

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

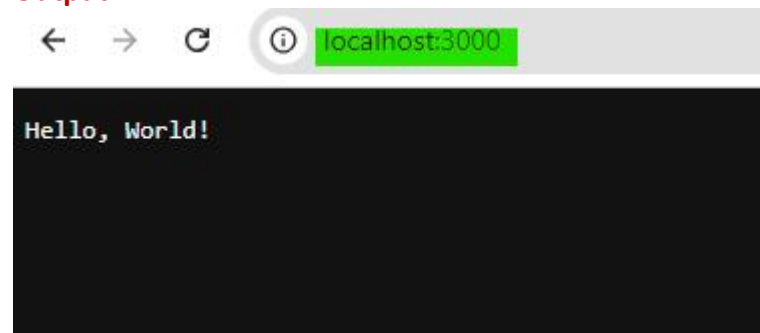
CODE:

```
//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:



2. 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");

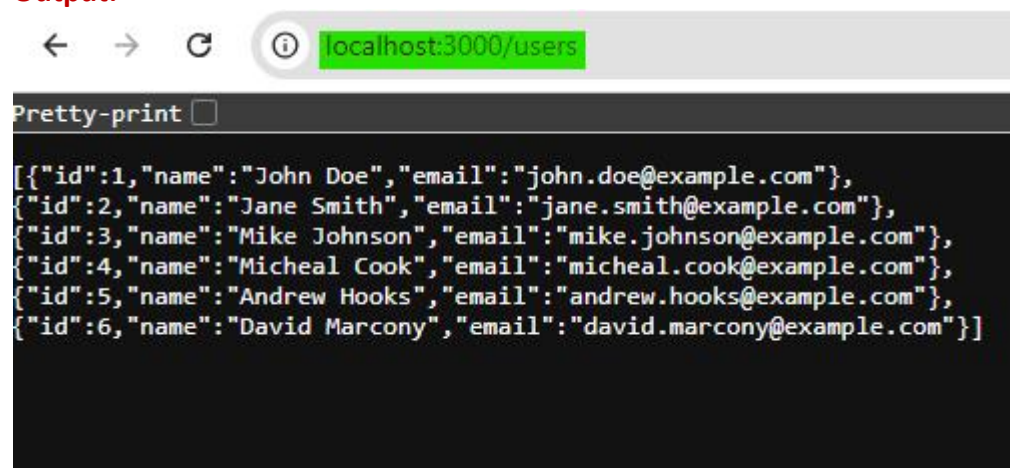
app.get('/users', (req, res) => {
  res.status(200).json(users);
});

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:



3. 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:

<div><div>←</div><div>→</div><div>↺</div><div><div>🔒</div>localhost:5173</div></div>		
<h1>User List</h1>		
ID	Name	Email
1	John Doe	john.doe@example.com
2	Jane Smith	jane.smith@example.com
3	Mike Johnson	mike.johnson@example.com
4	Micheal Cook	micheal.cook@example.com
5	Andrew Hooks	andrew.hooks@example.com
6	David Marcony	david.marcony@example.com

4. 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:

```

PS C:\Users\KIIT\Desktop\LEAMORE\MERN_DEVELOPER_ASSIGNMENT\1.MERN_STACK\4.MongoDB> node insertUser.js
● Connected to MongoDB
User inserted: {
  name: 'Vivek Singh',
  email: 'vivek.singh@example.com',
  age: 25,
  _id: new ObjectId('66f83621c606bb2b2a862996'),
  __v: 0
}
PS C:\Users\KIIT\Desktop\LEAMORE\MERN_DEVELOPER_ASSIGNMENT\1.MERN_STACK\4.MongoDB>

```

```

mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
test> show dbs
admin          40.00 KiB
amazon        72.00 KiB
college       40.00 KiB
config        72.00 KiB
local         76.00 KiB
mern_admin     8.00 KiB
relationDemo 288.00 KiB
test          80.00 KiB
usersDB       28.00 KiB
wanderlust    272.00 KiB
whatsapp      72.00 KiB
test> use usersDB
switched to db usersDB
usersDB> show collections
users
usersDB> db.users.find()
[
  {
    _id: ObjectId('66f83621c606bb2b2a862996'),
    name: 'Vivek Singh',
    email: 'vivek.singh@example.com',
    age: 25,
    __v: 0
  }
]
usersDB>

```

5. 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;

