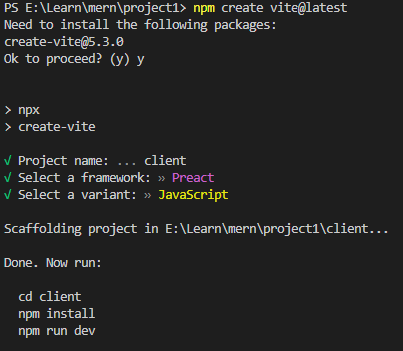
>>> cd ../ [Come outside server folder to parent folder(project1)]

>>> npm create vite@latest

Select projectName: client

Select Framework: React

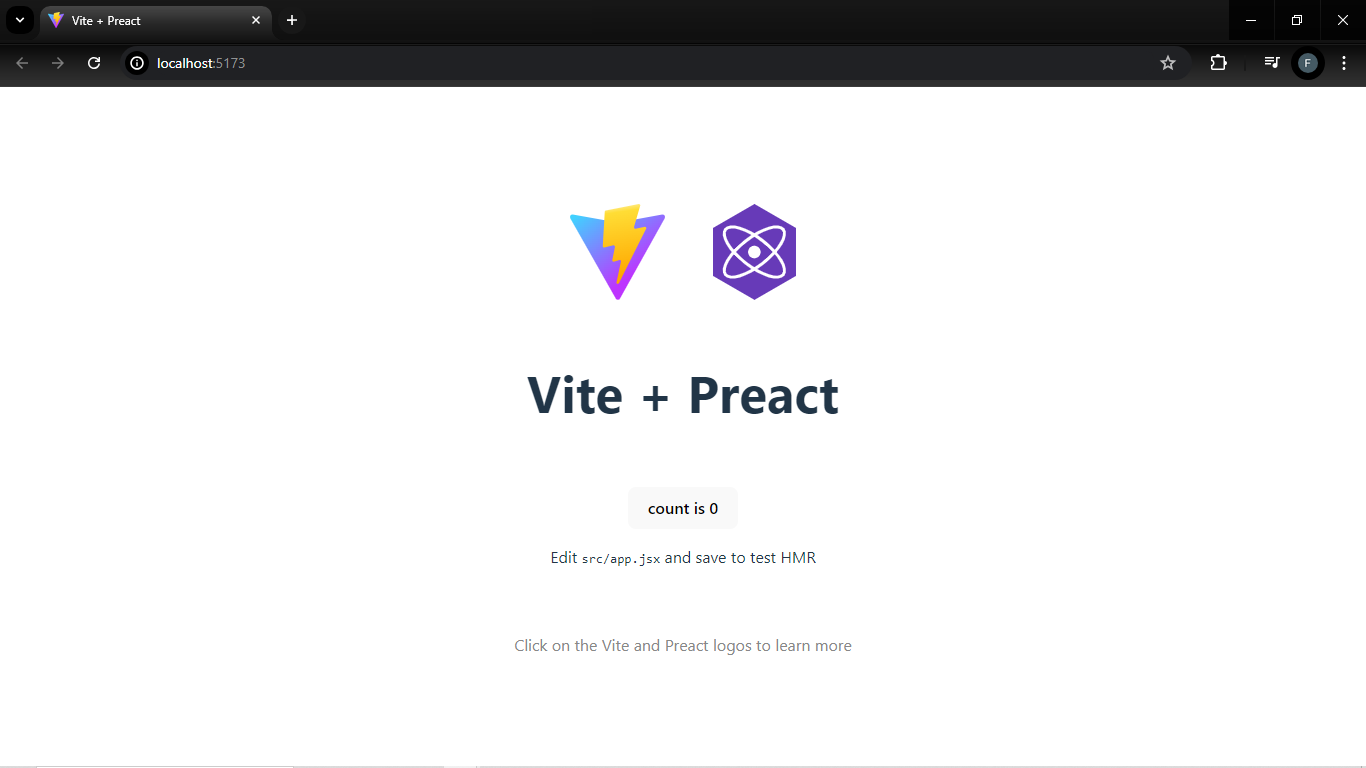
Select Variant: Javascript



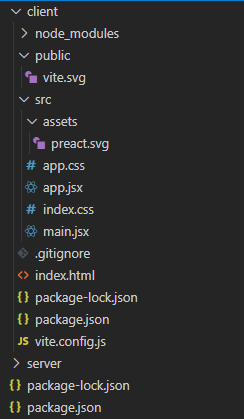
>>> cd client

>>> npm install

>>> npm run dev



Directory Structure:



