FULL STACK WITH MERN

ASSIGNMENT-III

Aim: Created a RESTful API using Express.js and integrated it with MongoDB to perform CRUD operations on a database.

- Here we used two main files named “route.js” and “package.json”.
- We installed Thunder Client to create RESTful API and perform CRUD operations.
- The output of the operations performed are shown in the terminal rather than in the local host.

I. Creating Basic API:

- Open new folder in VSCode named “course”
- Initialize npm in the directory: Run the following command to create a package.json file, which will hold your project's dependencies.
  - npm init -y
- To install react the following command need to be written:

npm create-next-app text

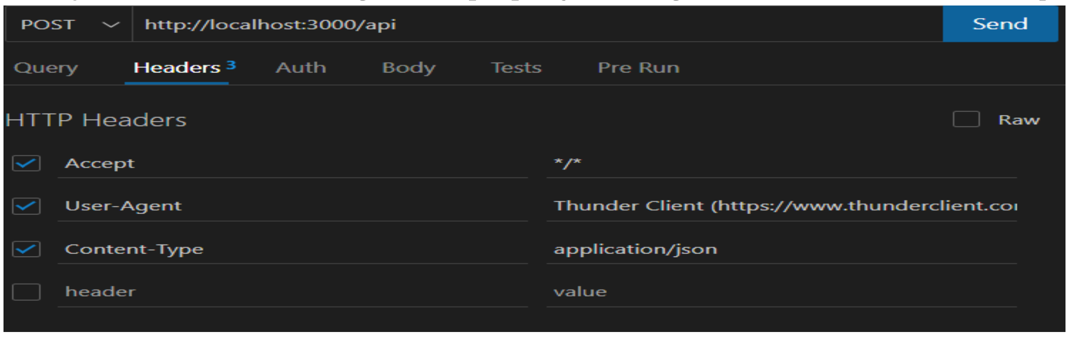
and create a folder in the app folder named “api” and in it create a file named “route.js”

Code in route .js

```
import { NextResponse } from "next/server";

export async function POST(request){
    const data = await request.json()
        console.log(data)
        return NextResponse.json({
            data: "HEY DEVELOPERS"
        })
}

import { NextResponse } from "next/server";

// Sample data to represent a resource (e.g., users)
let users = [];

export async function POST(request) {
    try {
        const data = await request.json();
        users.push(data); // Create a new resource (user)
        console.log("New user added:", data);
        return NextResponse.json({ success: true, message: "User created successfully" });
    } catch (error) {
        return NextResponse.error(error.message, { status: 500 });
    }
}

export async function GET(request) {
    try {
        return NextResponse.json(users); // Read all resources (users)
    } catch (error) {
        return NextResponse.error(error.message, { status: 500 });
    }
}

export async function PUT(request) {
    try {
        const data = await request.json();
        const { id } = data;
        if (id && users[id]) {
            users[id] = data; // Update an existing resource
            console.log("User updated:", data);
            return NextResponse.json({ success: true, message: "User updated successfully" });
        } else {
            return NextResponse.error("User not found", { status: 404 });
        }
    } catch (error) {
        return NextResponse.error(error.message, { status: 500 });
    }
}

export async function DELETE(request) {
    try {
        const data = await request.json();
        const { id } = data;
        if (id && users[id]) {
            users.splice(id, 1); // Delete an existing resource
            console.log("User deleted:", id);
            return NextResponse.json({ success: true, message: "User deleted successfully" });
        } else {
            return NextResponse.error("User not found", { status: 404 });
        }
    } catch (error) {
        return NextResponse.error(error.message, { status: 500 });
    }
}
```

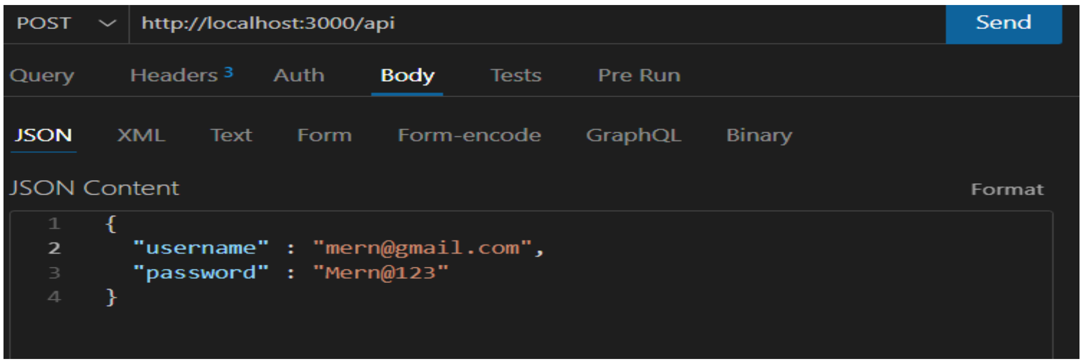
Go to extensions and install “Thunder Client”