```

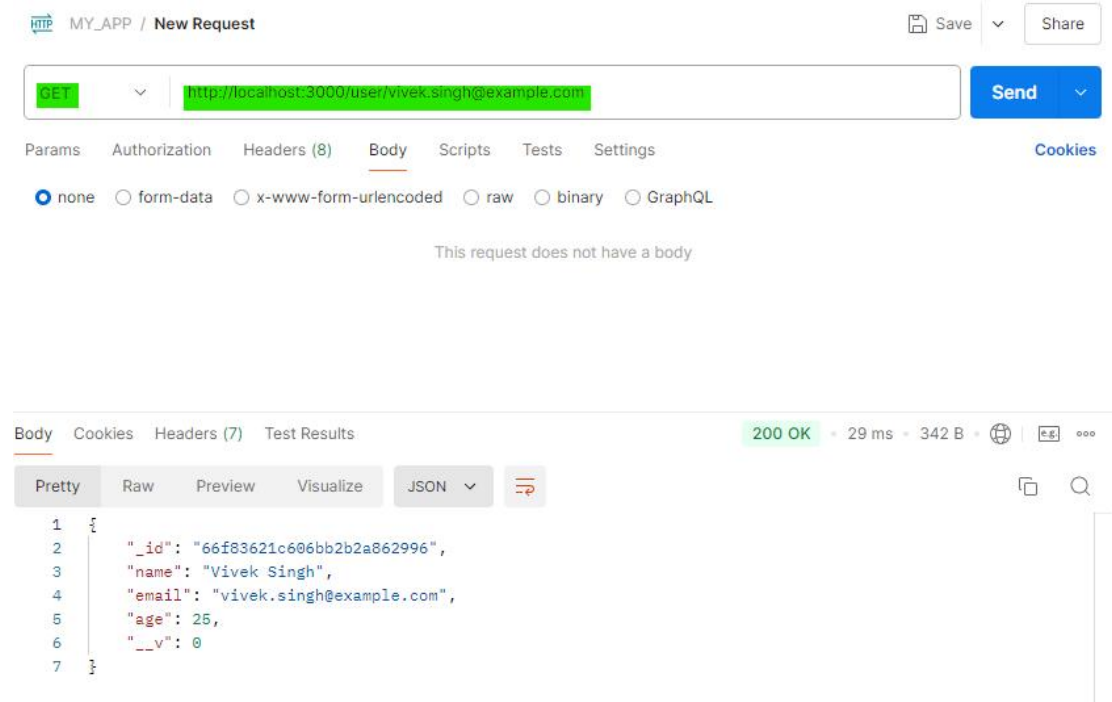
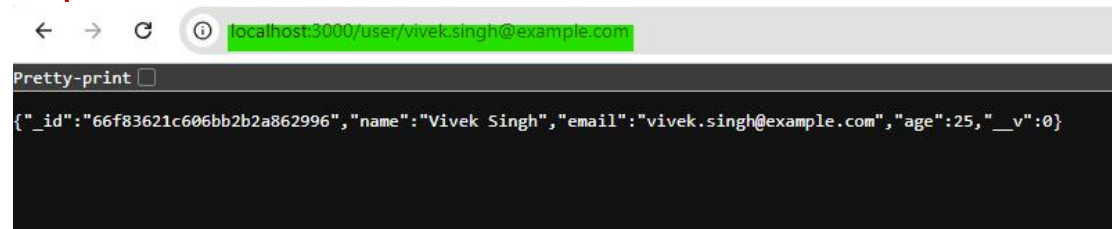
```

//
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:



6. **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>
);
}

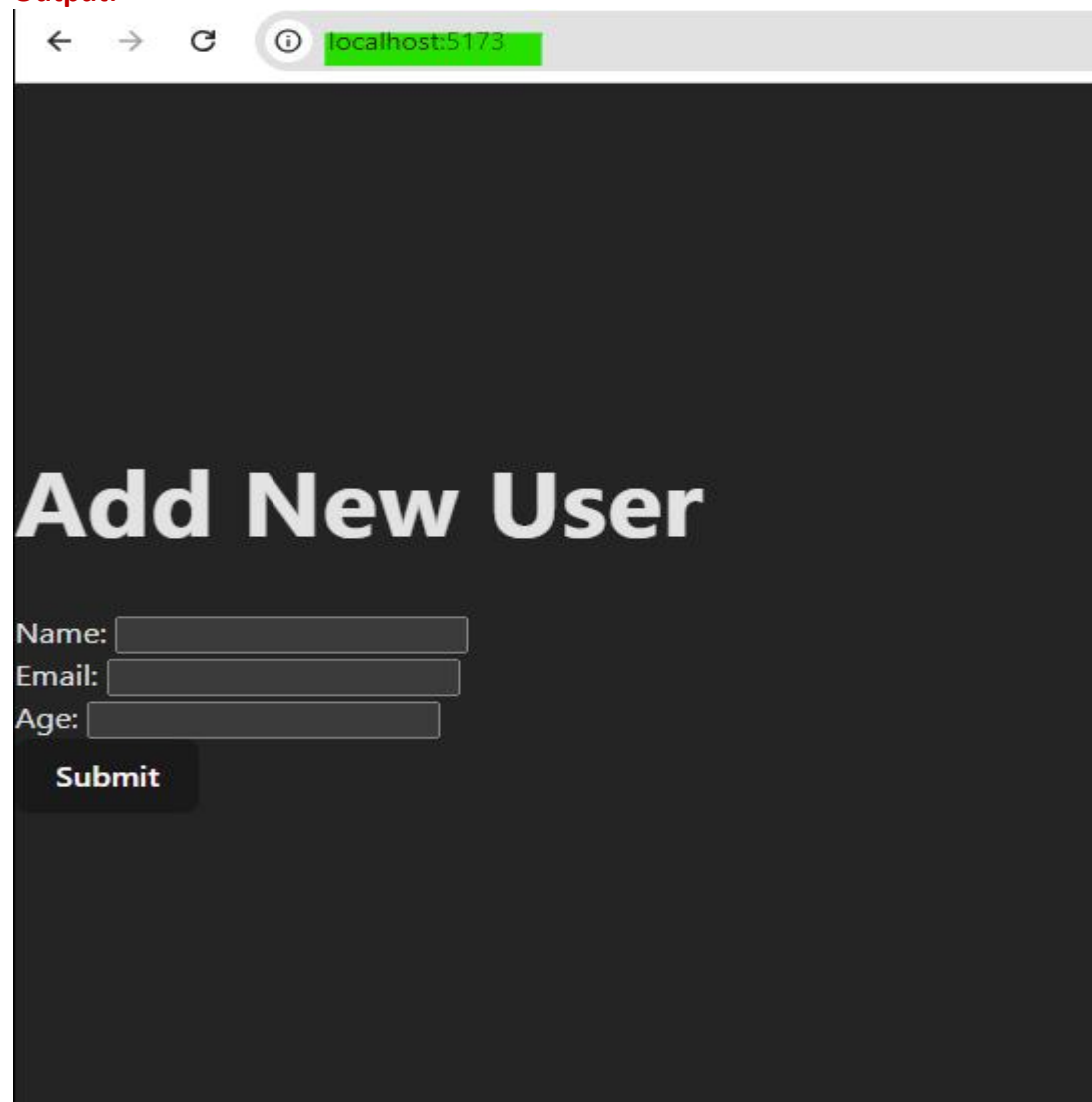
```

```
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:



localhost:5173

Add New User

Name:

Email:

Age:

Submit

←

→

↻

ⓘ

localhost:5173

Add New User

Name:

Email:

Age:

← → ↻ ⓘ localhost:5173

Add New User

Name:

Email:

Age:

Submit

New User Added:

Name: Devang Prasad

Email: devang.prasad@example.com

Age: 22

7. 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.j
sx
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>
    <h2>Users</h2>
    <ul>
      {users.map((user) => (
        <li>{user.name}</li>
      ))}
    </ul>
  </div>;
}

export default Users;