When you create a React app with Vite, it sets up a minimal project structure designed to get you started quickly while providing the necessary configuration for development and production builds. Here’s an overview of the folders and files that are created:

**Project Structure**

After creating a new Vite project with React, the directory structure typically looks like this:

my-vite-react-app/

├── index.html

├── package.json

├── public/

│ └── (assets such as images, not processed by Vite)

├── src/

│ ├── App.css

│ ├── App.jsx

│ ├── index.css

│ ├── main.jsx

│ └── (other components and files you add)

├── vite.config.js

├── .gitignore

└── node\_modules/

**Detailed Explanation**

1. **index.html**
   * This is the main HTML file for your application. Unlike CRA, Vite uses this as the template for serving your app.
   * The file typically includes a root <div> where your React app will be mounted.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Vite + React</title>

</head>

<body>

<div id="root"></div>

<script type="module" src="/src/main.jsx"></script>

</body>

</html>

1. **package.json**
   * This file contains the project’s dependencies, scripts, and other metadata.
   * Scripts for starting the development server and building the project for production are included.

{

"name": "my-vite-react-app",

"version": "0.0.0",

"scripts": {

"dev": "vite",

"build": "vite build",

"serve": "vite preview"

},

"dependencies": {

"react": "^17.0.2",

"react-dom": "^17.0.2"

},

"devDependencies": {

"vite": "^2.6.4",

"@vitejs/plugin-react": "^1.0.7"

}

}

1. **public/**
   * This folder is used for static assets that are not processed by Vite.
   * Any files you put in this folder will be copied as-is to the output directory when you build your project.
2. **src/**
   * This is where your application's source code lives. It contains the main React components, styles, and other modules.
   * **App.css**: The CSS file for styling your main App component.
   * **App.jsx**: The main App component where you define your application’s main structure.

import React from 'react';

import './App.css';

function App() {

return (

<div className="App">

<h1>Hello Vite + React!</h1>

</div>

);

}

export default App;

* + **index.css**: Global styles for your application.
  + **main.jsx**: The entry point for your React application. This file initializes the React app and renders it to the DOM.

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

1. **vite.config.js**
   * The configuration file for Vite. This is where you can customize the Vite build process, add plugins, and configure other settings.

import { defineConfig } from 'vite';

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

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

export default defineConfig({

plugins: [react()]

});

1. **.gitignore**
   * A file specifying which files and directories should be ignored by version control (Git).

node\_modules/

dist/

.env

1. **node\_modules/**
   * This directory is created when you run npm install or yarn install and contains all the dependencies for your project.

**Conclusion**

The Vite project structure is designed to be simple and minimal, providing only the essential files and folders you need to get started with a React application. This structure is flexible and can be easily extended as your project grows. The key files include index.html for the entry HTML template, the src directory for your React components and application logic, and vite.config.js for customizing Vite’s behavior. This setup offers a fast, modern development experience with minimal configuration required.

React Router Dom.

>>> npm install react-router-dom

**Why is React Router Dom Used**

**React Router Dom** is a popular library used to handle routing in React applications. Routing refers to the ability to navigate between different parts of your application, each represented by a URL path. React Router Dom helps you create single-page applications (SPAs) with navigation that feels like a traditional multi-page website.

**Benefits of Using React Router Dom**

1. **Single-Page Application Navigation**: It allows your app to navigate between different views or pages without refreshing the entire page. This results in a smoother, faster user experience.
2. **Dynamic Routing**: You can define routes that dynamically change based on the app’s state or user actions, making your app more interactive and responsive.
3. **Nested Routing**: React Router Dom supports nested routes, allowing you to build complex user interfaces with hierarchical navigation structures.
4. **URL Parameters**: It allows you to extract parameters from the URL, making it possible to create dynamic pages that change based on the URL path or query parameters.
5. **History Management**: React Router Dom manages the browser history for you, enabling features like back and forward navigation, and custom navigation logic.