- Open thunder client and click “New Request”
- Add a new HTTP request to the file.
- Set the request method to POST.
- Set the request URL to the “http://localhost:3000/api”.
- Add request headers and request body data in JSON format.
- Send the request.
- You should receive a response from the API endpoint containing the JSON data "HEY DEVELOPERS".
- Make sure to replace "http://your-api-endpoint-url" with the actual URL of your API endpoint when making requests in Thunder Client. Additionally, ensure that your Next.js server is running and properly configured to handle API requests


POST http://localhost:3000/api

HTTP Headers

✓	Accept	*/*
✓	User-Agent	Thunder Client (https://www.thunderclient.coı
✓	Content-Type	application/json
	header	value

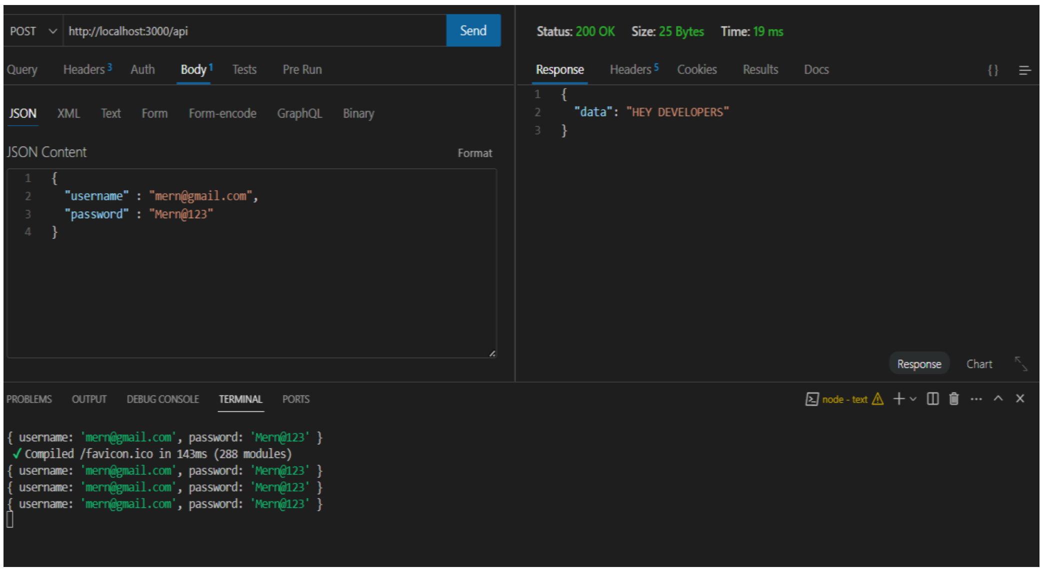

POST http://localhost:3000/api

JSON Content

```
{
  "username" : "mern@gmail.com",
  "password" : "Mern@123"
}
```

Output:

The output will be display in vscode not on google chrome. The data is saved in the server

