**MERN Stack (MongoDB, Express.js, React, Node.js)**

1. **Node.js: Write a basic Node.js server that listens on port 3000 and returns a "Hello, World!" message when the root URL is accessed.**

**Initialized Node Project:** npm init -y

**CODE:**

//MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\1.Node.js\server.js

const http = require('http');

const server = http.createServer((req, res) => {

    res.writeHead(200, { 'Content-Type': 'text/plain' });

    if (req.url === '/') {

        res.end("Hello, World!\n");

    } else {

        res.writeHead(404, { 'Content-Type': 'text/plain' });

        res.end("Not Found\n");

    }

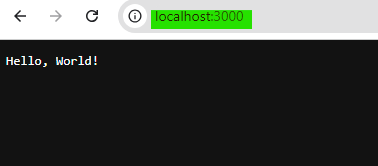
});

server.listen(3000, () => {

    console.log('Server is running on http://localhost:3000');

});

**Output:**



1. **Express.js: Create a simple REST API using Express.js with a single route /users that returns a JSON list of users.**

**Initialized Node Project:** npm init -y

**Installed Packages:** express

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\2.Express.js\server.js

const express = require('express');

const app = express();

const users = require("./listOfUsers");

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

    res.status(200).json(users);

});

app.listen(3000, () => {

    console.log('Server is running on http://localhost:3000');

});

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\2.Express.js\listOfUsers.js

const users = [

    { id: 1, name: 'John Doe', email: 'john.doe@example.com' },

    { id: 2, name: 'Jane Smith', email: 'jane.smith@example.com' },

    { id: 3, name: 'Mike Johnson', email: 'mike.johnson@example.com' },

    { id: 4, name: 'Micheal Cook', email: 'micheal.cook@example.com' },

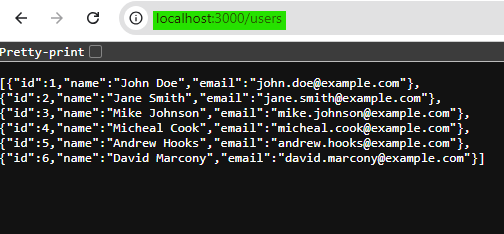
    { id: 5, name: 'Andrew Hooks', email: 'andrew.hooks@example.com' },

    { id: 6, name: 'David Marcony', email: 'david.marcony@example.com' }

];

module.exports = users;

**Output:**



1. **React: Build a basic React component that fetches the list of users from the /users API route (from question 2) and displays them in a table.**

**Created React App:** npm create vite@latest

**Installed Packages:** cors

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\3.React\client\src\App.jsx

import UserData from "./components/UserData";

function App() {

  return (

    <>

      <UserData />

    </>

  )

}

export default App

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\3.React\client\src\components\UserData.jsx

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

function UserData() {

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

    const [loading, setLoading] = useState(true);

    const [error, setError] = useState(null);

    useEffect(() => {

        fetch('http://localhost:3000/users')

            .then((response) => {

                if (!response.ok) {

                    throw new Error('Network response was not ok');

                }

                return response.json();

            })

            .then((data) => {

                setUsers(data);

                setLoading(false);

            })

            .catch((error) => {

                setError(error);

                setLoading(false);

            });

    }, []);

    if (loading) {

        return <p>Loading...</p>;

    }

    if (error) {

        return <p>Error: {error.message}</p>;

    }

    return (

        <div>

            <h1>User List</h1>

            <table border="1" cellPadding="10">

                <thead>

                    <tr>

                        <th>ID</th>

                        <th>Name</th>

                        <th>Email</th>

                    </tr>

                </thead>

                <tbody>

                    {users.map((user) => (

                        <tr key={user.id}>

                            <td>{user.id}</td>

                            <td>{user.name}</td>

                            <td>{user.email}</td>

                        </tr>

                    ))}

                </tbody>

            </table>

        </div>

    );

}

export default UserData;

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\2.Express.js\server.js

const cors = require("cors");

const corsOptions = {

    origin: "http://localhost:5173",

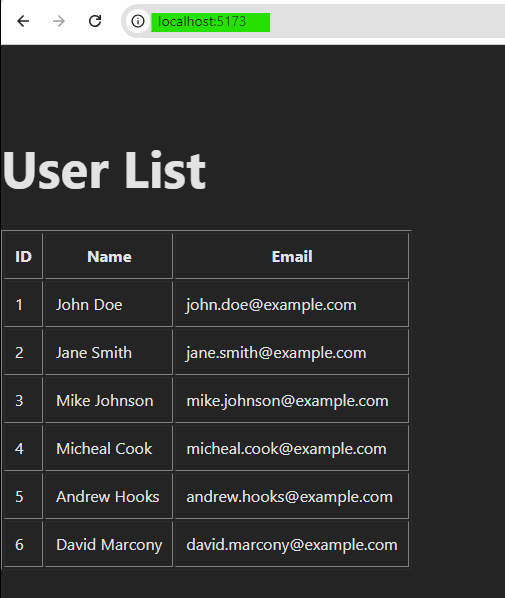
    methods: "GET, POST, PUT, DELETE, PATCH, HEAD",

    credentials: true,

};

app.use(cors(corsOptions));

**Output:**



1. **MongoDB: Create a MongoDB schema for storing user data (name, email, age), and write a script to insert a new user into the collection.**

**Initialized Node Project:** npm init -y

**Installed Packages:** mongoose

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\4.MongoDB\insertUser.js

const mongoose = require('mongoose');

const User = require('./userModel');

mongoose.connect('mongodb://127.0.0.1:27017/usersDB')

.then(() => {

  console.log('Connected to MongoDB');

}).catch((error) => {

  console.error('Error connecting to MongoDB:', error);

});

const insertUser = async () => {

  try {

    const newUser = new User({

      name: 'Vivek Singh',

      email: 'vivek.singh@example.com',

      age: 25

});

    const savedUser = await newUser.save();

    console.log('User inserted:', savedUser);

  } catch (error) {

    console.error('Error inserting user:', error);

  } finally {

    mongoose.connection.close();

  }

};

insertUser();

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\4.MongoDB\userModel.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

  name: {

    type: String,

    required: true

  },

  email: {

    type: String,

    required: true,

    unique: true

  },

  age: {

    type: Number,

    required: true

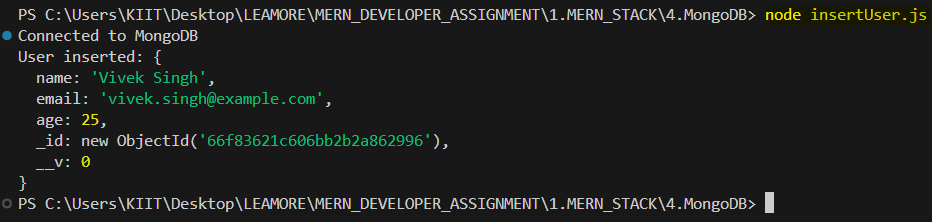
  }

});

const User = mongoose.model('User', userSchema);

module.exports = User;

**Output:**





1. **Express.js + MongoDB: Create an Express.js route to fetch a user by their email from the MongoDB database.**

**Initialized Node Project:** npm init -y

**Installed Packages:** express, mongoose

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\5.Express.jsAndMongoDB\server.js

const express = require("express");

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

require("./database");

const app = express();

app.get('/user/:email', async (req, res) => {

    const email = req.params.email;

    try {

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

        if (!user) {

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

        }

        res.json(user);

    } catch (error) {

        console.error('Error fetching user:', error);

        res.status(500).json({ message: 'Internal server error' });

    }

});

app.listen(3000, () => {

    console.log('Server is running on http://localhost:3000');

});

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\5.Express.jsAndMongoDB\models\user.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

    name: {

        type: String,

        required: true

    },

    email: {

        type: String,

        required: true,

        unique: true

    },

    age: {

        type: Number,

        required: true

    }

});

const User = mongoose.model('User', userSchema);

module.exports = User;

