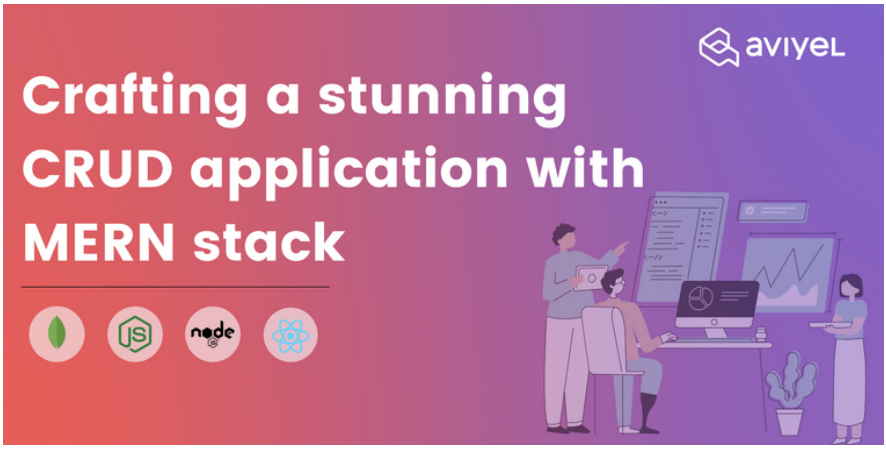
5 FULLSTACK MERN PROJECTS

REFINE YOUR MERN STACK SKILL

aviyel

All content is curated by the Aviyel team.

TABLE OF CONTENTS



In this section, we will set up a full-stack app to perform CRUD operations using the MERN stack, MongoDB for the database, Express and Node for the backend, and React as the frontend. This tutorial should help you understand the basic MERN stack CRUD operations.

Setting up Frontend

We'll start by setting up our frontend first using create-react-app. So, without further ado, let's get started.

Create a two folder name client and server inside your project directory, then open it in Visual Studio Code or any code editor of your choice.

Making Client directory

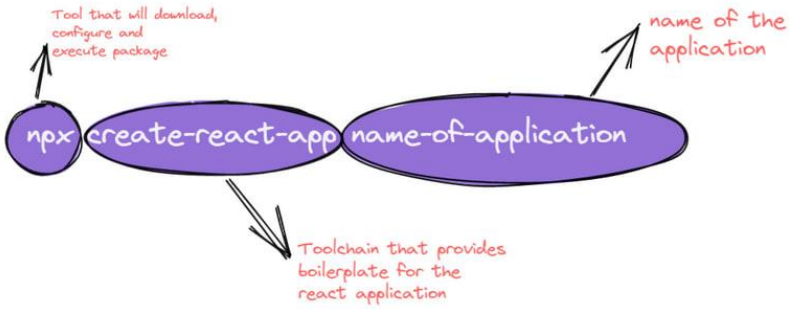
Making server directory

We will be building the UI and its functionalities from absolute ground level. Now, let's start and craft our application.

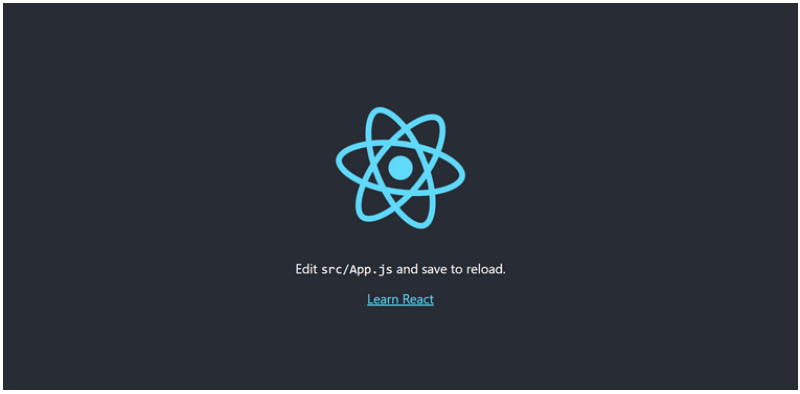
Installing react application

Let us begin with the frontend part and craft it using react. So, if Node.js isn't already installed on your system, the first thing you should do is install it. So, go to the official Node.js website and install the correct and appropriate version. We need node js so that we can use the node package manager, also known as NPM.

Now, open the client folder inside the code editor of your choice. For this tutorial, I will be using VScode. Next step, let's open the integrated terminal and type `npx create-react-app` . this command will create the app inside the current directory and that application will be named as client

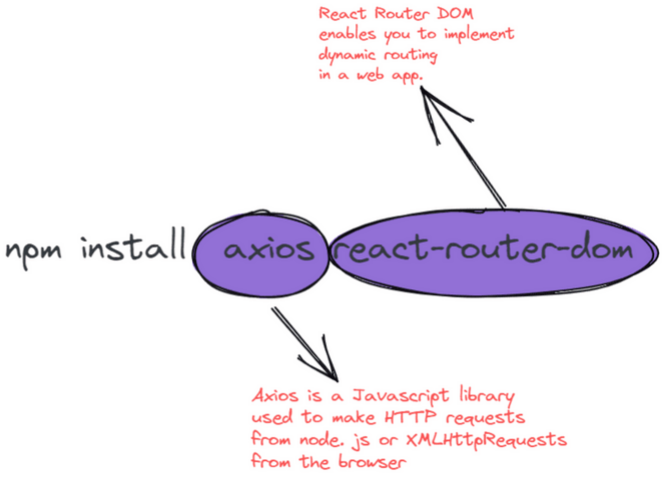


It usually takes only a few minutes to install. Normally, we would use npm to download packages into the project, but in this case, we are using npx, the package runner, which will download and configure everything for us so that we can start with an amazing template. It's now time to start our development server, so simply type npm start, and the browser will automatically open react-app.



Now, within the client folder install the following dependencies.

```
npm i axios react-router-dom
```

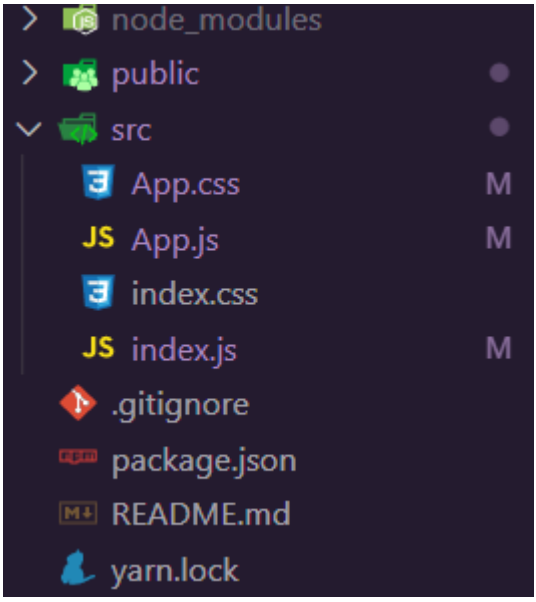


The "package.json" file should look like this after the dependencies have been installed.

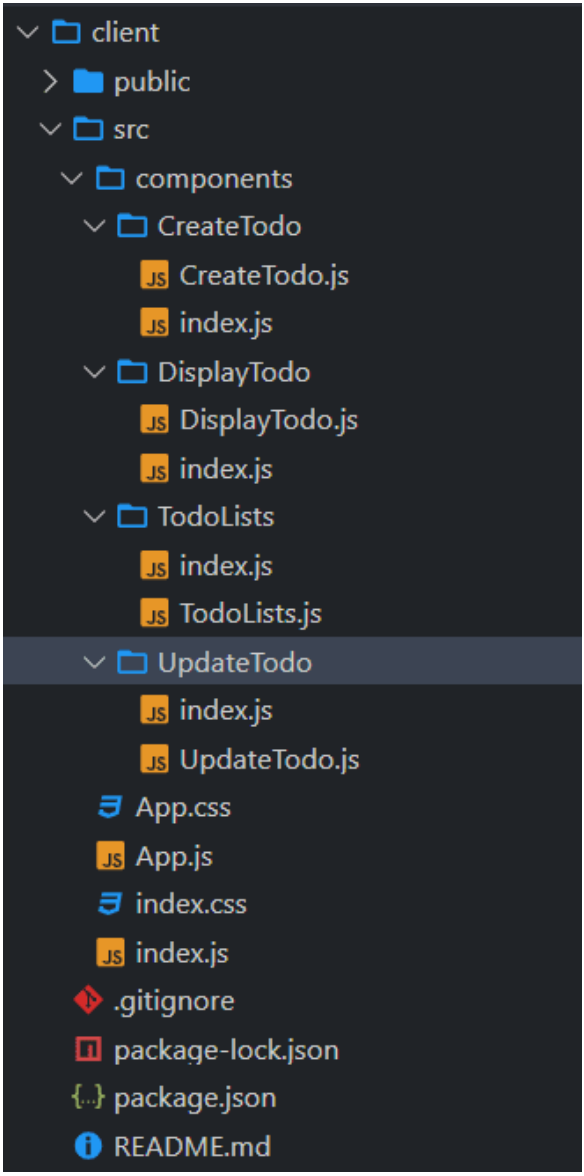
```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.14.1",
    "@testing-library/react": "^11.2.7",
    "@testing-library/user-event": "^12.8.3",
    "axios": "^0.24.0",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-router-dom": "^5.3.0",
    "react-scripts": "4.0.3",
    "web-vitals": "^1.1.2"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Project cleanup

Before we begin building our projects, we must first clean them up by removing some of the files provided by create-react-app. Your src files should look like this after you've cleaned them up



Now, within the src folder, make another folder called components, and within that folder, make three folder/components: DisplayTodo, CreateTodo, TodoLists and UpdateTodo.


client
public
src
components
CreateTodo
CreateTodo.js
index.js
DisplayTodo
DisplayTodo.js
index.js
TodoLists
index.js
TodoLists.js
UpdateTodo
index.js
UpdateTodo.js
App.css
App.js
index.css
index.js
.gitignore
package-lock.json
package.json
README.md

DisplayTodo

To begin, we will create the DisplayTodo component, which will read all of the documents created. The first step is, import the react useState and useEffect hooks and then import axios from the Axios package. We will retrieve documents from the database and store them in the state todoData in the DisplayTodo function component. To retrieve the document, we will use Axios to send a GET request to the backend. When we receive the data, we will use setTodoData to store it in todoData and log it. If we receive an error, we will also log it. Because we want the data to load when the page loads, we'll make the GET request from the useEffect hook.