```

```

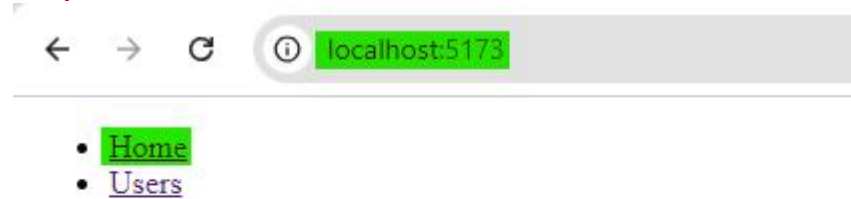
    }

    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:



Home Page

Welcome to the Home page!

- [Home](#)
- [Users](#)

Users Page

ID	Name	Email
	Vivek Singh	vivek.singh@example.com
	Rohit Sharma	rohit.sharma@example.com
	Vikram Rathore	vikram.rathore@example.com
	Manisha Yadav	manisha.yadav@example.com
	Vaibhav Sharma	vaibhav.sharma@example.com
	Swati Saini	swati.saini@example.com
	Sumit Yadav	sumit.yadav@example.com
	Manoj Desai	manoj.desai@example.com
	Sheetal Rai	sheetal.rao@example.com
	Ragini Rai	ragini.rao@example.com

8. 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:

MY_APP / New Request Save Share

POST ▼ http://localhost:3000/api/blog/posts Send ▼

Params Authorization Headers (10) **Body** ● Scripts Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON ▼ Beautify

```

1  {
2    "title": "Neural Network in ML",
3    "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the
        human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and
        are used for tasks such as classification, regression, image recognition, and more.",
4    "author": "Vivek Singh"
5  }

```

Body Cookies Headers (10) Test Results 201 Created · 15 ms · 800 B · 🌐 📄 Save Response ⋮

Pretty Raw Preview Visualize JSON ▼ 🔧

```

1  {
2    "title": "Neural Network in ML",
3    "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the
        human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and
        are used for tasks such as classification, regression, image recognition, and more.",
4    "author": "Vivek Singh",
5    "_id": "66f86d7af988fa97756905c8",
6    "createdAt": "2024-09-28T20:56:26.929Z",
7    "__v": 0
8  }

```

HTTP

http://localhost:3000/api/blog/posts

Save

Share

GET

http://localhost:3000/api/blog/posts

Send

ParamsAuthorizationHeaders (7)BodyScriptsTestsSettingsCookie

BodyCookiesHeaders (10)Test Results200 OK · 10 ms · 1.79 KB · Save Response

PrettyRawPreviewVisualizeJSON

```
16      "_v": 0
17    },
18  ],
19  "_id": "66f868bffa97756905be",
20  "title": "Machine Learning",
21  "content": "Machine learning is a subset of artificial intelligence that enables a system to autonomously learn and
                improve using neural networks and deep learning, without being explicitly programmed, by feeding it large
                amounts of data.",
22  "author": "Vimal Shrivastava",
23  "createdAt": "2024-09-28T20:36:15.191Z",
24  "_v": 0
25  },
26  ],
27  "_id": "66f86d7af988fa97756905c8",
28  "title": "Neural Network in ML",
29  "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of
                the human brain. It consists of interconnected layers of nodes (also called neurons or units), which process
                data and are used for tasks such as classification, regression, image recognition, and more.",
30  "author": "Vivek Singh",
31  "createdAt": "2024-09-28T20:56:26.929Z",
32  "_v": 0
33  }
34  ]
```

HTTP

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Save

Share

GET

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Send

ParamsAuthorizationHeaders (7)BodyScriptsTestsSettingsCookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

BodyCookiesHeaders (10)Test Results200 OK · 12 ms · 795 B · Save Response

PrettyRawPreviewVisualizeJSON