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\5.Express.jsAndMongoDB\database\index.js

const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/usersDB')

.then(() => {

  console.log('Connected to MongoDB');

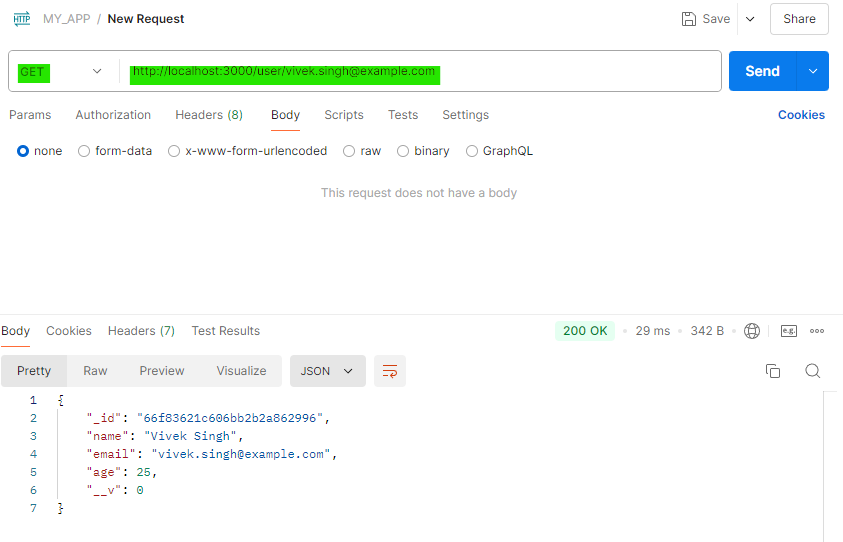
}).catch((error) => {

  console.error('Error connecting to MongoDB:', error);

});

**Output:**





1. **React + State Management: Build a form component in React that allows users to submit their name, email, and age. On submission, send the data to the backend API and update the state to display the new user.**

**Initialized Node Project:** npm init -y

**Installed Packages:** express, mongoose, cors

**Created React App:** npm create vite@latest

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\6.ReactAndState\_Management\server\server.js

const cors = require("cors");

const corsOptions = {

    origin: "http://localhost:5173",

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

    credentials: true,

};

app.use(cors(corsOptions));

app.use(express.json());

app.post('/users', async (req, res) => {

    const { name, email, age } = req.body;

    try {

        const newUser = new User({ name, email, age });

        const savedUser = await newUser.save();

        res.status(201).json(savedUser);

    } catch (error) {

        console.error('Error creating user:', error);

        res.status(500).json({ message: 'Internal server error' });

    }

});

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\6.ReactAndState\_Management\client\src\App.jsx

import Form from "./components/Form";

function App() {

  return (

    <>

      <Form />

    </>

  )

}

export default App

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\6.ReactAndState\_Management\client\src\components\Form.jsx

import React, { useState } from 'react';

import User from "./User";

function Form() {

  const [formData, setFormData] = useState({

    name: '',

    email: '',

    age: ''

  });

  const [newUser, setNewUser] = useState(null);

  const [loading, setLoading] = useState(false);

  const [error, setError] = useState(null);

  const handleChange = (e) => {

    const { name, value } = e.target;

    setFormData((prevData) => ({

      ...prevData,

      [name]: value

    }));

  };

  const handleSubmit = async (e) => {

    e.preventDefault();

    setLoading(true);

    try {

      const response = await fetch('http://localhost:3000/users', {

        method: 'POST',

        headers: {

          'Content-Type': 'application/json'

        },

        body: JSON.stringify(formData)

      });

      if (!response.ok) {

        throw new Error('Failed to add user');

      }

      const result = await response.json();

      setNewUser(result);

      setFormData({ name: '', email: '', age: '' });

    } catch (error) {

      setError(error.message);

    } finally {

      setLoading(false);

    }

  };

  return (

    <div>

      <h1>Add New User</h1>

      {error && <p style={{ color: 'red' }}>Error: {error}</p>}

      <form onSubmit={handleSubmit}>

        <div>

          <label>Name: </label>

          <input

            type="text"

            name="name"

            value={formData.name}

            onChange={handleChange}

            required

          />

        </div>

        <div>

          <label>Email: </label>

          <input

            type="email"

            name="email"

            value={formData.email}

            onChange={handleChange}

            required

          />

        </div>

        <div>

          <label>Age: </label>

          <input

            type="number"

            name="age"

            value={formData.age}

            onChange={handleChange}

            required

          />

        </div>

        <button type="submit" disabled={loading}>

          {loading ? 'Submitting...' : 'Submit'}

        </button>

      </form>

      {newUser && (

        <User name={newUser.name} email={newUser.email} age={newUser.age}/>

      )}

    </div>

  );

}

export default Form;

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\6.ReactAndState\_Management\client\src\components\User.jsx

function User({name, email, age}){

    return(

        <div>

          <h2>New User Added:</h2>

          <p>Name: {name}</p>

          <p>Email: {email}</p>

          <p>Age: {age}</p>

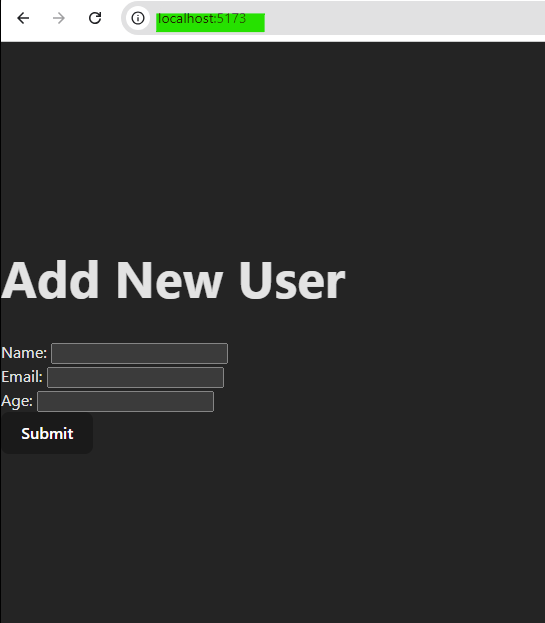
        </div>

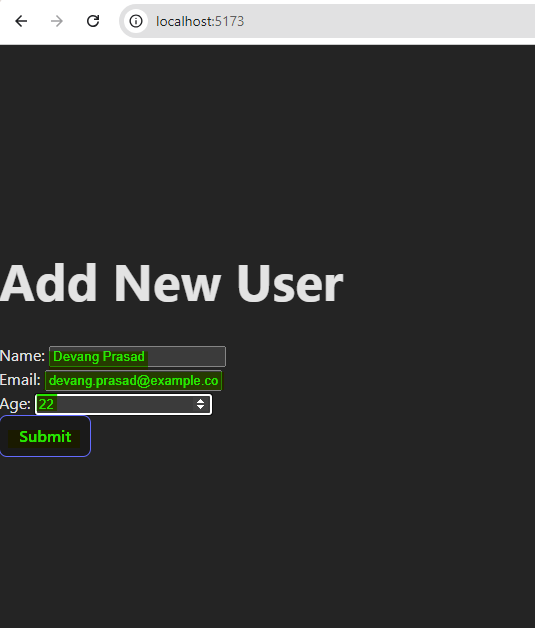
    );

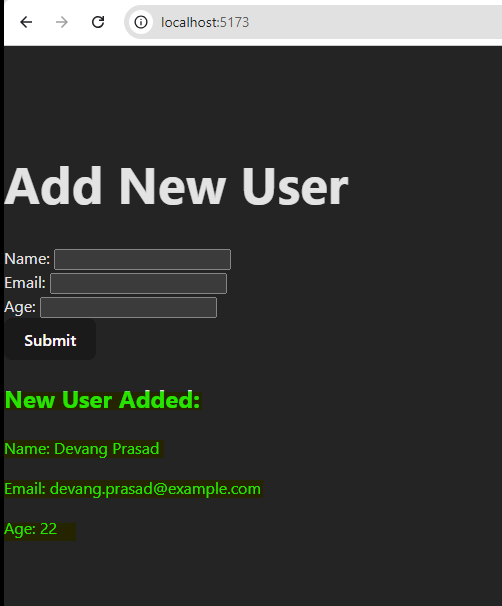
}

export default User;