```
// components/DisplayTodo.js

import { useState, useEffect } from "react";
import axios from "axios";
import { Link } from "react-router-dom";
import TodoLists from "../TodoLists";

const DisplayTodo = () => {
  const [infoTodo, setInfoTodo] = useState([]);

  useEffect(() => {
    axios
      .get("http://localhost:4000/api/todoapp")
      .then((res) => {
        console.log(res.data);
        setInfoTodo(res.data);
      })
      .catch((err) => {
        console.log(err);
      });
  }, []);

  const deleteHandler = (e) => {
    axios.delete(`http://localhost:4000/api/todoapp/${e.target.name}`);
    setInfoTodo((data) => {
      return data.filter((todo) => todo._id !== e.target.name);
    });
  };

  return (
    <section className="todo-container">
      <Link to="/add-todo" className="todo-btn-new">
        <button className="todo-btn">Add new todo</button>
      </Link>
      <section className="todo-data">
        <h1></h1>
        <ul className="todo-list-block">
          {infoTodo.map((data) => (
            <TodoCard data={data} deleteHandler={deleteHandler} />
          ))}
        </ul>
      </section>
    </section>
  );
};
export default DisplayTodo;
```

TodoList

Then we'll make the TodoList component to display the todo's contents. We will iterate over `todoData` and pass the contents to `TodoList` , which will display the contents of each to-do document.

```
// components/TodoList.js

import React from "react";

const TodoList = ({ todoInfos, deleteHandler }) => {
  const { _id, tilte, description } = todoInfos;
  return (
    <li key={_id}>
      <div className="title-description">
        <h3>{title}</h3>
        <p>{description}</p>
      </div>

      <div className="button-container">
        <button name={_id} className="button">
          🖊️
        </button>
        <button name={_id} className="button"
onClick={deleteHandler}>
          🗑️
        </button>
      </div>
    </li>
  );
};
```

CreateTodo

To create a new todo, we will use Axios to send a POST request to our server. So let's import the react useState hook and after that, import the Link from react-router-dom.
Now, create a function handler change that and you'll get the input data again create a new function handler. Finally, Submitting this will cause the POST request to be sent to the server. Declare data using the useState hook and the JSON below.

```
"description": "", "title": ""
```

When the input changes, we will update the data in the handleChange method. We'll call setTodoInfo() and declare an arrow function inside that will copy the previous data's contents if any exist. In this case, e.target.name will be the name of the input element, which will either have a title or a description. In the submitHanlder method, To prevent the page from reloading when the submit button is clicked, use e.preventDefault() Send the data in the form of a POST request to the server. If the data was successfully transmitted to the server, the state data should be reset.

```
// components/CreateTodo.js
import { useState } from "react";
import axios from "axios";

const CreateTodo = () => {
  const [todoInfo, setTodoInfo] = useState({ title: "", description: ""
});

  function handleChange(e) {
    setTodoInfo((data) => ({ ...data, [e.target.name]: e.target.value
}));
  }

  function handleSubmit(e) {
    e.preventDefault();

    axios
      .post("http://localhost:4000/api/todoapp", todoInfo)
      .then((res) => {
        setTodoInfo({ title: "", description: "" });
        console.log(res.data.message);
      })
      .catch((err) => {
        console.log("Error couldn't create TODO");
        console.log(err.message);
      });
  }

  return (
    <section className="container">
      <button type="button" className="todo-btn todo-btn-back">
        🔙 back
      </button>

      <section className="todo-data">
        <form onSubmit={handleSubmit} className="form-container"
noValidate>
          <label className="label" htmlFor="title">
            Todo Title
          </label>
```

```
          <input
            type="text"
            name="title"
            value={todoInfo.title}
            onChange={handleChange}
            className="input"
          />
          <label className="label" htmlFor="description">
            Describe it !
          </label>
          <input
            type="textarea"
            name="description"
            value={todoInfo.description}
            onChange={handleChange}
            className="input"
          />
          <button type="submit" className="todo-btn">
            ➕ create todo
          </button>
        </form>
      </section>
    </section>
  );
};

export default CreateTodo;
```

Now, let's define a deleteHandler function inside DisplayTodo component that will send a DELETE request to the server. To delete a document from the database, this function will require the document's _id. It will also add the filtered array to the array todo. TodoList component accepts the deleteHandler method as a parameter. TodoList component should be updated to include the deleteHandler parameter.

Add a onClick event for the delete button and pass the deleteHandler method as a parameter.
After making the aforementioned changes, the code will look something like this.


```
//components/DisplayTodo.js

import { useState, useEffect } from "react";
import axios from "axios";
import TodoLists from "../TodoLists";

const DisplayTodo = () => {
  const [infoTodo, setInfoTodo] = useState([]);
  const [id, setId] = useState("");
  const [update, setUpdate] = useState(false);
  const [infoTodo, setInfoTodo] = useState([]);
  const [modal, setModal] = useState(false);

  useEffect(() => {
    axios
      .get("http://localhost:8000/api/todo")
      .then((res) => {
        console.log(res.data);
        setInfoTodo(res.data);
      })
      .catch((err) => {
        console.log(err);
      });
  }, []);

  const updateHandler = () => {
    setUpdate(!update);
  };

  const closeHandler = () => {
    setId("");
    setModal(false);
```

```
  };

  const deleteHandler = (e) => {
    axios.delete(`http://localhost:8000/api/todo/${e.target.name}`);
    setInfoTodo((data) => {
      return data.filter((todo) => todo._id !== e.target.name);
    });
  };

  return (
    <section className="container">
      <button className="todo-btn">➕ Add new todo</button>
      <section className="todo-data">
        <h1></h1>
        <ul className="todo-list-block">
          {infoTodo.map((todoInfo, index) => (
            <TodoLists
              key={index}
              todoInfos={todoInfo}
              deleteHandler={deleteHandler}
            />
          ))}
        </ul>
      </section>
      {modal ? (
        <section className="update-container">
          <div className="update-todo-data">
            <p onClick={closeHandler} className="close">
              &times;
            </p>
          </div>
        </section>
      ) : (
        ""
      )}
    </section>
  );
};

export default DisplayTodo;
```

TodoList component should look something like this:


```
// components/TodoList.js

import React from "react";

const TodoLists = ({ todoInfos }) => {
  const { _id, title, description } = todoInfos;

  return (
    <li key={_id}>
      <div className="title-description">
        <h2>{title}</h2>
        <h1></h1>
        <p>{description}</p>
      </div>
      <h1></h1>
      <div className="todo-btn-container">
        <button className="todo-btn" name={_id}>
          🖊️
        </button>
        <button className="todo-btn" name={_id}>
          🗑️
        </button>
      </div>
    </li>
  );
};

export default TodoLists;\
```

We must first update the App.js file before we can use the CreateTodo component. BrowserRouter and Route should be imported from react-router-dom. Import the CreateTodo component from the components/createTodo directory. Create a Route for the home page and pass the ShowTodoList component through it and make a Route for adding a new todo /add-list and wrap the Routes within the BrowserRouter.
After you've made the changes, the App.js file should look like this.