<BrowserRouter>

A <BrowserRouter> stores the current location in the browser's address bar using clean URLs and navigates using the browser's built-in history stack.

How It Works?

When the user navigates to the root URL (/), the Home component is displayed.

When the user navigates to /about, the About component is displayed.

When the user navigates to /contact, the Contact component is displayed.

The navigation links update the URL without refreshing the page, thanks to BrowserRouter.

This setup allows you to build complex navigation structures in your React application while maintaining a smooth, single-page application experience.

App.jsx:

import { BrowserRouter, Route, Routes } 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"

export function App() {

  return (

    <>

      <BrowserRouter>

        <Routes>

          <Route 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 />} />

        </Routes>

      </BrowserRouter>

    </>

  )

}

Explanation:

BrowserRouter: This component wraps your application and enables the routing capabilities. It uses the HTML5 history API to keep your UI in sync with the URL.

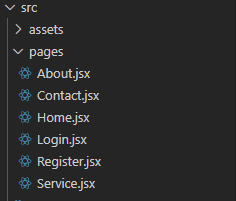
Routes: This component is a container for all your route definitions. It ensures that only one of the routes is rendered at a time.

Route: Each Route defines a path and the component to render when the URL matches that path.

**Breakdown of Routes**

* **<Route path="/" element={<Home />} />**: When the URL is /, the Home component is rendered.
* **<Route path="/about" element={<About />} />**: When the URL is /about, the About component is rendered.
* **<Route path="/contact" element={<Contact />} />**: When the URL is /contact, the Contact component is rendered.
* **<Route path="/service" element={<Service />} />**: When the URL is /service, the Service component is rendered.
* **<Route path="/register" element={<Register />} />**: When the URL is /register, the Register component is rendered.
* **<Route path="/login" element={<Login />} />**: When the URL is /login, the Login component is rendered.

Create a pages folder in your source file and create a files as:



Example file1[About.jsx]:

export const About = ()=>{

    return <h1>This is my about page</h1>

}

Do this same for everyone

Start your app

>>> npm run dev

YourURL/about etc:

<http://localhost:5173/about>

Adding NavBar:

Navbar is a component that would be same in all the pages so we will make a component folder and place the navbar there and call it in every function where required.

>>> cd src; mkdir components; cat > Navbar.jsx [PascalCase]

Navbar.js:

export const Navbar = () =>{

    return(

    <>

        <header>

            <div className="container">

                <div className="logo-brand">

                    <a href="/">KhanOmar</a>

                </div>

                <nav>

                    <ul>

                        <li><a href="/">Home</a></li>

                        <li><a href="/about">About</a></li>

                        <li><a href="/contact">Contact</a></li>

                        <li><a href="/service">Service</a></li>

                    </ul>

                </nav>

            </div>

        </header>

    </>

    );

}

Home.jsx:

import { Navbar } from "../components/Navbar"

export const Home = () => {

    return (

        <>

            <Navbar />

            <h1>This is Home Page</h1>

        </>

    );

}

NOTE:

To include in all pages and you’ll see the web page reloading when clicking on the nav. To avoid this:

All pages [include in app.jsx]:

Avoid reloading [Navlink]: <https://reactrouter.com/en/main/components/nav-link>

Instead of <a> use <Navlink>, instead of <href> use <to>

Example:

Don’t use: <li><a href="/">Home</a></li>

Use: <li><NavLink to="/">Home</NavLink></li>

Import:

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

app.jsx:

import { Navbar } from "./components/Navbar"

export function App() {

  return (

    <>

      <BrowserRouter>

        <Navbar />

        <Routes>

          <Route 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 />} />

        </Routes>

      </BrowserRouter>

    </>

  )

}

Navbar.jxs:

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

import "./Navbar.css"

export const Navbar = () =>{

    return(

    <>

        <header>

            <div className="container">

                <div className="logo-brand">

                    <NavLink to="/">KhanOmar</NavLink>

                </div>

                <nav>

                    <ul>

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

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

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

                        <li><NavLink to="/service">Service</NavLink></li>

                    </ul>

                </nav>

            </div>

        </header>

    </>

    );

}