**Output:**







1. **React Routing: Set up React Router in an application to navigate between a Home page and a Users page.**

**Initialized Node Project:** npm init -y

**Installed Packages:** express, mongoose, cors, react-router-dom

**Created React App:** npm create vite@latest

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\7.React\_Routing\client\src\main.jsx

import { StrictMode } from 'react'

import { createRoot } from 'react-dom/client'

import App from './App.jsx'

import { BrowserRouter as Router } from 'react-router-dom';

createRoot(document.getElementById('root')).render(

  <Router>

    <StrictMode>

      <App />

    </StrictMode>

  </Router>

)

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\7.React\_Routing\client\src\App.jsx

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

import Home from './components/Home';

import Users from './components/Users';

function App() {

  return (

    <>

      <div>

        <nav>

          <ul>

            <li>

              <Link to="/">Home</Link>

            </li>

            <li>

              <Link to="/users">Users</Link>

            </li>

          </ul>

        </nav>

        <Routes>

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

          <Route path="/users" element={<Users />} />

        </Routes>

      </div>

    </>

  )

}

export default App

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\7.React\_Routing\client\src\components\Home.jsx

function Home() {

    return (

        <div>

            <h1>Home Page</h1>

            <p>Welcome to the Home page!</p>

        </div>

    );

}

export default Home;

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\7.React\_Routing\client\src\components\Users.jsx

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

function Users() {

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

    const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

    useEffect(() => {

        fetch('http://localhost:3000/getallusers')

            .then((response) => {

                if (!response.ok) {

                    throw new Error('Network response was not ok');

                }

                return response.json();

            })

            .then((data) => {

                setUsers(data);

                setLoading(false);

            })

            .catch((error) => {

                setError(error);

                setLoading(false);

            });

}, []);

    if (loading) {

        return <p>Loading...</p>;

}

    if (error) {

        return <p>Error: {error.message}</p>;

}

    return (

        <div>

            <h1>Users Page</h1>

            <table border="1" cellPadding="10">

                <thead>

                    <tr>

                        <th>ID</th>

                        <th>Name</th>

                        <th>Email</th>

                    </tr>

                </thead>

                <tbody>

                    {users.map((user) => (

                        <tr key={user.id}>

                            <td>{user.id}</td>

                            <td>{user.name}</td>

                            <td>{user.email}</td>

                        </tr>

                    ))}

                </tbody>

            </table>

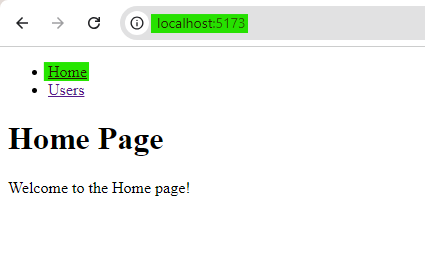
        </div>

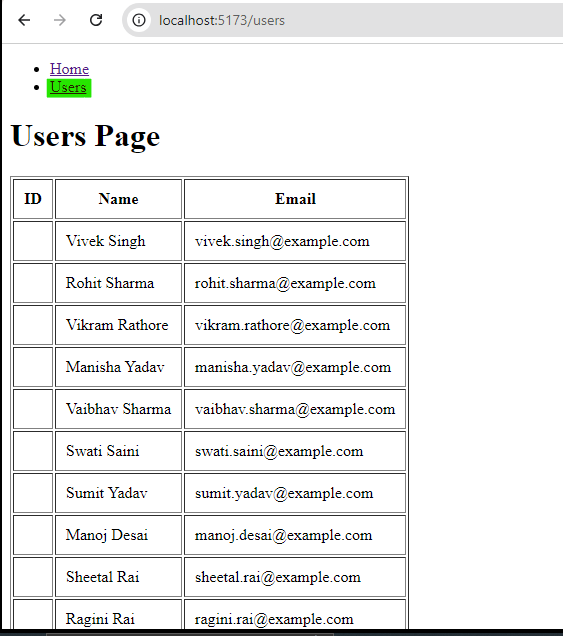
    );

}

export default Users;

**Output:**





1. **RESTful API Design: Design and implement a REST API in Express.js for a simple blog application with routes for creating, reading, updating, and deleting blog posts.**

**Initialized Node Project:** npm init -y

**Installed Packages:** express, mongoose, cors

**Routes for CRUD Operation:**

* **Create a Post** (POST /api/blog/posts): Adds a new blog post.
* **Get All Posts** (GET /api/blog/posts): Retrieves all blog posts.
* **Get a Single Post** (GET /api/blog/posts/:id): Fetches a specific post by its ID.
* **Update a Post** (PUT /api/blog/posts/:id): Updates a post by its ID.
* **Delete a Post** (DELETE /api/blog/posts/:id): Deletes a post by its ID.

**CODE:**

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\8.RESTful\_API\_Design\server.js

const express = require("express");

const cors = require("cors");

const blogRoute  = require("./router/blog-router");

require("./database");

const app = express();

const corsOptions = {

    origin: "http://localhost:5173",

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

    credentials: true,

};

app.use(cors(corsOptions));

app.use(express.json());

app.use("/api/blog", blogRoute);

app.listen(3000, () => {

    console.log('Server is running on http://localhost:3000');

});

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\8.RESTful\_API\_Design\database\index.js

const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/blogDB')

.then(() => {

  console.log('Connected to MongoDB');

}).catch((error) => {

  console.error('Error connecting to MongoDB:', error);

});

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\8.RESTful\_API\_Design\router\blog-router.js

const express = require('express');

const router = express.Router();

const blogController = require("../controllers/blog-controller");

router.route("/posts").post(blogController.createNewPost);

router.route("/posts").get(blogController.getAllPosts);

router.route("/posts/:id").get(blogController.getSinglePostById);

router.route("/posts/:id").put(blogController.updatePostById);

router.route("/posts/:id").delete(blogController.deletePostById);

module.exports = router;

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\8.RESTful\_API\_Design\controllers\blog-controller.js

const Post = require("../models/Post");

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

    const { title, content, author } = req.body;

    try {

        const newPost = new Post({ title, content, author });

        const savedPost = await newPost.save();

        res.status(201).json(savedPost);

    } catch (error) {

        res.status(500).json({ message: 'Error creating post', error });

    }

};

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

    try {

        const posts = await Post.find();

        res.status(200).json(posts);

    } catch (error) {

        res.status(500).json({ message: 'Error fetching posts', error });

    }

};

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

    try {

        const post = await Post.findById(req.params.id);

        if (!post) return res.status(404).json({ message: 'Post not found' });

        res.status(200).json(post);

    } catch (error) {

        res.status(500).json({ message: 'Error fetching post', error });

    }

};

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

    const { title, content, author } = req.body;

    try {

        const updatedPost = await Post.findByIdAndUpdate(

            req.params.id,

            { title, content, author },

            { new: true, runValidators: true }

        );

        if (!updatedPost) return res.status(404).json({ message: 'Post not found' });

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

    } catch (error) {

        res.status(500).json({ message: 'Error updating post', error });

    }

};

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

    try {

        const deletedPost = await Post.findByIdAndDelete(req.params.id);

        if (!deletedPost) return res.status(404).json({ message: 'Post not found' });

        res.status(200).json({ message: 'Post deleted successfully' });

    } catch (error) {

        res.status(500).json({ message: 'Error deleting post', error });

    }

};

module.exports = { createNewPost, getAllPosts, getSinglePostById, updatePostById, deletePostById };