```
// App.js

import { BrowserRouter, Route } from "react-router-dom";
import DisplayTodo from "./components/DisplayTodo";
import CreateTodo from "./components/CreateTodo";
import "./App.css";

function App() {
  return (
    <div className="todo-Container">
      <BrowserRouter>
        <Route exact path="/" component={DisplayTodo} />
        <Route path="/add-list" component={CreateTodo} />
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Now, import the Link from react-router-dom. and wrap a button within a Link tag. After you've made the changes, the DisplayTodo should look like this.


```
// components/DisplayTodo.js

import { useState, useEffect } from "react";
import axios from "axios";
import { Link } from "react-router-dom";
import TodoLists from "../TodoLists";

export function DisplayTodo() {
  const [id, setId] = useState("");
  const [update, setUpdate] = useState(false);
  const [infoTodo, setInfoTodo] = useState([]);
  const [modal, setModal] = useState(false);

  useEffect(
    function () {
      axios
        .get("http://localhost:4000/api/todoapp")
        .then((res) => {
          setInfoTodo(res.data);
        })
        .catch((err) => {
          console.log(err.message);
        });
    },
    [update]
  );
  const editHandler = (e) => {
    setId(e.target.name);
    setModal(true);
  };

  const updateHandler = () => {
    setUpdate(!update);
  };
```

```
  const deleteHandler = (e) => {
    axios.delete(`http://localhost:4000/api/todoapp/${e.target.name}`);

    setInfoTodo((data) => {
      return data.filter((todo) => todo._id !== e.target.name);
    });
  };

  const closeHandler = () => {
    setId("");
    setModal(false);
  };
  return (
    <section className="container">
      <Link to="/add-list" className="button-new">
        <button className="todo-btn">➕ Add new todo</button>
      </Link>
      <section className="todo-data">
        <h1></h1>
        <ul className="todo-list-block">
          {infoTodo.map((todoInfo, index) => (
            <TodoLists
              key={index}
              todoInfos={todoInfo}
              editHandler={editHandler}
              deleteHandler={deleteHandler}
            />
          ))}
        </ul>
      </section>
      {modal ? (
        <section className="update-container">
          <div className="update-todo-data">
            <p onClick={closeHandler} className="close">
              &times;
            </p>
          </div>
        </section>
      ) : (
        ""
      )}
    </section>
  );
}

export default DisplayTodo;
```

Now again, import the Link from react-router-dom and wrap a button within a Link tag. After you've made the changes, the CreateTodo should look like this.


```
// components/CreateTodo.js

import { useState } from "react";
import { Link } from "react-router-dom";
import axios from "axios";

const CreateTodo = () => {
  const [todoInfo, setTodoInfo] = useState({ title: "", description: "" });

  function handleChange(e) {
    setTodoInfo((data) => ({ ...data, [e.target.name]: e.target.value }));
  }

  function handleSubmit(e) {
    e.preventDefault();

    axios
      .post("http://localhost:4000/api/todoapp", todoInfo)
      .then((res) => {
        setTodoInfo({ title: "", description: "" });
        console.log(res.data.message);
      })
      .catch((err) => {
        console.log("Error couldn't create TODO");
        console.log(err.message);
      });
  }

  return (
    <section className="container">
      <Link to="/">
        <button type="button" className="todo-btn todo-btn-back">
          🔙 back
        </button>
```

```
      </Link>

      <section className="todo-data">
        <form onSubmit={handleSubmit} className="form-container" noValidate>
          <label className="label" htmlFor="title">
            Todo Title
          </label>
          <input
            type="text"
            name="title"
            value={todoInfo.title}
            onChange={handleChange}
            className="input"
          />
          <label className="label" htmlFor="description">
            Describe it !
          </label>
          <input
            type="textarea"
            name="description"
            value={todoInfo.description}
            onChange={handleChange}
            className="input"
          />
          <button type="submit" className="todo-btn">
            ➕ create todo
          </button>
        </form>
      </section>
    </section>
  );
};

export default CreateTodo;
```

Now, import the useState from react and import axios from the axios package. Finally, The UpdateTodo component will have three properties._id ,closeHandler , updateHandler
The UpdateTodo component may look something like this.

<UpdateTodo/>

```
//components/UpdateTodo.js

import { useState } from "react";
import axios from "axios";

function UpdateTodo({ _id, closeHandler, updateHandler }) {
  const [todoInfo, setTodoInfo] = useState({ title: "", description: "" });

  const handleChange = (e) => {
    setTodoInfo((data) => ({ ...data, [e.target.name]: e.target.value }));
  };

  const submitHanlder = (e) => {
    e.preventDefault();

    axios
      .put(`http://localhost:4000/api/todoapp/${_id}`, todoInfo)
      .then((res) => {
        setTodoInfo({ title: "", description: "" });
      })
      .catch((err) => {
        console.error(err);
      });
  };

  return (
    <form
      className="form-container"
      onSubmit={(e) => {
        submitHanlder(e);
        updateHandler();
        closeHandler();
      }}
    >
      <label htmlFor="title" className="label">
        Todo Title
      </label>
      <input
        type="text"
        name="title"
```

```
        className="input"
        onChange={handleChange}
      />
      <label htmlFor="description" className="label">
        Todo Description
      </label>
      <input
        type="textarea"
        name="description"
        className="input"
        onChange={handleChange}
      />
      <button type="submit" className="todo-btn">
        ➕ Add
      </button>
    </form>
  );
}
export default UpdateTodo;
```

Import the UpdateTodo component from UpdateTodo.js and then declare modal with the useState hook which is set to false by default. The modal value will be either true or false. The UpdateTodo component will be rendered conditionally if the edit button is pressed on any of the todo, we will set setModal to true when the UpdateTodo component is rendered, and then declare id using the useState hook. The _id of the todo that needs to be updated will be saved.It will be passed to the UpdateTodo component as a prop. Use the useState hook to declare an update. This will be used to retrieve all of the to-do items from the database. When a todo document is updated, the update will toggle between true and false. Now, define a function editHandler this function will replace the state id with the document's _id and set the modal state to true. Next, create a function called

updateHandler. If the todo has been updated by the user, this will invert the state of the update. Inverting the state will cause the useEffect hook to update the todo array. Finally, define a function closeHandler, which will be used to close the UpdateTodo component. This will set the id to an empty string and the modal property to false.
After you've made the changes, the DisplayTodo and TodoList should look like this.

```
//components/DisplayTodo.js

import { useState, useEffect } from "react";
import axios from "axios";
import { Link } from "react-router-dom";
import UpdateTodo from "../UpdateTodo";
import TodoLists from "../TodoLists";

export function DisplayTodo() {
  const [id, setId] = useState("");
  const [update, setUpdate] = useState(false);
  const [infoTodo, setInfoTodo] = useState([]);
  const [modal, setModal] = useState(false);

  useEffect(
    function () {
      axios
        .get("http://localhost:4000/api/todoapp")
        .then((res) => {
          setInfoTodo(res.data);
        })
        .catch((err) => {
          console.log(err.message);
        });
    },
    [update]
  );

  const editHandler = (e) => {
    setId(e.target.name);
    setModal(true);
```

```
};

const updateHandler = () => {
  setUpdate(!update);
};

const deleteHandler = (e) => {
  axios.delete(`http://localhost:4000/api/todoapp/${e.target.name}`);

  setInfoTodo((data) => {
    return data.filter((todo) => todo._id !== e.target.name);
  });
};

const closeHandler = () => {
  setId("");
  setModal(false);
};

return (
  <section className="container">
    <Link to="/add-list" className="button-new">
      <button className="todo-btn">➕ Add new todo</button>
    </Link>
    <section className="todo-data">
      <h1></h1>
      <ul className="todo-list-block">
        {infoTodo.map((todoInfo, index) => (
          <TodoLists
            key={index}
            todoInfos={todoInfo}
            editHandler={editHandler}
            deleteHandler={deleteHandler}
          />
        ))}
      </ul>
    </section>
    {modal ? (
      <section className="update-container">
        <div className="update-todo-data">
          <p onClick={closeHandler} className="close">
            &times;
          </p>

          <UpdateTodo
            _id={id}
```

```
              closeHandler={closeHandler}
              updateHandler={updateHandler}
            />
          </div>
        </section>
      ) : (
        ""
      )}
    </section>
  );
}

export default DisplayTodo;
```

<TodoList />

```
//components/TodoList.js
import React from "react";
const TodoLists = ({ todoInfos, editHandler, deleteHandler }) => {
const { _id, title, description } = todoInfos;
  return (
    <li key={_id}>
      <div className="title-description">
        <h2>{title}</h2>
        <h1></h1>
        <p>{description}</p>
      </div>
      <h1></h1>
      <div className="todo-btn-container">
        <button className="todo-btn" name={_id} onClick={editHandler}>
          🖊
        </button>
        <button className="todo-btn" name={_id} onClick={deleteHandler}>
          🗑
        </button>
      </div>
    </li>
  );
};

export default TodoLists;
```

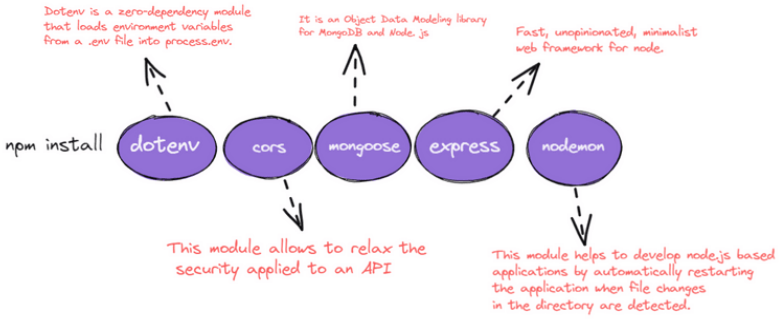
Finally, let's incorporate some styles into our project. Now, go to your App.css file and update your style, or simply copy and paste the following CSS code.

https://gist.github.com/pramit-marattha/e88d83b66ce7ca9a01e840f486cf9fc8

Setting up Backend

Now,we'll start by setting up our backend with npm and installing relevant packages, then set up a MongoDB database, then set up a server with Node and Express, then design a database schema to define a Todo, and then set up API routes to create, read, update, and delete documents from the database.
Now go to your server directory and use the command prompt to run the code below.
npm init -y

Updating package.json
To install the dependencies, use the following instructions in the terminal.
npm install cors express dotenv mongoose nodemon




```
Dotenv: To retrieve data from.env files

Express: It is a web application framework written in/for Node.js.

Mongoose: It's a MongoDB Object Data Modeling (ODM) library built on Node.

Nodemon: It will keep the server running eternally.

Cors: It Allows API queries from multiple sources.
```