```
1  {
2    "_id": "66f86d7af988fa97756905c8",
3    "title": "Neural Network in ML",
4    "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the
                    human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and
                    are used for tasks such as classification, regression, image recognition, and more.",
5    "author": "Vivek Singh",
6    "createdAt": "2024-09-28T20:56:26.929Z",
7    "_v": 0
8  }
```


MY_APP / New Request

SaveShare

PUT

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Send

ParamsAuthorizationHeaders (10)BodyScriptsTestsSettingsCookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautify

```
1 {
2   "title": "Neural Network in ML",
3   "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the
              human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and
              are used for tasks such as classification, regression, image recognition, and more.",
4   "author": "Unknown Person"
5 }
```

BodyCookiesHeaders (10)Test Results200 OK · 13 ms · 798 B · Save Response

PrettyRawPreviewVisualizeJSON

```
1 {
2   "_id": "66f86d7af988fa97756905c8",
3   "title": "Neural Network in ML",
4   "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the
              human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and
              are used for tasks such as classification, regression, image recognition, and more.",
5   "author": "Unknown Person",
6   "createdAt": "2024-09-28T20:56:26.929Z",
7   "__v": 0
8 }
```

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

SaveShare

GET

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Send

ParamsAuthorizationHeaders (7)BodyScriptsTestsSettingsCookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

BodyCookiesHeaders (10)Test Results200 OK · 9 ms · 798 B · Save Response

PrettyRawPreviewVisualizeJSON

```
1 {
2   "_id": "66f86d7af988fa97756905c8",
3   "title": "Neural Network in ML",
4   "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the
              human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and
              are used for tasks such as classification, regression, image recognition, and more.",
5   "author": "Unknown Person",
6   "createdAt": "2024-09-28T20:56:26.929Z",
7   "__v": 0
8 }
```

MY_APP / New Request

Save

Share

DELETE

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Send

Params

Authorization

Headers (10)

Body

Scripts

Tests

Settings

Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautify

```
1 {
2   "title": "Neural Network in ML",
3   "content": "A Neural Network (NN) is a type of machine learning model inspired by the structure and functioning of the human brain. It consists of interconnected layers of nodes (also called neurons or units), which process data and are used for tasks such as classification, regression, image recognition, and more.",
4   "author": "Unknown Person"
5 }
```

Body

Cookies

Headers (10)

Test Results

200 OK

11 ms

380 B

Save Response

...

Pretty

Raw

Preview

Visualize

JSON

...

```
1 {
2   "message": "Post deleted successfully"
3 }
```

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Save

Share

GET

http://localhost:3000/api/blog/posts/66f86d7af988fa97756905c8

Send

Params

Authorization

Headers (7)

Body

Scripts

Tests

Settings

Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (10)

Test Results

404 Not Found

8 ms

376 B

Save Response

...

Pretty

Raw

Preview

Visualize

JSON

...

```
1 {
2   "message": "Post not found"
3 }
```

Docker

9. 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.
- 4) Copying App Files: COPY . . - Copies all the remaining application files to the container.
 - 5) Expose Port: EXPOSE 3000 - Exposes port 3000, which is the default port where the Express server runs.
 - 6) 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:

I. Build the Docker image:

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

II. 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>.

10. 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.

3) 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.

11. 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.

2) **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:

I. Ensure Docker Compose is running:

`docker-compose up --build`

II. 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.

12. 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

13. 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.

14. 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.

1) 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.

2) 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

3) 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

15. 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
```

2) Pull the latest changes from the remote:

```
git pull origin main
```

3) Switch to your feature branch:

```
git checkout feature/new-feature
```

4) 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

2) 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

2) 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

16. 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.

1) 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
```

2) 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.

3) 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

```

4) 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.).

5) 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.

6) 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.

7) 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.

8) 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.