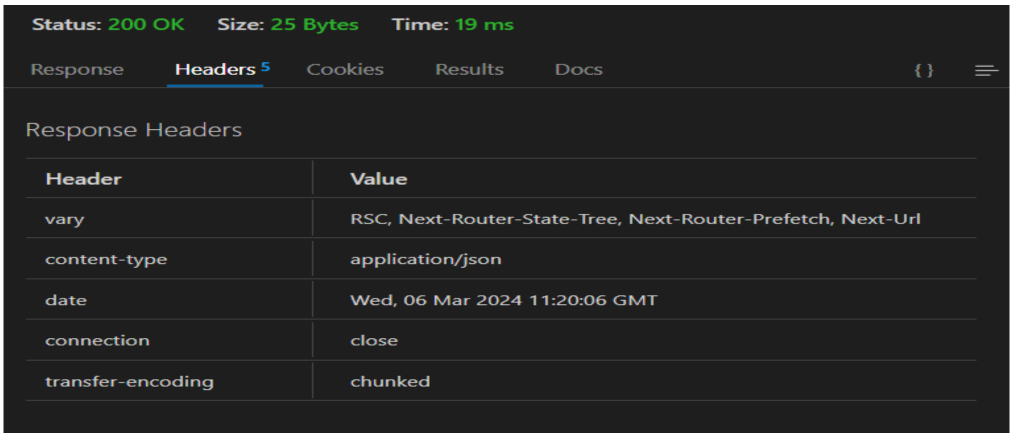

POST http://localhost:3000/api

Status: 200 OK Size: 25 Bytes Time: 19 ms

JSON Content
```
{
  "username" : "mern@gmail.com",
  "password" : "Mern@123"
}
```

Response
```
{
  "data": "HEY DEVELOPERS"
}
```

TERMINAL
```
{ username: 'mern@gmail.com', password: 'Mern@123' }
 ✓ Compiled /favicon.ico in 143ms (288 modules)
{ username: 'mern@gmail.com', password: 'Mern@123' }
{ username: 'mern@gmail.com', password: 'Mern@123' }
{ username: 'mern@gmail.com', password: 'Mern@123' }
```


Status: 200 OK Size: 25 Bytes Time: 19 ms

Response Headers

Header	Value
vary	RSC, Next-Router-State-Tree, Next-Router-Prefetch, Next-Url
content-type	application/json
date	Wed, 06 Mar 2024 11:20:06 GMT
connection	close
transfer-encoding	chunked

Finally, the function returns a JSON response using NextResponse.json(). This response contains an object with a data property set to the string "HEY DEVELOPERS". This is a simple example response indicating a successful handling of the request.

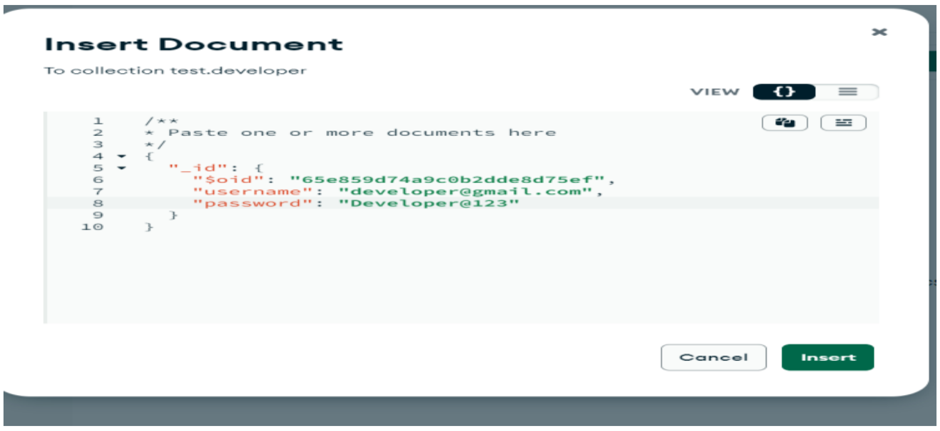
When this API endpoint is invoked with a POST request, it will log the incoming JSON data to the console (if any), and then it will respond with a JSON object containing the message "HEY DEVELOPERS". This response can be received by the client making the request, indicating that the request was successfully processed by the server.

II. Integrate MongoDB with your Express.js application to store and retrieve data. Test your API using tools like Postman or curl to ensure all endpoints are functioning correctly.

After installing MongoDB

Create a database name test and create collection name as developer

Add file “insert document” and write



```
/**
* Paste one or more documents here
*/
{
  "_id": {
    "$oid": "65e859d74a9c0b2dde8d75ef",
    "username": "developer@gmail.com",
    "password": "Developer@123"
  }
}
```

Integrating MongoDB with Express js

Code

```
const mongoose = require('mongoose')

import { NextResponse } from 'next/server'

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

const UserSchema = new mongoose.Schema({
    _id: Object,
    name: String,
    description: String
})

const UserModel = mongoose.model("users")

export async function POST(request) {
    let data = ""

    const users = await UserModel.find({})
    console.log(users)
    return NextResponse.json({"message":users})
}
```

CRUD Operations:

POST Method:

This function handles the creation of a new resource (user). It parses the incoming JSON data from the request body, adds the new user to the users array, and returns a success message.