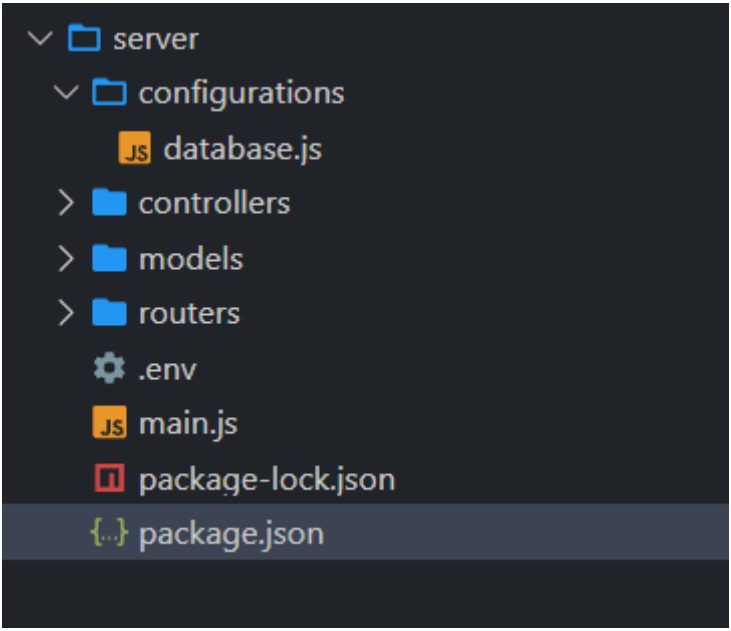
The "package.json" file should look like this after the dependencies have been installed.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon main.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^10.0.0",
    "express": "^4.17.1",
    "mongoose": "^6.0.12",
    "nodemon": "^2.0.14"
  }
}
```

And also, remember to update the scripts as well.

Structuring the folder inside the server:

- ***configurations*** *:* Make a file called database.js in the config folder. The necessary code for connecting to the MongoDB database will be contained in this file.
- ***controllers****:* The files in the controllers' folder will contain the methods for the endpoints to interface with the database.
- ***models****:* The files that specify the MongoDB schema will be found in the model's folder.
- ***routers****:* The files with the endpoints will be found in the routers folder.



Configuring main.js

- Import express module.
- Use express() to start our app.
- Using the app, create a get method for the endpoint http://localhost:4000.
- For our server to run, set the PORT to 4000.
- Using our app, you may listen to PORT.

```
const express = require("express");
const cors = require("cors");

const dotenv = require("dotenv");

dotenv.config();

const app = express();

const PORT = process.env.PORT || 5000;

// listen
app.listen(PORT, () =>
    console.log(`Server is running on
http://localhost:${PORT}`)
);
```

Now open your .env file, or create one if you don't have one, and paste the following code inside it.