// MERN\_DEVELOPER\_ASSIGNMENT\1.MERN\_STACK\8.RESTful\_API\_Design\models\Post.js

const mongoose = require('mongoose');

const postSchema = new mongoose.Schema({

  title: {

    type: String,

    required: true

  },

  content: {

    type: String,

    required: true

  },

  author: {

    type: String,

    required: true

  },

  createdAt: {

    type: Date,

    default: Date.now

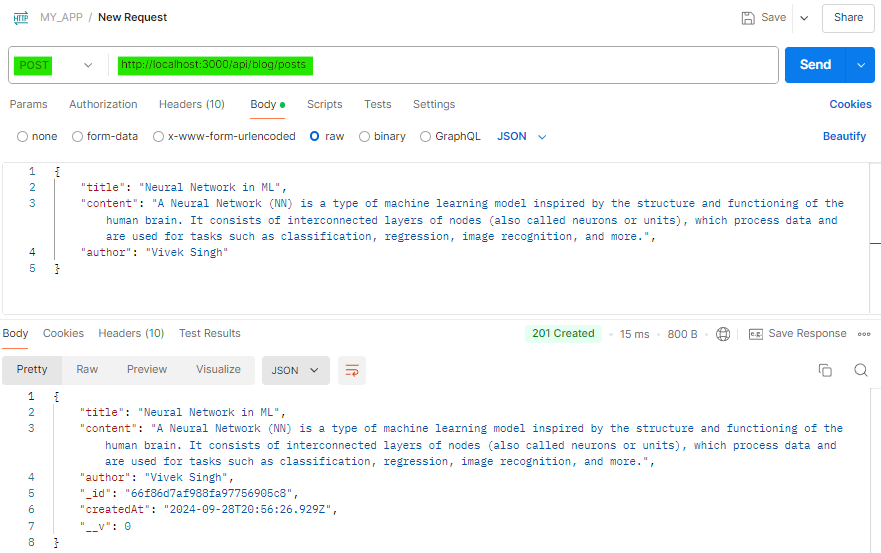
  }

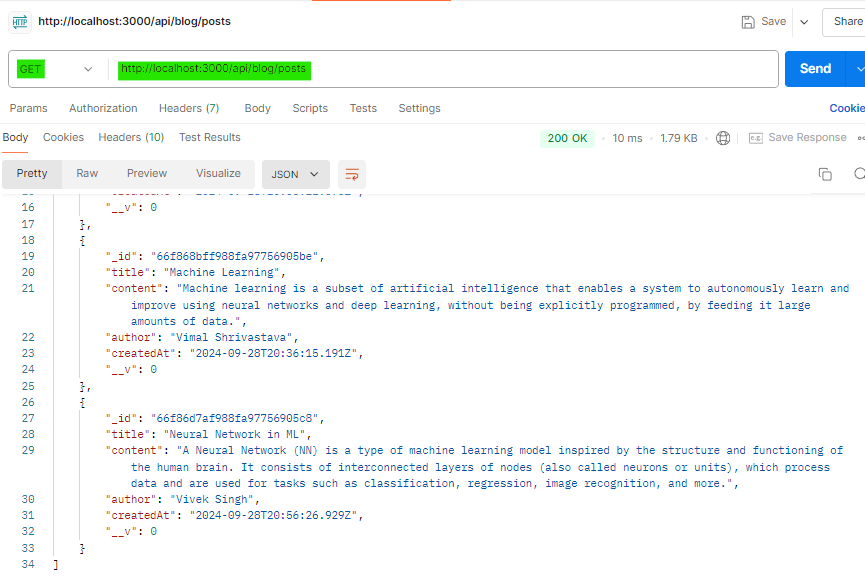
});

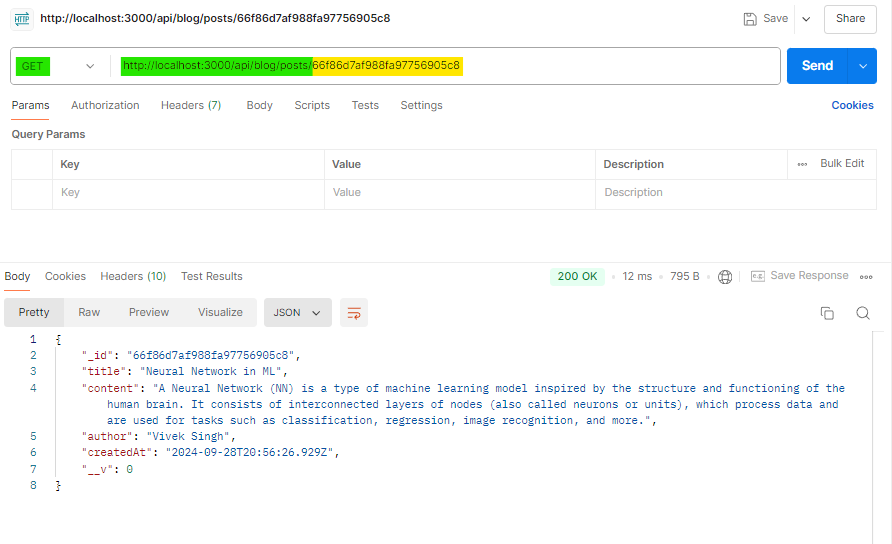
const Post = mongoose.model('Post', postSchema);

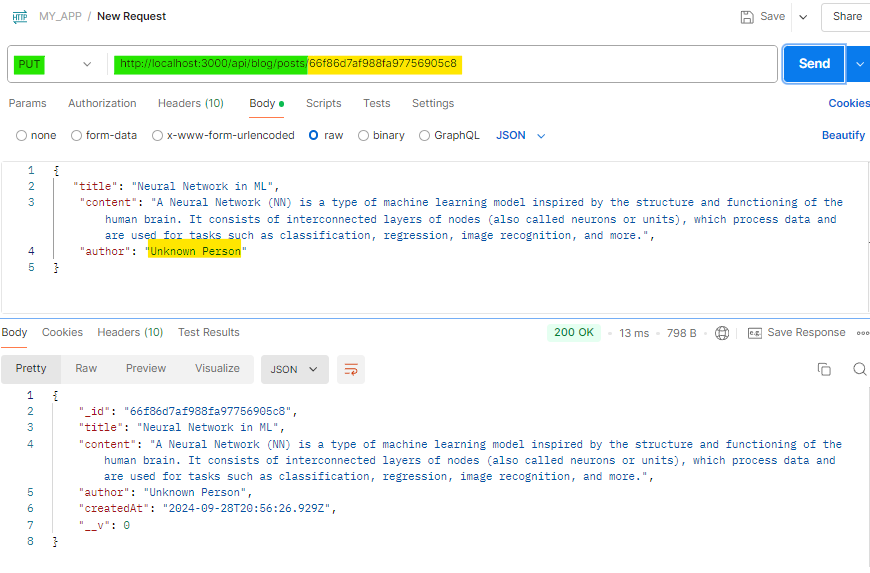
module.exports = Post;