9) 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

17. 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

Out[2]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	NaN	Yes	Sat	NaN	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

data.shape

Out[3]: (244, 7)

data.head(6)

Out[4]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
5	25.29	4.71	Male	No	Sun	Dinner	4

Drop rows with missing values

```
cleaned_data = data.dropna()
```

```
cleaned_data
```

```
Out[6]:
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...
238	35.83	4.67	Female	No	Sat	Dinner	3
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

```
cleaned_data.shape
```

```
Out[7]: (238, 7)
```

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

```
cleaned_data.to_csv('tips_cleaned.csv', index=False)
```

18. 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

`Out[3]:`

	name	age
0	John	25
1	Alice	32
2	Bob	28
3	Clara	41
4	David	30
5	Eva	35

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)
```

	name	age
1	Alice	32
3	Clara	41
5	Eva	35

19. 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

Out[3]:

	Student ID	Class	Study hrs	Sleeping hrs	Social Media usage hrs	Mobile Games hrs	Percantege	age
0	1001	10	2	9	3	5	50	31
1	1002	10	6	8	2	0	80	25
2	1003	10	3	8	2	4	60	40
3	1004	11	0	10	1	5	45	34
4	1005	11	4	7	2	0	75	25
5	1006	11	10	7	0	0	96	27
6	1007	12	4	6	0	0	80	33
7	1008	12	10	6	2	0	90	39
8	1009	12	2	8	2	4	60	29
9	1010	12	6	9	1	0	85	30

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)
```

```
25    2
27    1
29    1
30    1
31    1
33    1
34    1
39    1
40    1
Name: age, dtype: int64
```

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')
```

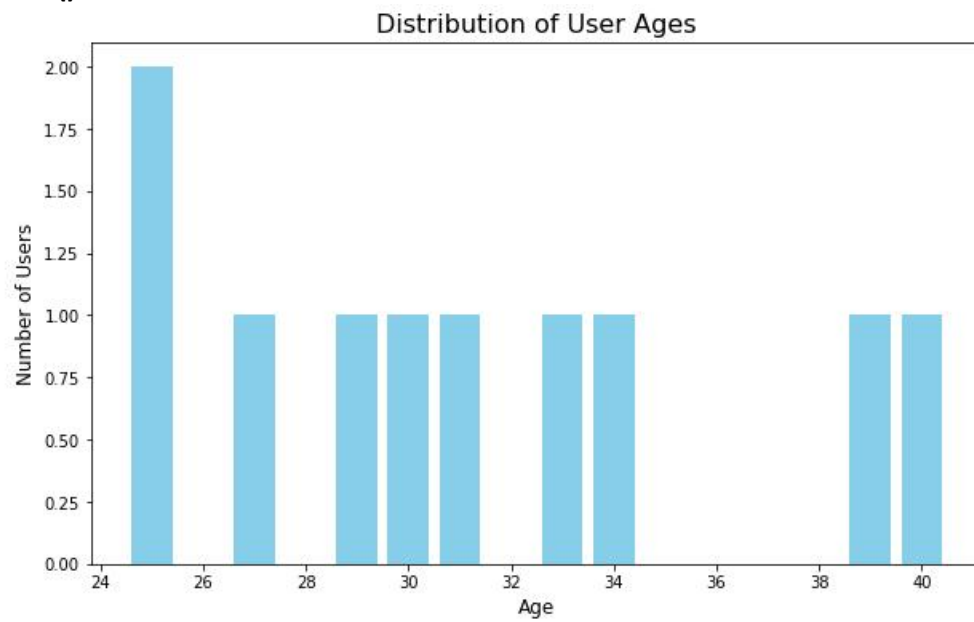
Add titles and labels

```
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')
```

```
<Figure size 432x288 with 0 Axes>
```


20. 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

Out[3]:

	Student ID	Class	Study hrs	Sleeping hrs	Social Media usage hrs	Mobile Games hrs	Percantege	age
0	1001	10	2	9	3	5	50	31
1	1002	10	6	8	2	0	80	25
2	1003	10	3	8	2	4	60	40
3	1004	11	0	10	1	5	45	34
4	1005	11	4	7	2	0	75	25
5	1006	11	10	7	0	0	96	27
6	1007	12	4	6	0	0	80	33
7	1008	12	10	6	2	0	90	39
8	1009	12	2	8	2	4	60	29
9	1010	12	6	9	1	0	85	30

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}")

Mean age: 31.3

Median age: 30.5

Standard Deviation of age: 5.271516754112509