```
PORT=4000
```

Now use the following code to start the server with nodemon. Ensure that the following command is executed from the project directory.
npm start

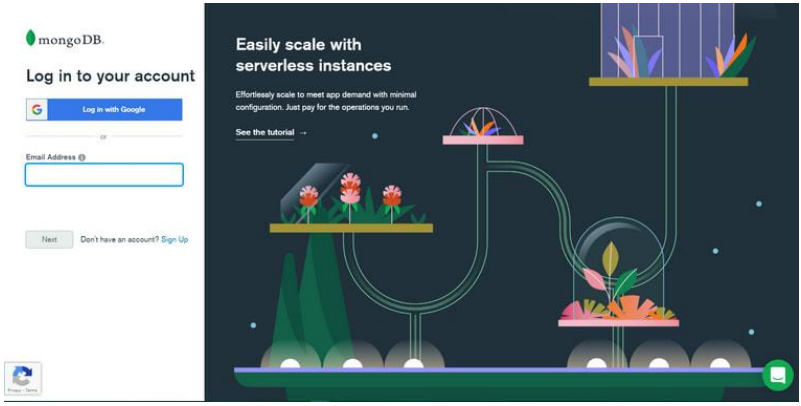
If the server has started successfully, the terminal should display the following message.

Getting started with MongoDB

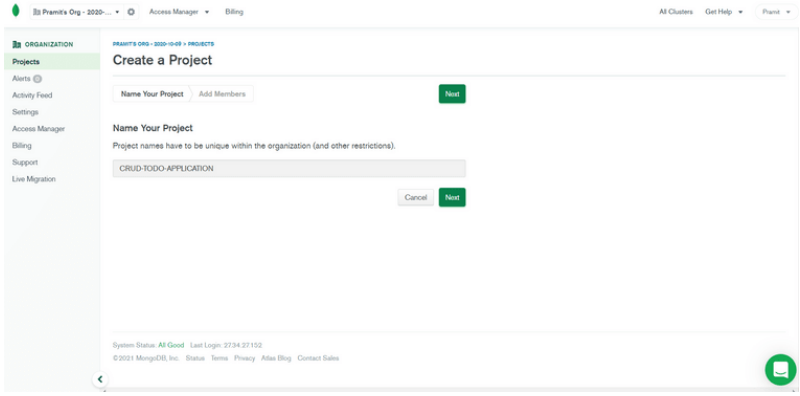
So, what is MongoDB?

MongoDB is an open source, cross-platform document-oriented database program. MongoDB is a NoSQL database that uses JSON-like documents and optional schemas to store data. MongoDB is a database developed by MongoDB Inc. and published under the terms of the Server Side Public License.

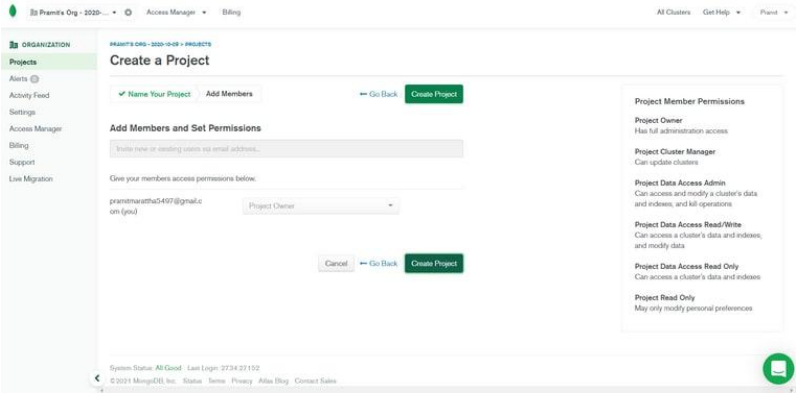
Sign in to MongoDB



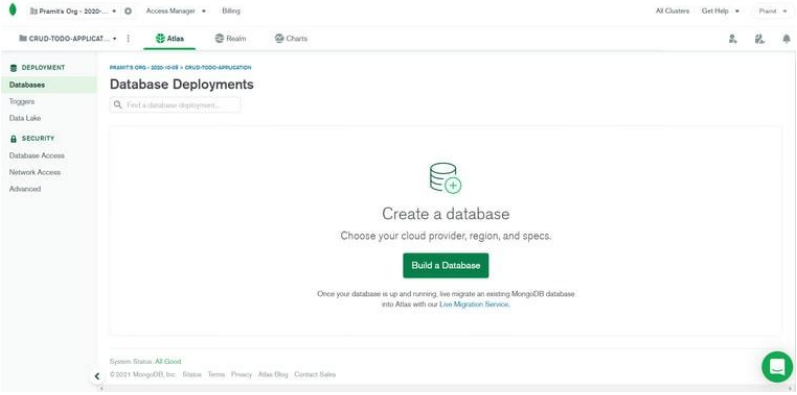
Make a new project.



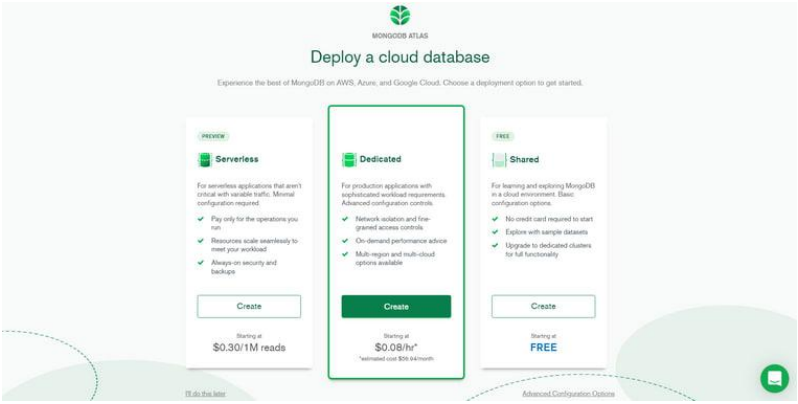
Create a Project



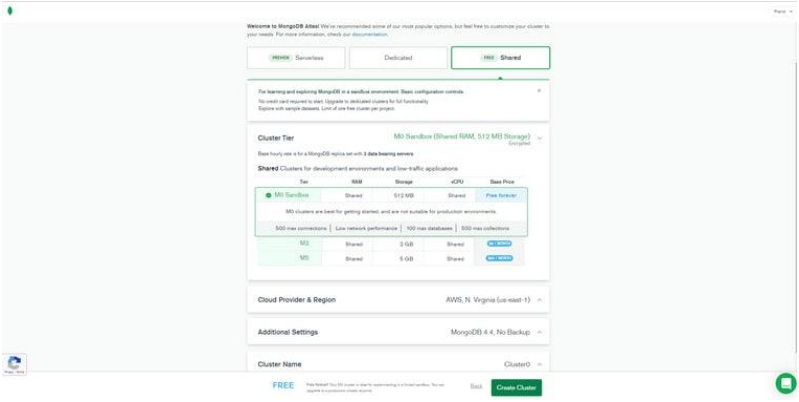
Building a database



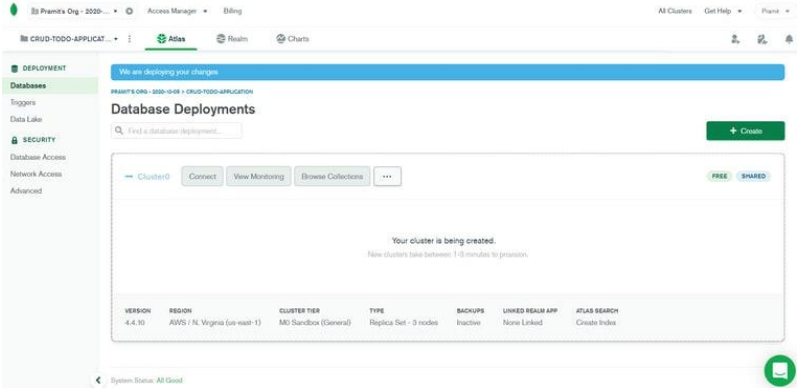
Creating a cluster



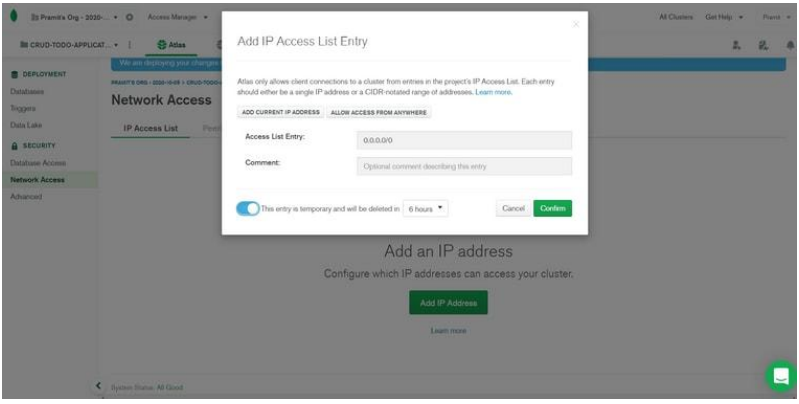
Selecting a cloud service provider



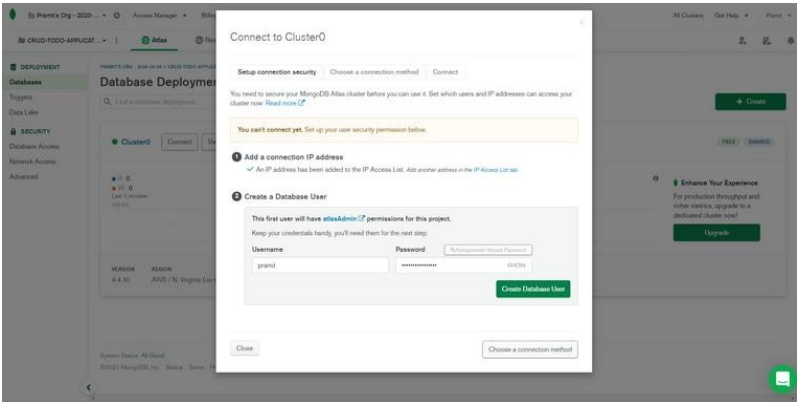
Make a cluster and wait for the cluster to be built before proceeding (usually takes around 5 -10 minutes)



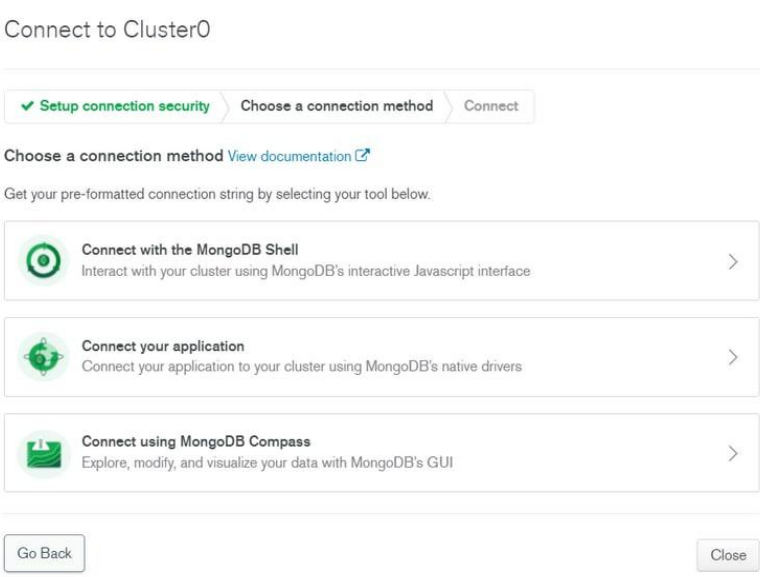
Allow access from anywhere by clicking connect. Then IP address should be added.



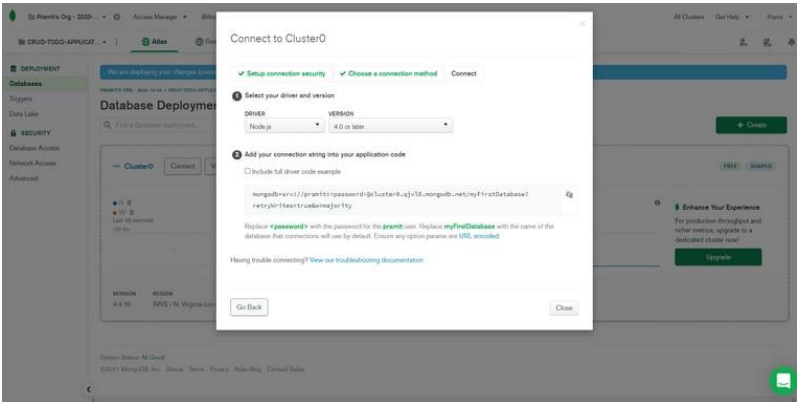
In the database, create a user. You'll need the username and password for the MongoDB URI and finally create a database user.



Now, select the Choose a connection method.



Connect your application by clicking on it and finally select the correct driver and version.



Insert mongodb+srv into the .env file.

```
PORT=4000
DATABASE_URL=mongodb+srv://pramit:<password>@cluster0.qjvl6.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
```

Now open the database.js file located inside the configurations folder and make the modifications listed below.
Import Mongoose module.

Import dotenv package and configure it. Create DATABASE_URL inside env file and add your credential inside it and then you are able to import it from the .env file.

Importing dotenv module

```
const dotenv = require('dotenv');
dotenv.config();

const databaseURL = process.env.DATABASE_URL;
```

Define the databaseConfiguration method for establishing a database connection.

The databaseConfiguration method should be exported and called in main.js.

```
const databaseConfiguration = async () => {
  try {
    await mongoose.connect(databaseURL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Database connected');
  } catch (err) {
    console.log(err);
    process.exit(1);
  }
};

module.exports = databaseConfiguration;
```

Now, database.js file should resemble something like this.

database.js

```
//database.js

const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config();

const databaseURL = process.env.DATABASE_URL;

const databaseConfiguration = async () => {
  try {
    await mongoose.connect(databaseURL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Database connected');
  } catch (err) {
    console.log(err);
    process.exit(1);
  }
};

module.exports = databaseConfiguration;
```

Add the following changes on main.js file

main.js

```
// main.js
const express = require('express');
const cors = require('cors');
const databaseConfiguration =
require('./configurations/database.js');

const dotenv = require('dotenv');

dotenv.config();

const app = express();

const PORT = process.env.PORT || 5000;

//connecting to the mongodb database
databaseConfiguration();

// add the middlewares
app.use(express.json({ extended: false }));
app.get('/', (req, res) => res.send('<h1>Server is up and
running</h1>'));

// listen
app.listen(PORT, () =>
  console.log(`Server is running on http://localhost:${PORT}`)
);
```

Adding database schema:

Add a models.js file inside the models folder. We will define the entire database schema inside this particular file.