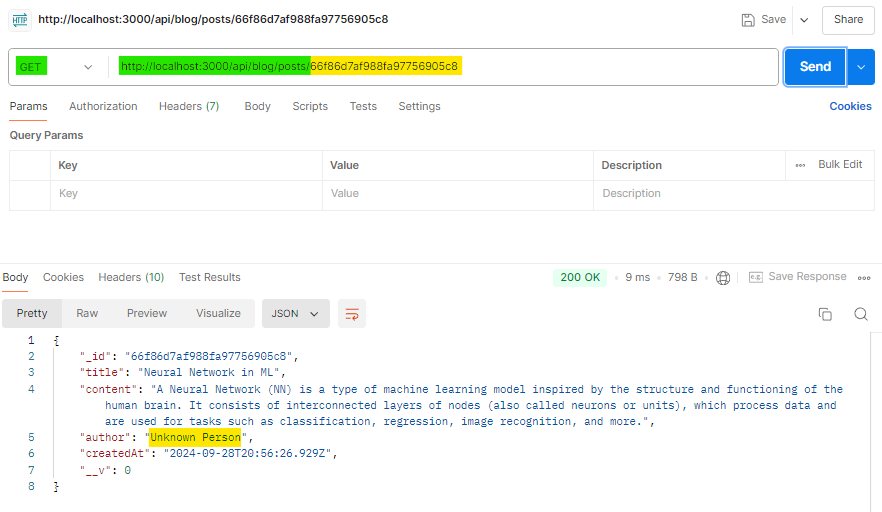
**Output:**



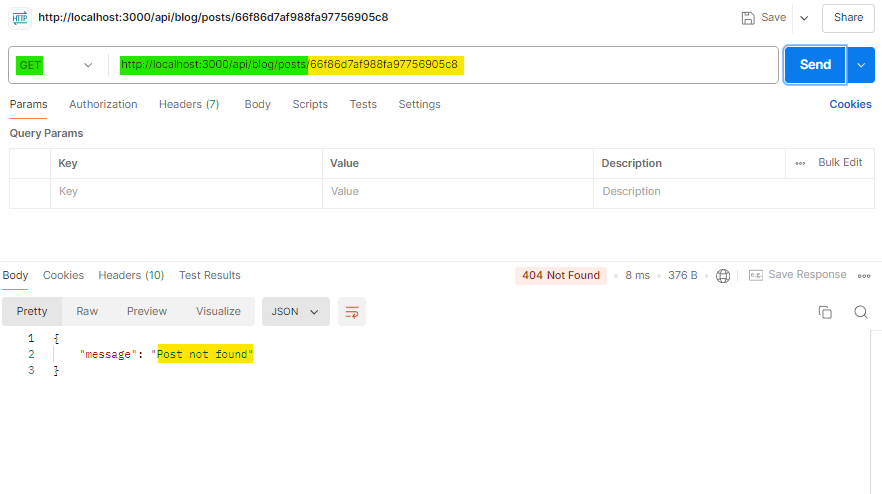












**Docker**

1. **Basic Dockerfile: Write a Dockerfile for a Node.js Express application that installs dependencies and runs the server on port 3000.**

**CODE:**

// 2.DOCKER\1.Basic\_Dockerfile\index.js

const express = require('express');

const app = express();

const PORT = 3000;

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

  res.send('Hello, World!');

});

app.listen(PORT, () => {

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

});

# 2.DOCKER\1.Basic\_Dockerfile\Dockerfile

# Use the official Node.js image as the base image

FROM node:18-alpine

# Set the working directory inside the container

WORKDIR /app

# Copy package.json and package-lock.json to the working directory

COPY package\*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application files to the working directory

COPY . .

# Expose port 3000

EXPOSE 3000

# Command to start the server

# CMD ["node", "index.js"]

CMD ["npm", "start"]

**EXPLANATION:**

1. Base Image: FROM node:18 - This pulls the official Node.js image (version 18 in this case) from Docker Hub.
2. Working Directory: WORKDIR /usr/src/app - Sets the working directory in the container to /usr/src/app. This is where the app code will reside.
3. Dependency Installation:

* COPY package\*.json ./ - Copies the package.json and package-lock.json to the working directory.
* RUN npm install - Runs npm install inside the container to install all the dependencies.

1. Copying App Files: COPY . . - Copies all the remaining application files to the container.
2. Expose Port: EXPOSE 3000 - Exposes port 3000, which is the default port where the Express server runs.
3. Start the Server: CMD ["npm", "start"] - Runs npm start, which starts the Express server. Ensure your package.json file has a start script defined, such as:

"scripts": {

    "start": "node index.js"

  },

**Command for Terminal:**

1. **Build the Docker image:**

docker build -t viveksingh/my-node-app:0.0.1 .

1. **Run the container:**

docker run -p 3000:3000 viveksingh/my-node-app:0.0.1

This will run your Node.js Express app inside a Docker container, and it will be accessible on <http://localhost:3000.>

1. **Docker Compose: Using Docker Compose, create a configuration file that sets up a multi-container application with a Node.js server and a MongoDB database.**

**CODE:**

# 2.DOCKER\2.Docker\_Compose\Dockerfile

FROM node:18

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]

# 2.DOCKER\2.Docker\_Compose\docker-compose.yml

version: '3'

services:

  # Node.js Application

  app:

    build: .

    container\_name: node\_app

    restart: always

    ports:

      - "3000:3000"

    depends\_on:

      - mongo

    environment:

      MONGO\_URL: mongodb://127.0.0.1:27017/myDB

    volumes:

      - .:/usr/src/app

  # MongoDB Database

  mongo:

    image: mongo:latest

    container\_name: mongo\_db

    restart: always

    ports:

      - "27017:27017"

    volumes:

      - mongo-data:/data/db

# Define named volume for persistent MongoDB data

volumes:

  mongo-data:

**EXPLANATION:**

1. **Version: '3' specifies the version of Docker Compose.**
2. **Services:**

**app: The Node.js application.**

* build: The Dockerfile for the Node.js app is in the current directory (.).
* container\_name: Names the container node\_app.
* restart: Ensures the container restarts automatically in case of failure.
* ports: Maps port 3000 of the container to port 3000 of the host.
* depends\_on: Ensures that the MongoDB service starts before the Node.js app.
* environment: Sets the environment variable MONGO\_URL for the app, pointing to the MongoDB container (mongo).
* volumes: Mounts the current directory into /usr/src/app in the container for code sharing.

**mongo: The MongoDB service.**

* image: Pulls the latest MongoDB image from Docker Hub.
* container\_name: Names the MongoDB container mongo\_db.
* ports: Maps MongoDB’s default port (27017) to the host machine.
* volumes: Persists MongoDB data to a named volume (mongo-data) for durability.

1. **Volumes:**

**mongo-data:** A named volume used to store MongoDB data persistently, ensuring that data is not lost when the container restarts.

**Command for Terminal:**

**Run the multi-container application:**

docker-compose up --build

This command will start both the Node.js application and MongoDB, linking them together. The Node.js app can access the MongoDB instance using the MONGO\_URL environment variable.

1. **Docker Networking: Modify the Docker Compose configuration to ensure that the Node.js application can communicate with the MongoDB database via Docker networking.**

**CODE:**

# 2.DOCKER\3.Docker\_Networking\docker-compose.yml

version: '3'

services:

  # Node.js Application

  app:

    build: .

    container\_name: node\_app

    restart: always

    ports:

      - "3000:3000"

    depends\_on:

      - mongo

    environment:

      MONGO\_URL: mongodb://127.0.0.1:27017/myDB  # Mongo service as hostname

    volumes:

      - .:/usr/src/app

    networks:

      - app-network

  # MongoDB Database

  mongo:

    image: mongo:latest

    container\_name: mongo\_db

    restart: always

    ports:

      - "27017:27017"

    volumes:

      - mongo-data:/data/db

    networks:

      - app-network

# Define named volume for MongoDB data persistence

volumes:

  mongo-data:

# Create a custom network for both services

networks:

  app-network:

    driver: bridge

**EXPLANATION:**

1. **Networking:**

* Both the app and mongo services are explicitly placed on a custom network called app-network. This ensures the Node.js application and MongoDB can communicate with each other using Docker's internal DNS, allowing the app to resolve mongo as the MongoDB hostname.
* Driver: The bridge driver is used, which is the default network driver for Docker containers. This allows containers to communicate over a private network.

1. **MONGO\_URL Environment Variable:**

The environment variable MONGO\_URL is set to mongodb://127.0.0.1:27017/myDB. Here, mongo is the service name for the MongoDB container, acting as the hostname that the Node.js application uses to connect to MongoDB.

**Command for Terminal:**

1. **Ensure Docker Compose is running:**

docker-compose up --build

1. **Application Access:**

The Node.js app can use the MONGO\_URL in its configuration to access MongoDB, and mongo will be resolved to the correct internal IP by Docker’s DNS.