GET Method:

This function handles retrieving all resources (users). It simply returns the users array containing all existing users.

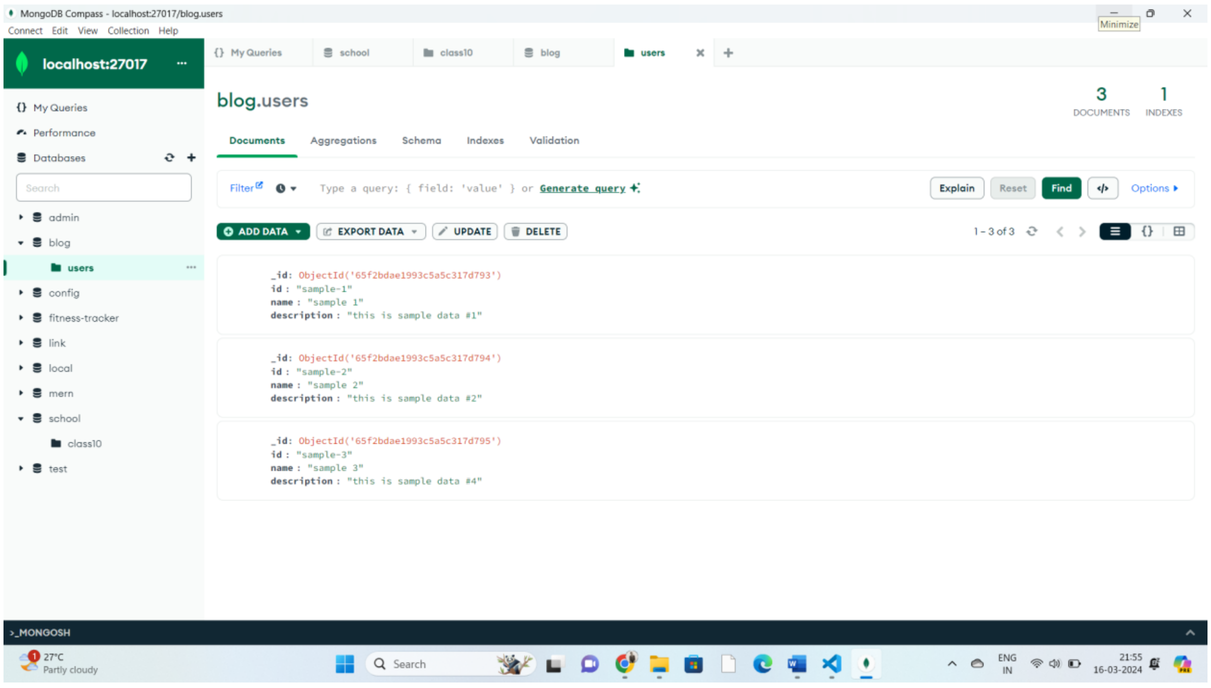
PUT Method:

This function handles updating an existing resource (user). It parses the incoming JSON data, extracts the user ID, checks if the user exists in the users array, updates the user data, and returns a success message.

DELETE Method:

This function handles deleting an existing resource (user). It parses the incoming JSON data, extracts the user ID, checks if the user exists in the users array, removes the user from the array, and returns a success message.

Database in MongoDB


MongoDB Compass - localhost:27017/blog.users
Connect Edit View Collection Help
localhost:27017
My Queries
Performance
Databases
admin
blog
users
config
fitness-tracker
link
local
mern
school
class10
test
My Queries school class10 blog users
blog.users
3 DOCUMENTS
1 INDEXES
Documents Aggregations Schema Indexes Validation
Filter
Type a query: { field: 'value' } or Generate query
Explain Reset Find Options
ADD DATA EXPORT DATA UPDATE DELETE
1 - 3 of 3
>_MONGOSH

```
_id: ObjectId('65f2bdae1993c5a5c317d793')
id : "sample-1"
name : "sample 1"
description : "this is sample data #1"

_id: ObjectId('65f2bdae1993c5a5c317d794')
id : "sample-2"
name : "sample 2"
description : "this is sample data #2"

_id: ObjectId('65f2bdae1993c5a5c317d795')
id : "sample-3"
name : "sample 3"
description : "this is sample data #4"
```

Output on localhost

When running the provided code on localhost, several points can be observed regarding its output:

API Endpoint URLs:

Each route function (`POST`, `GET`, `PUT`, `DELETE`) represents a different API endpoint. These endpoints can be accessed through specific URLs when running the Next.js application locally.The server is running on `http://localhost:3000`, then the endpoint for creating a new resource (POST) might be `http://localhost:3000/api/create`.