```
// models.js
const mongoose = require('mongoose');

const TodoListSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
  },
  description: {
    type: String,
  },
  date: {
    type: Date,
    default: Date.now,
  },
});

const Todo = mongoose.model('todo', TodoListSchema);

module.exports = Todo;
```

Defining the entire endpoint of our API

todo.routes.js

```
//todo.routes.js

const express = require("express");

const router = express.Router();

const {
  listAllTodo,
  createOneTodo,
  updateOneTodo,
  deleteTodo,
} = require("../controllers/todo.controller.js");

router.get("/", listAllTodo);

router.post("/", createOneTodo);

router.put("/:id", updateOneTodo);

router.delete("/:id", deleteTodo);

module.exports = router;
```

Defining the methods for endpoint

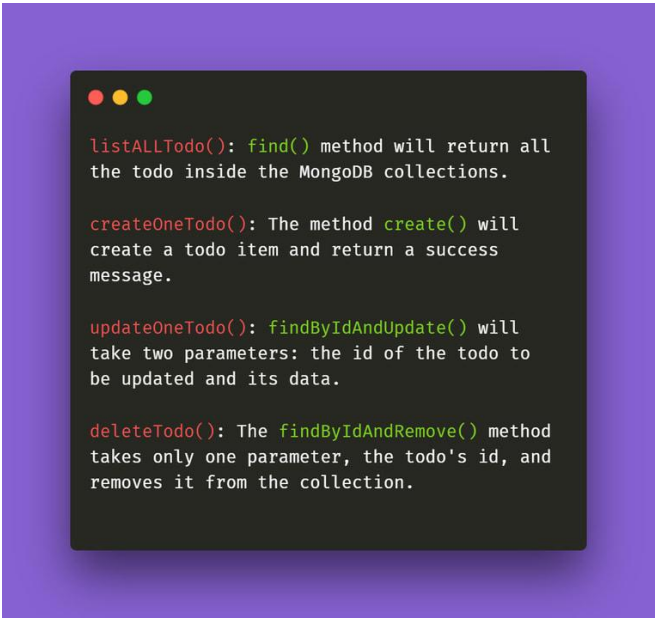
The methods for the endpoints will be defined in the controllers folder and inside controllers.js file.
Now open the controllers.js file located inside the controllers folder and make the modifications listed below.

controllers.js

```
//controllers.js

const AppTodo = require("../models/models.js");
exports.createOneTodo = (req, res) => {
  AppTodo.create(req.body)
    .then((todo) => {
      console.log({ todo });
      res.json({
        message: "Cheers!! You have successfully added TODO",
        todo,
      });
    })
    .catch((err) => {
      res.status(404).json({
        message: "Sorry your todo list cannot be added",
        error: err.message,
      });
    });
};

exports.listAllTodo = (req, res) => {
  AppTodo.find()
    .then((todo) => {
```

```
      console.log({ todo });
      res.json(todo);
    })
    .catch((err) => {
      res
        .status(404)
        .json({ message: "There isnt any todo available",
error: err.message });
    });
};
exports.updateOneTodo = (req, res) => {
  AppTodo.findByIdAndUpdate(req.params.id, req.body)
    .then((todo) => {
      console.log({ todo });
      return res.json({
        message: "Cheers!! You have successfully updated
TODO",
        todo,
      });
    })
    .catch((err) => {
      res.status(404).json({
        message: "Sorry your todo list cannot be updated",
        error: err.message,
      });
    });
};
exports.deleteTodo = (req, res) => {
  AppTodo.findByIdAndRemove(req.params.id, req.body)
    .then((todo) => {
      console.log({ todo });
      res.json({
        message: "Cheers!! You have successfully deleted your
TODO",
        todo,
      });
```

```
    })
    .catch((err) => {
      res.status(404).json({
        message: "Sorry your todo is not there",
        error: err.message,
      });
    });
};
```



Finally, add the endpoint to the main.js file. Also, don't forget to include the cors so that we can make API calls from the frontend application. As a result, your main.js file should look something like this.

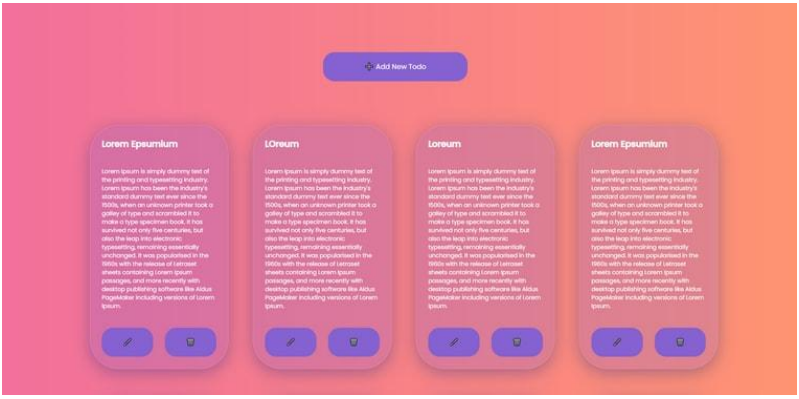
main.js

```
//main.js
const express = require("express");
const cors = require("cors");
const databaseConfiguration =
require("./configurations/database.js");
const todo = require("./routes/todo.routes.js");

const dotenv = require("dotenv");

dotenv.config();
const app = express();

const PORT = process.env.PORT || 5000;

//connecting to  mongodb
databaseConfiguration();
//adding cors
app.use(cors({ origin: true, credentials: true }));

// adding  middlewares
app.use(express.json({ extended: false }));
app.get("/", (req, res) =>
  res.send("Hello there!! Cheers !! The server is up and
running")
);

// using the todo routes
app.use("/api/todoapp", todo);

// listen
app.listen(PORT, () =>
  console.log(`Server is running on http://localhost:${PORT}`)
);
```

After restarting the server, you should see something similar to this:

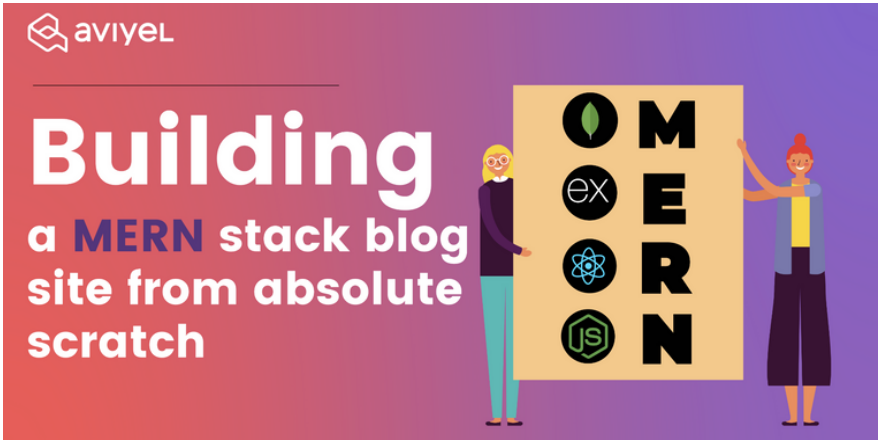
```
> rest@1.0.0 start
> nodemon server.js

[nodemon] 2.0.13
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server is running on http://localhost:4000
Database connected
```

Finally, start both the client and the server, and you should see something similar to this.



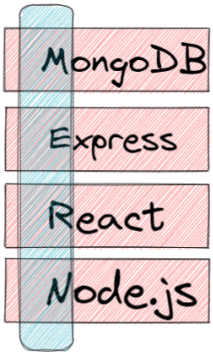
The complete source code for the application can be found here:

https://github.com/pramit-marattha/MERN-awesome-crud



Even a few years ago, web application development was not what it is today. Nowadays, there are so many options that the uninitiated are frequently perplexed as to what is best for them. This applies not only to the overall stack but also to the development tools; there are so many options. This tutorial asserts that the MERN stack is ideal for developing a full web application, and it walks the reader through the entire process in a very detailed manner.

So, what exactly is the MERN stack?

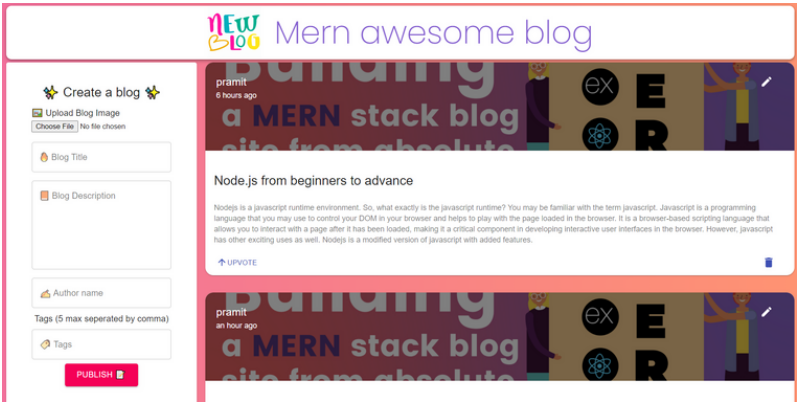


The MERN stack is a popular set of technologies for creating a modern Single Page Application (SPA). MERN stands for MongoDB, Express, React, and Node.js:

- **Node.js** is a popular server-side framework that allows us to run JavaScript code on a web server.
- **Express** is a Node.js web application framework that makes Node application development simpler and faster.
- **MongoDB** is a NoSQL database that stores data persistently in the form of collections and documents.
- **React** is a JavaScript frontend library for creating user interfaces.

In thissection, we will create a full-stack blog application that performs CRUD operations by utilizing the MERN stack. This tutorial should help you understand the fundamental operations of the MERN stack.

Here is our application's final sneak peek.



Setting up the backend

Create a two folder name client and server inside your project directory, then open it inside the Visual Studio Code or any code editor of your choice.

Now, we'll begin by configuring our backend with npm and installing necessary packages, followed by configuring a MongoDB database, configuring a server with Node and Express, designing a database schema to define our Blog, and configuring API routes to create, read, update, and delete blog data and information from the database.

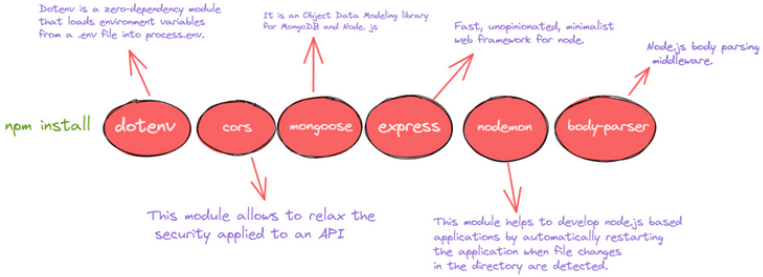
so, now navigate to your server's directory and run the code below from the command prompt.