With this configuration, the Node.js application can securely and reliably communicate with the MongoDB database using Docker's internal networking features.

1. **Containerization: Explain the advantages of using Docker for deploying a MERN Stack application and provide an example of a real-world use case where Docker enhances development workflows.**

**Advantages of Using Docker for Deploying a MERN Stack Application:**

* **Consistency Across Environments:** Docker ensures that the entire MERN (MongoDB, Express, React, Node.js) application runs the same way across development, staging, and production environments. This eliminates the "it works on my machine" problem since the app is containerized with all dependencies included.
* **Simplified Dependency Management:** A Docker container contains everything needed to run an application (OS, libraries, environment variables, etc.), so there’s no need to worry about conflicts between different versions of Node.js, MongoDB, or other dependencies in the stack.
* **Scalability and Resource Isolation:** Docker enables you to easily scale individual services (MongoDB, Express, Node.js, React) as separate containers. You can allocate specific resources (CPU, memory) to each service, ensuring better resource isolation and efficient scaling.
* **Efficient CI/CD Integration**: Docker can easily integrate with continuous integration and deployment pipelines (CI/CD). With Docker, you can build, test, and deploy containers automatically, speeding up deployment cycles and ensuring consistent builds.
* **Portability:** Since Docker containers encapsulate everything, you can run them on any platform that supports Docker, be it local machines, cloud servers (AWS, Google Cloud, Azure), or hybrid environments, without worrying about compatibility.
* **Fast and Lightweight:** Containers are more lightweight compared to traditional virtual machines (VMs) because they share the host machine's operating system kernel. This makes spinning up containers faster, reducing boot times and improving developer productivity.
* **Microservices Architecture:** Docker makes it easier to break the MERN stack into separate microservices. Each component (MongoDB, Express, Node.js, React) can be run in its own container, allowing you to scale and manage them independently.

**Real-World Use Case: Docker Enhancing Development Workflow**

**Development Environment:** Each team (frontend, backend, database) can work in isolated Docker containers, using different versions of Node.js or MongoDB as needed. Docker Compose can be used to spin up all services together, so developers can work on different parts of the system while still being able to interact with the entire stack in their local environment.

* The front-end team can run React in one container, ensuring their setup is isolated.
* The back-end team can have their Node.js + Express environment in another container.
* MongoDB can run in its own container.

**CI/CD Integration:** In a CI/CD pipeline, Docker images are built for each service. Every time a new feature is developed or a bug is fixed, the relevant service is containerized and tested in isolation. Automated tests can run against these containers, ensuring that updates don’t introduce bugs.

* If the front-end team pushes changes to React, Docker builds a new container for the front-end and runs it alongside the existing services for automated testing.
* Similarly, if the back-end team pushes an update to the Express.js API, the back-end container is rebuilt and tested without affecting other services.

**Deployment and Scaling:** When deploying to production, Docker containers make scaling much easier. If the back-end API is handling heavy traffic, the Node.js container can be scaled up independently without affecting other services. Kubernetes (or Docker Swarm) can be used to orchestrate and manage scaling automatically.

* For example, the Node.js container handling the Express API could be scaled up to handle more requests during peak hours, while MongoDB containers may need less scaling.

**Version Control for DevOps:** Docker images provide version control for environments. If something breaks in production, developers can roll back to a previous Docker image quickly and reliably. This can prevent downtime and make disaster recovery more manageable.

**CODE:**

# 2.DOCKER\4.Containerization\docker-compose.yml

version: '3'

services:

  # MongoDB service

  mongo:

    image: mongo:latest

    container\_name: mongo\_db

    restart: always

    ports:

      - "27017:27017"

    volumes:

      - mongo-data:/data/db

  # Express.js and Node.js service

  backend:

    build: ./backend

    container\_name: node\_backend

    restart: always

    ports:

      - "5000:5000"

    depends\_on:

      - mongo

    environment:

      MONGO\_URL: mongodb://127.0.0.1:27017/myDB

  # React front-end service

  frontend:

    build: ./frontend

    container\_name: react\_frontend

    restart: always

    ports:

      - "3000:3000"

volumes:

  mongo-data:

**EXPLANATION:**

1. MongoDB, Node.js (Express), and React all run in separate containers, with MongoDB data stored in a persistent volume.
2. This setup works in both development and production, ensuring consistency across environments.

**CONCLUSION:**

Docker helps in making the development, testing, and deployment of MERN stack applications more efficient by providing isolated environments, simplifying dependency management, enabling CI/CD pipelines, and scaling services independently. This ensures a smooth workflow for both small teams and large-scale applications.

**GitHub/Bitbucket and Version Control**

1. **Basic Git Commands: Explain the steps and Git commands to initialize a repository, make a commit, and push the code to GitHub.**

1. **Initialize a Repository**

git init

2. **Check the Current Status**

git status

3. **Add Files to Staging Area**

git add <filename>

git add . //To add all files at once

4. **Make a Commit**

git commit -m "Your commit message"

5. **Connect to a GitHub Repository**

git remote add origin <repository-url>

6. **Push the Code to GitHub**

git push -u origin main

**Summary of Git Commands:**

**git init** – Initialize a repository.

**git status** – Check the status of your files.

**git add .** – Add all files to the staging area.

**git commit -m "Commit message"** – Commit the staged changes with a message.

**git remote add origin <repository-url>** – Connect your local repository to GitHub.

**git push -u origin main** – Push your code to GitHub.

1. **Branching Strategy: Describe a common branching strategy (such as GitFlow or feature branching) used in software development teams and how you would implement it for a new feature.**
2. **Feature Branching Strategy Overview**

* **Main Branch (Production/Stable):** This branch (commonly called main or master) holds the stable, production-ready code. No direct development happens on this branch except for important hotfixes.
* **Develop Branch (Integration/Testing):** This branch is often used as a middle-ground to integrate features or bug fixes before they are merged into the main branch. The develop branch serves as the base for feature branches.
* **Feature Branches:** Each new feature or improvement is developed in its own branch, typically branched off from develop. The naming convention usually follows feature/feature-name.
* **Pull Request and Code Review:** Once a feature is completed and tested, the feature branch is merged back into develop (or another integration branch) via a pull request. This is when the code undergoes peer reviews and automated testing to ensure it doesn't break the existing system.
* **Release Branch:** Once the develop branch has several features ready for release, a release branch may be created. This branch is tested rigorously before merging into main for production deployment.
* **Hotfix Branches:** If an urgent bug is found in production, a hotfix branch is created from the main branch, fixed, and merged back into both main and develop to ensure the fix is available in future versions.

1. **Implementing Feature Branching for a New Feature**

* **Start from develop branch**

git checkout develop

git pull origin develop

* **Create a Feature Branch**

git checkout -b feature/new-feature-name

* **Work on the Feature**

git add .

git commit -m "Implement part 1 of new feature"

* **Push the Feature Branch to Remote**

git push origin feature/new-feature-name

* **Create a Pull Request**
* **Code Review and Testing**
* **Merge into develop**

git checkout develop

git merge feature/new-feature-name

* **Delete the Feature Branch**

git branch -d feature/new-feature-name

git push origin --delete feature/new-feature-name

1. **Summary of Commands**

**git checkout develop** # Switch to the develop branch

**git pull origin develop** # Pull the latest changes from develop

**git checkout -b feature/new-feature** # Create and switch to a new feature branch

**git add .**  # Stage your changes

**git commit -m "Add new feature"** # Commit your changes

**git push origin feature/new-feature** # Push your feature branch to remote

# Open a pull request and merge after review

**git checkout develop** # Switch back to develop branch

**git merge feature/new-feature** # Merge the feature branch into develop

**git branch -d feature/new-feature**  # Delete the local feature branch

**git push origin --delete feature/new-feature**  # Delete the remote feature branch

1. **Merging and Resolving Conflicts: Write a step-by-step guide to resolve a merge conflict when merging a feature branch into the main branch.**

**Step 1: Ensure Your Branches Are Up to Date**

1. **Switch to the main branch:**

git checkout main

1. **Pull the latest changes from the remote:**

git pull origin main

1. **Switch to your feature branch:**

git checkout feature/new-feature

1. **Pull the latest changes from the remote feature branch (if applicable):**

git pull origin feature/new-feature

**Step 2: Merge the main Branch into the Feature Branch**

**Merge main into your feature branch:**

git merge main

**Step 3: Identify Merge Conflicts**

**After initiating the merge, Git will flag files that have conflicts. The terminal output will look something like this:**

Auto-merging <file-path>

CONFLICT (content): Merge conflict in <file-path>

Automatic merge failed; fix conflicts and then commit the result.