HTTP Requests and Responses:

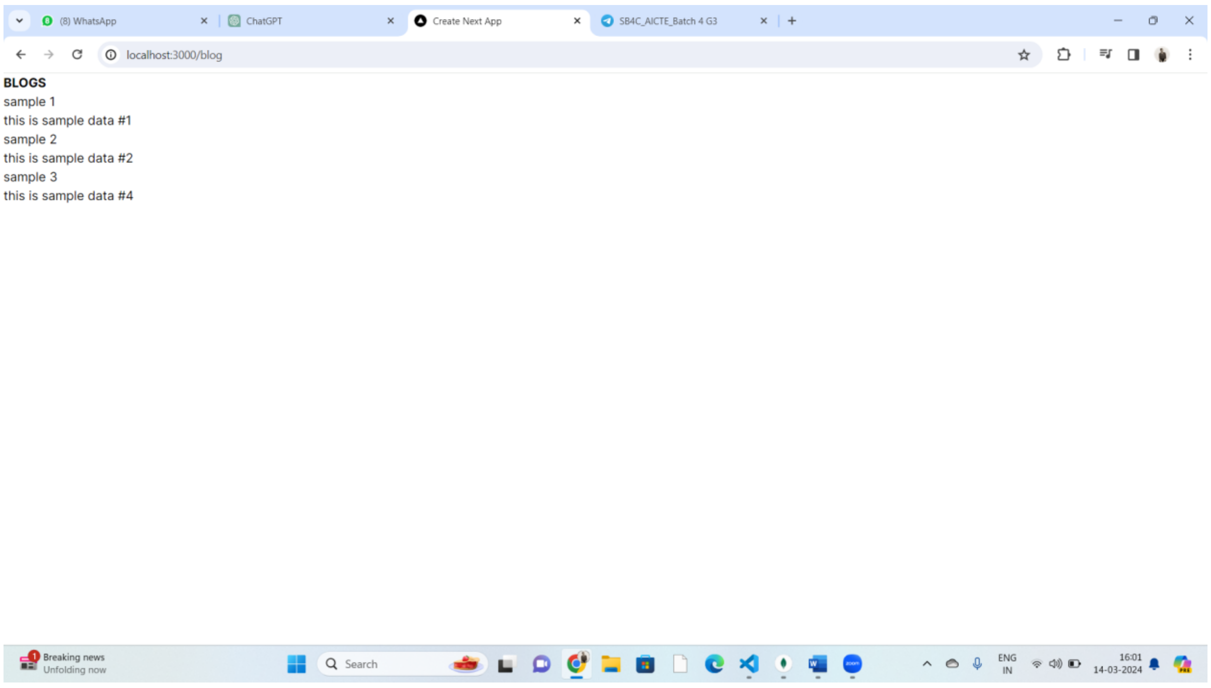
When making HTTP requests to these endpoints (e.g., using Thunder Client or any other HTTP client), you can observe the corresponding HTTP responses. Successful requests will receive JSON responses indicating the success of the operation (e.g., "User created successfully"). Unsuccessful requests (e.g., due to invalid input or missing resources) will receive appropriate error messages and status codes (e.g., 404 for "User not found").

Console Output:

The route functions include `console.log()` statements to log certain information to the console. For example, when creating a new user (`POST` request), the data of the newly created user will be logged to the console. Similarly, when updating or deleting a user, information about the affected user will be logged.

Data Persistence:

The code maintains a `users` array to store the resource data (e.g., user information). When the server is running locally, any changes made to this array (e.g., creating, updating, or deleting users) will persist only for the duration of the server's runtime. If the server is restarted, the `users` array will be reset to its initial state, losing any changes made during the previous runtime.



BLOGS
sample 1
this is sample data #1
sample 2
this is sample data #2
sample 3
this is sample data #4

Testing and Debugging:

Running the Next.js application locally allows for easy testing and debugging of the API endpoints. Developers can use tools like Thunder Client, Postman, or cURL to send HTTP requests to the endpoints and observe the responses. The console output provides additional visibility into the execution flow and helps in debugging any issues encountered during development.