```
npm init -y
```

Updating package.json

To install the dependencies, execute the following commands in the terminal.

```
npm install cors express dotenv mongoose
nodemon body-parser
```




```
PS D:\AviyelDemo\mern-awesome-blog\mern-awesome-blog\server> npm install cors express dotenv mongoose nodemon body-parser

up to date, audited 197 packages in 1s

19 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

The "**package.json**" file should look like this after the dependencies have been installed.

```
{
  "name": "server",
  "version": "1.0.0",
  "main": "server.js",
  "type": "module",
  "devDependencies": {},
  "scripts": {
    "start": "nodemon server.js"
  },
  "dependencies": {
    "body-parser": "^1.19.0",
    "cors": "^2.8.5",
    "dotenv": "^10.0.0",
    "express": "^4.17.1",
    "mongoose": "^6.0.12",
    "nodemon": "^2.0.14"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

And also, remember to update the scripts and type as well.

Now, navigate to your server directory and create a server.js file within it.

Configuring server.js

- Import express module.
- Import bodyParser module
- Import mongoose module
- Import CORS module
- Use express() to start our app.

```
//server.js
import express from "express";
import bodyParser from "body-parser";
import mongoose from "mongoose";
import cors from "cors";

const app = express();
```

Now we can use all of the different methods on that app instance. First, let's do some general setup. We'll use app.use and simply pass the bodyParser and limit it by 20 to 50mb because we're sending some images that can be very large in size, and we'll also specify the extended to true and do the same thing with the bodyParser URL encoded and pass the same parameter and now we are also going to use the CORS and call it as a function.

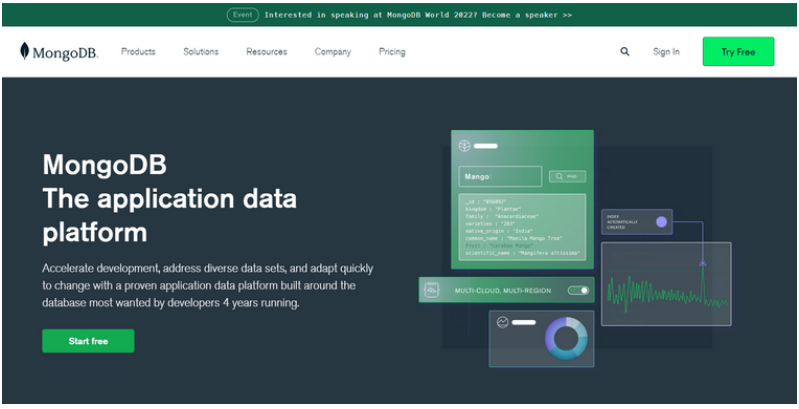
```
//server.js
import express from "express";
import bodyParser from "body-parser";
import mongoose from "mongoose";
import cors from "cors";

const app = express();