**Conflicting files will be marked with the status U (unmerged):**

git status

**Example output:**

both modified: src/components/Header.js

both modified: src/App.js

**Step 4: Open and Resolve Conflicts Manually**

**Open the conflicting files in your code editor. You'll see conflict markers like this:**

<<<<<<< HEAD

// Code from the `main` branch

=======

// Code from the `feature/new-feature` branch

>>>>>>> feature/new-feature

**Step 5: Edit the Code to Resolve Conflicts**

* **Accept the changes from the main branch:** Delete the feature branch section, keeping the main branch section.
* **Accept the changes from the feature branch:** Delete the main branch section, keeping the feature branch section.
* **Combine both changes:** Modify the code to incorporate both changes appropriately.

**Step 6: Mark Conflicts as Resolved**

**Add the resolved file(s):**

git add <file-path> or git add .

**Step 7: Complete the Merge**

**Commit the resolved merge:**

git commit or git commit -m "Resolved merge conflict between main and feature/new-feature"

**Step 8: Merge the Feature Branch into the main Branch**

1. **Switch back to the main branch:**

git checkout main

1. **Merge the feature branch:**

git merge feature/new-feature

**Step 9: Push Changes to Remote**

**Push to remote:**

git push origin main

**Step 10: Clean Up**

1. **Delete the local feature branch:**

git branch -d feature/new-feature

1. **Delete the remote feature branch:**

git push origin --delete feature/new-feature

**Summary of Commands**

**# Step 1:** **Checkout and pull the main branch**

git checkout main

git pull origin main

**# Step 2:** **Checkout and merge main into feature branch**

git checkout feature/new-feature

git merge main

**# Step 3:** **Resolve conflicts manually in files**

**# Step 4:** **Mark conflicts as resolved**

git add <file-path>

**# Step 5:** **Commit the resolved merge**

git commit

**# Step 6:** **Merge the feature branch into main**

git checkout main

git merge feature/new-feature

**# Step 7:** **Push the merged main branch to the remote**

git push origin main

**# Step 8:** **Clean up**

git branch -d feature/new-feature

git push origin --delete feature/new-feature

1. **CI/CD Integration: Explain how to set up a basic CI/CD pipeline using GitHub Actions to automatically test and deploy a Node.js application when changes are pushed to the repository.**
2. **Create a GitHub Repository for the Node.js Application**

* **Initialize a Git repository in your project directory by running:**

git init

* **Add the files, make a commit, and push it to GitHub:**

git add .

git commit -m "Initial commit"

git remote add origin <your-github-repository-url>

git push -u origin main

1. **Set Up GitHub Actions Workflow File**

**Steps to Create the Workflow:**

* Create a .github/workflows directory in the root of your project.
* Inside that directory, create a YAML file (e.g., ci-cd.yml) that defines the CI/CD steps.

1. **Create a Basic CI/CD Workflow for Node.js**

# .github/workflows/ci-cd.yml

name: Node.js CI/CD Pipeline

# Triggers the workflow on push or pull requests to the main branch

on:

  push:

    branches:

      - main

  pull\_request:

    branches:

      - main

jobs:

  # Define the first job: Build and Test

  build-and-test:

    runs-on: ubuntu-latest  # Use the latest version of Ubuntu

    steps:

    # Checkout the code from the repository

    - name: Checkout code

      uses: actions/checkout@v3

    # Set up Node.js environment with the version you specify

    - name: Setup Node.js

      uses: actions/setup-node@v3

      with:

        node-version: '18.x'  # Specify Node.js version

    # Install application dependencies

    - name: Install dependencies

      run: npm install

    # Run tests to ensure code integrity

    - name: Run tests

      run: npm test

  # Define the second job: Deployment (only if build and tests pass)

  deploy:

    runs-on: ubuntu-latest  # Use Ubuntu for deployment as well

    needs: build-and-test   # Run this job only after build-and-test passes

    steps:

    # Checkout code again for deployment

    - name: Checkout code

      uses: actions/checkout@v3

    # Set up Node.js environment for deployment

    - name: Setup Node.js

      uses: actions/setup-node@v3

      with:

        node-version: '18.x'

    # Install dependencies again for the deployment step

    - name: Install dependencies

      run: npm install

    # Deploy the application

    - name: Deploy to Production

      run: |

        # Example deployment script (customize based on your deployment provider)

        npm run deploy

1. **Explanation of Each Part of the Workflow**

**Triggers (on):**

The workflow is triggered when code is pushed to or a pull request is opened for the main branch. You can customize this to any branch or event, like tags or releases.

**Jobs:**

**Build and Test Job:**

* runs-on: Specifies that the job runs on an Ubuntu machine.
* Checkout code: Uses the actions/checkout@v3 action to fetch the code from the GitHub repository.
* Setup Node.js: Uses actions/setup-node@v3 to set up the Node.js environment, specifying the version (18.x).
* Install dependencies: Runs npm install to install dependencies required by the project.
* Run tests: Executes npm test to run any defined test suites.

**Deploy Job:**

* Depends on the success of the build-and-test job. It won’t run unless the build and tests pass.
* Similar to the build job, it checks out the code and sets up the Node.js environment.
* Executes deployment using a custom command (npm run deploy). You would adjust this step to fit your deployment environment (Heroku, AWS, etc.).

1. **Deployment Setup**

**In the deployment section, you would replace the npm run deploy command with actual deployment steps depending on the platform. For example:**

* Heroku Deployment: You can use the Heroku CLI to deploy by authenticating with Heroku API keys.
* AWS Deployment: You could set up AWS CLI commands to deploy to AWS services like Elastic Beanstalk or EC2.
* Docker Deployment: You could use Docker to build and push images to a container registry.

1. **Add Secrets for Secure Deployment**

**To securely deploy your application, you’ll likely need API keys or authentication tokens. GitHub provides a way to store secrets:**

* Go to Settings > Secrets > Actions in your GitHub repository.
* Add environment-specific secrets, such as:
* HEROKU\_API\_KEY
* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY
* These secrets can then be referenced in your GitHub Actions workflow to authenticate with external services.

1. **Test the Workflow**

**Once you push changes to the main branch, the GitHub Actions workflow will be triggered:**

* Navigate to the Actions tab in your repository to monitor workflow progress.
* You’ll be able to view logs for each step, which will help in debugging if any issues occur.

1. **Extend the Workflow**

**You can extend the basic workflow to include more steps like:**

* Code Linting: Automatically lint your code using ESLint.
* Building Docker Images: If your app uses Docker, you can add steps to build and push Docker images.
* Deploying to Multiple Environments: You can create different jobs for staging and production deployments, triggered by different branches or tags.

1. **Scaling the Workflow for Future Growth**

**As your application grows, you can:**

* Add more tests: Include unit, integration, and end-to-end tests to ensure code quality at every step.
* Parallelize jobs: If the workflow becomes slow, consider running jobs in parallel, such as building in one job and testing in another.
* Monitor performance: Integrate performance monitoring tools like New Relic or Datadog for continuous monitoring.

**Python for Data Analysis**

1. **Data Cleaning: Write a Python script that reads a CSV file using Pandas, drops rows with missing values, and outputs the cleaned data.**

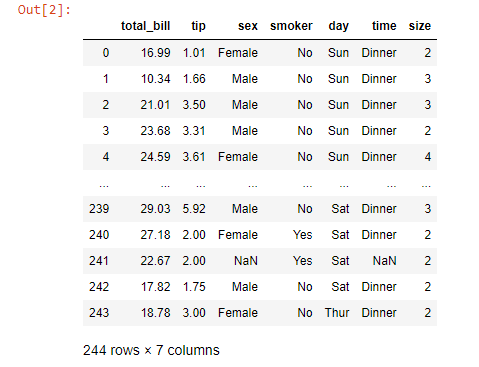
**#Import libraries**

**import pandas as pd**

**# Read the CSV file**

**data = pd.read\_csv('tips\_uncleaned.csv')**

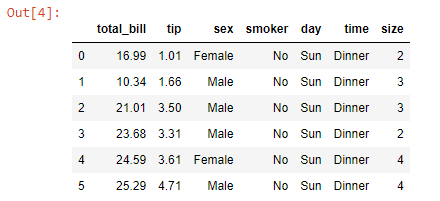
**data**



**data.shape**



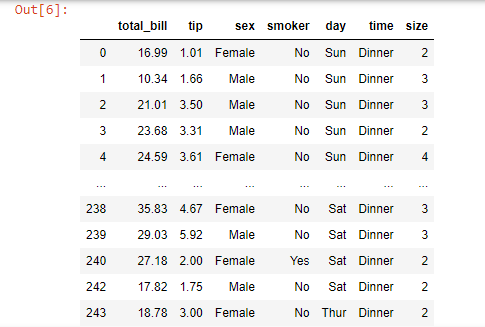
**data.head(6)**



**# Drop rows with missing values**

**cleaned\_data = data.dropna()**

**cleaned\_data**



**cleaned\_data.shape**



**# Save the cleaned data to a new CSV file**

**cleaned\_data.to\_csv('tips\_cleaned.csv', index=False)**

1. **Data Manipulation: Using Pandas, write a Python function that takes a DataFrame and returns the top 5 rows where a specific column (e.g., "age") has values greater than 30.**

**#Import libraries**

**import pandas as pd**

**# Sample data for demonstration**

**data = {'name': ['John', 'Alice', 'Bob', 'Clara', 'David', 'Eva'],**

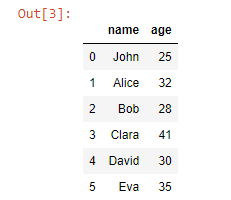
**'age': [25, 32, 28, 41, 30, 35]}**

**# Create a DataFrame**

**df = pd.DataFrame(data)**

**# Display DataFrame**

**Df**



**# Function to filter and return top 5 rows where a specific column has values greater than a threshold**

**def filter\_top\_rows(df, column\_name, threshold):**

**# Filter rows where the values in the specified column are greater than the threshold**

**filtered\_df = df[df[column\_name] > threshold]**

**# Return the top 5 rows**

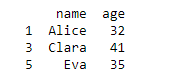
**return filtered\_df.head(5)**

**# Filter top 5 rows where 'age' > 30**

**result = filter\_top\_rows(df, 'age', 30)**

**# Display the result**

**print(result)**



1. **Data Visualization: Create a bar chart using Matplotlib to visualize the distribution of user ages from a dataset.**

**#Import libraries**

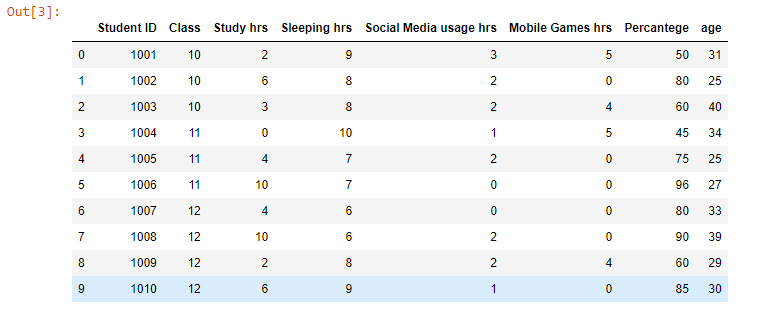
**import pandas as pd**

**import matplotlib.pyplot as plt**

**# Load dataset from CSV file**

**df = pd.read\_csv('student\_results.csv')**

**df**



**# Get the frequency distribution of ages**

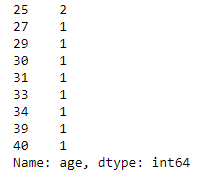
**age\_distribution = df['age'].value\_counts()**

**# Sort the distribution by age values for a better visual representation**

**age\_distribution = age\_distribution.sort\_index()**

**# Display the age distribution (for understanding)**

**print(age\_distribution)**



**# Plot a bar chart**

**plt.figure(figsize=(10, 6)) # Set the size of the figure**

**plt.bar(age\_distribution.index, age\_distribution.values, color='skyblue')**

**# Add titles and labels**

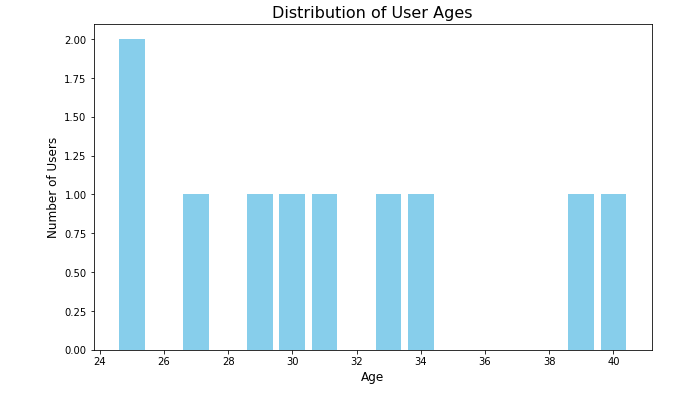
**plt.title('Distribution of User Ages', fontsize=16)**

**plt.xlabel('Age', fontsize=12)**

**plt.ylabel('Number of Users', fontsize=12)**

**# Display the bar chart**

**plt.show()**



**# Save the figure as a PNG file**

**plt.savefig('age\_distribution.png')**



1. **Descriptive Statistics: Using NumPy and Pandas, write a script that calculates the mean, median, and standard deviation of a column (e.g., "age") in a dataset.**

**#Import libraries**

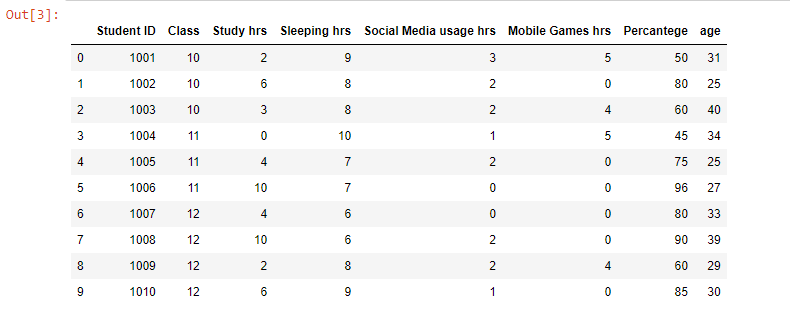
**import numpy as np**

**import pandas as pd**

**# Load dataset from CSV file**

**df = pd.read\_csv('student\_results.csv')**

**df**



**# Calculate mean, median, and standard deviation using Pandas**

**mean\_age = df['age'].mean() # Mean**

**median\_age = df['age'].median() # Median**

**std\_dev\_age = df['age'].std() # Standard Deviation**

**# Display the results**

**print(f"Mean age: {mean\_age}")**

**print(f"Median age: {median\_age}")**

**print(f"Standard Deviation of age: {std\_dev\_age}")**