app.use(bodyParser.json({ limit: '50mb', extended: true }))
app.use(bodyParser.urlencoded({ limit: '50mb', extended: true
}))
app.use(cors());
```

Now it's time to link our server application to the real database, so we'll utilize the MongoDB database, especially the MongoDB cloud Atlas version, which means we'll be hosting our database onto their cloud.

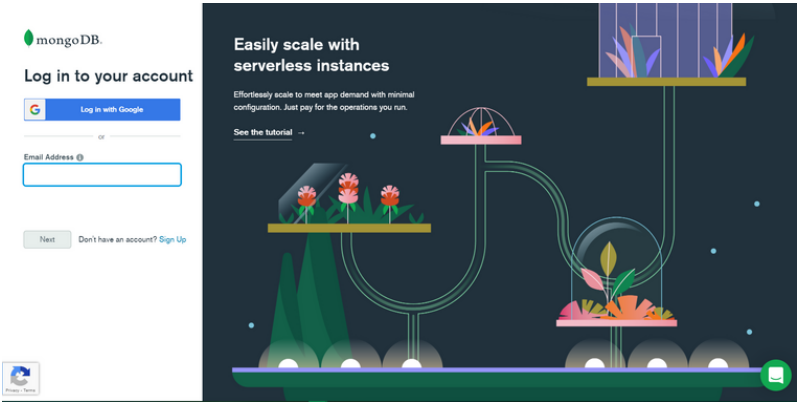
Setting up MongoDB cloud cluster

MongoDB is a document-oriented database that is open source and cross-platform. MongoDB is a NoSQL database that stores data in JSON-like documents with optional schemas. MongoDB is a database created and distributed by MongoDB Inc. under the provisions of the Server Side Public License.

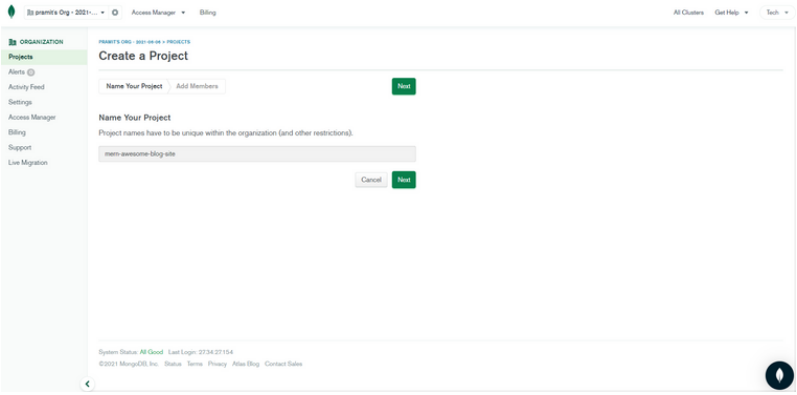
Official MongoDB website



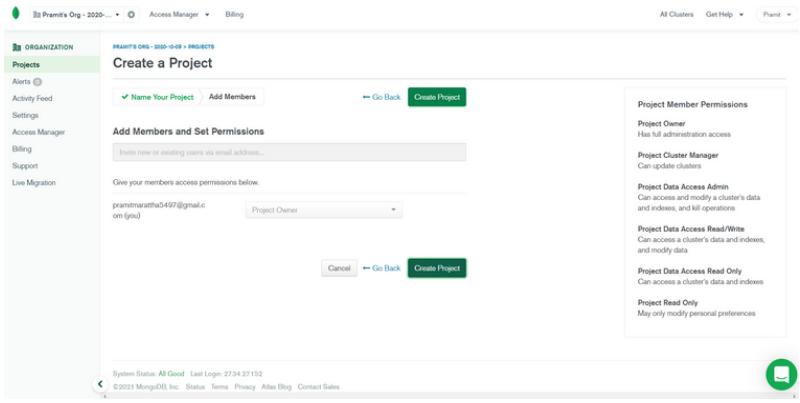
Sign in to MongoDB



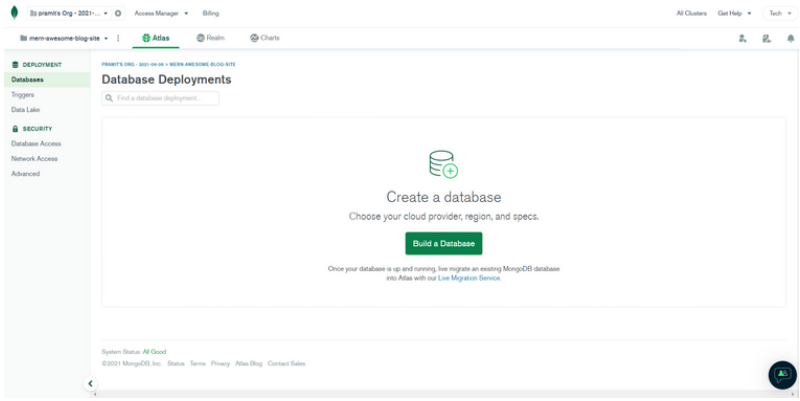
Create a Project



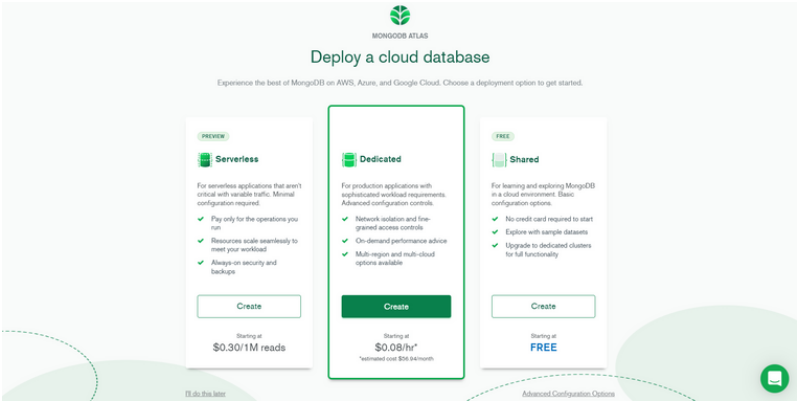
Adding members



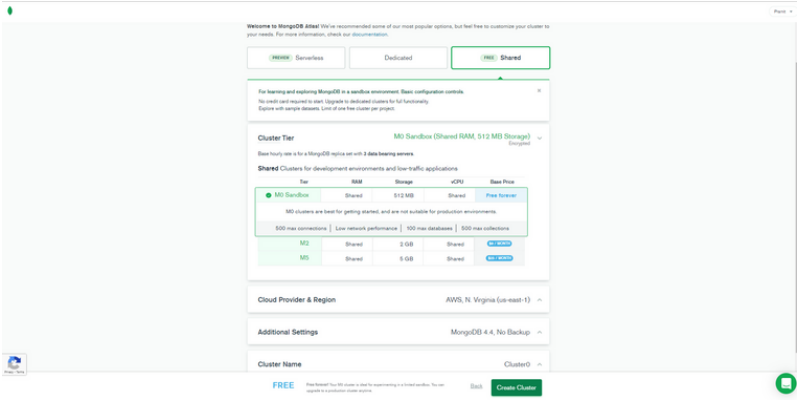
Building a database



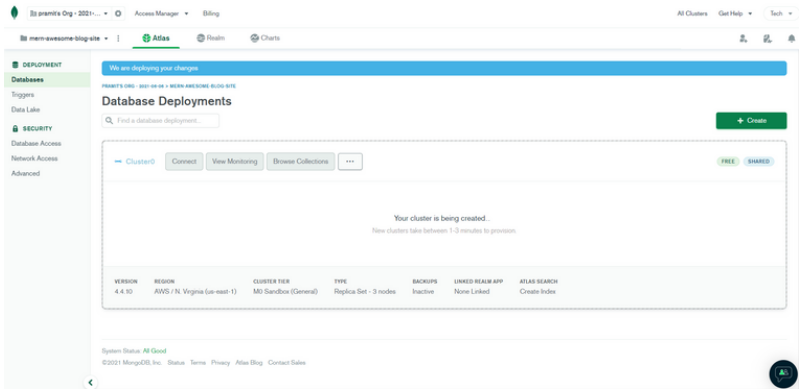
Creating a cluster



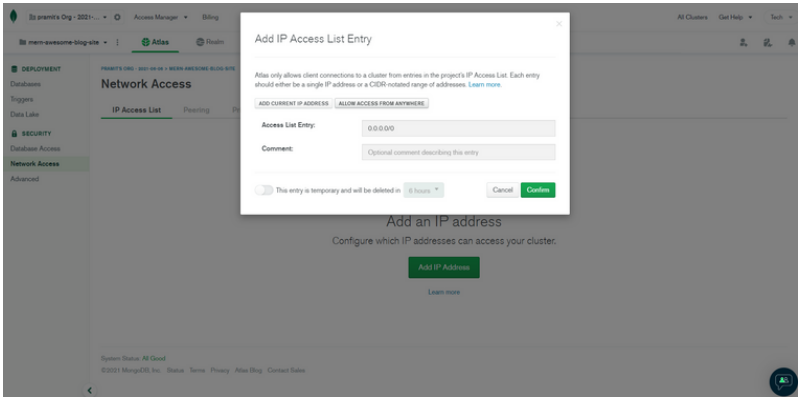
Selecting a cloud service provider



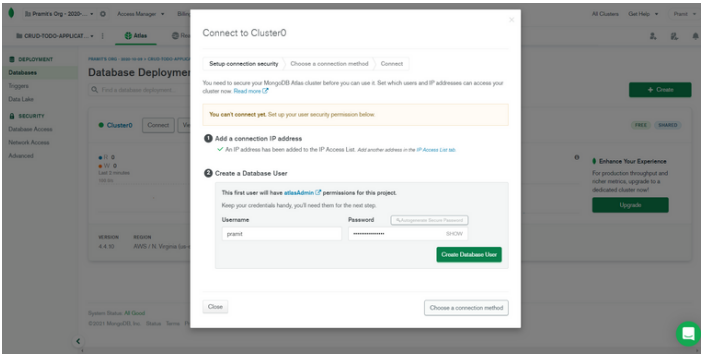
Make a cluster and wait for the cluster to be built before proceeding (usually takes around 5 -10 minutes)



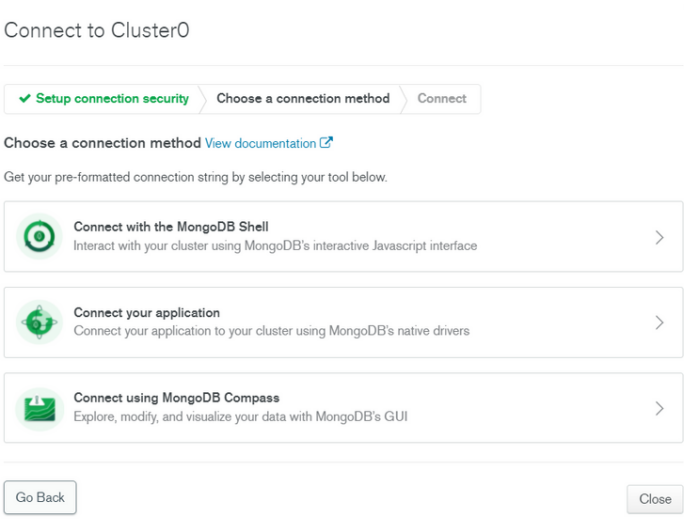
Navigate to the network access tab and select "Add IP address.



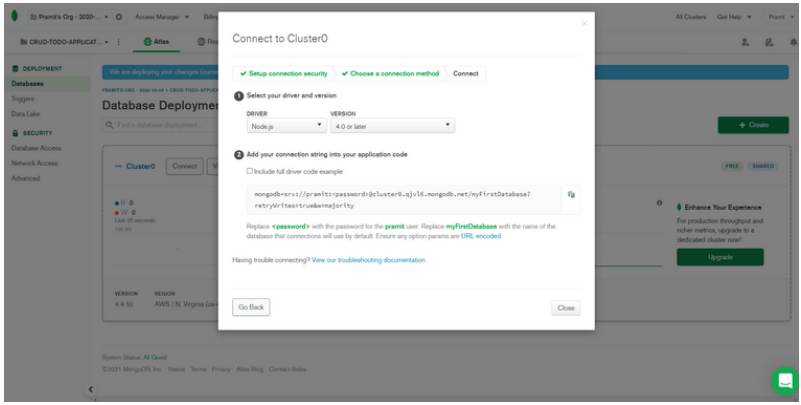
In the database, create a user. You'll need the username and password for the MongoDB URI and finally, create a database user.

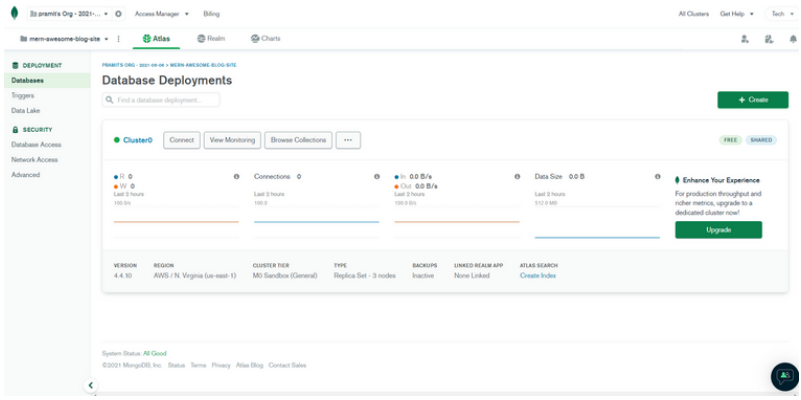


Now, select the Choose a connection method.



Connect your application by clicking on it and finally select the correct driver and version.





Now, inside server.js create a new variable and name it DB_CONNECTION. Inside it, create a string and simply paste the copied mongo DB connection URL. Now, inside it, enter your username and password, making sure to remove all the brackets and enter your own credentials. We'll secure the credential later by creating environmental variables, but for now, let's add it this way. The second thing we need is a PORT, so simply enter the port number, for now, 6000, and finally, we will use mongoose to connect to our database, so enter mongoose. connect() which is a function with two different parameters. The first will be the DB CONNECTION, and the second will be an object with two different options. The first is useNewUrlParser, which we will set to true, and the second is useUnifiedTopology, which we will also set to true. These objects are not required, but we will see some errors or warnings on our console. Following that, let's chain a.then() and.catch() because this will return a promise, so inside .then() will call the app and invoke listen, which has two parameters, the first of which is PORT and the second of which is the callback function that will be executed if our application is successfully connected and finally, if the connection to the database is not successful we will simply console log our error message.

```
//server.js
import express from "express";
import bodyParser from "body-parser";
import mongoose from "mongoose";
import cors from "cors";
import dotenv from "dotenv";

dotenv.config();

const app = express();

app.use(bodyParser.json({ limit: "50mb", extended: true
}));
app.use(bodyParser.urlencoded({ limit: "50mb", extended:
true }));
app.use(cors());

const DB_CONNECTION = process.env.DATABASE_URL;
const PORT = process.env.PORT || 6000;

mongoose
  .connect(DB_CONNECTION, { useNewUrlParser: true,
useUnifiedTopology: true })
  .then(() =>
    app.listen(PORT, () =>
      console.log(`Server is running @ :
http://localhost:${PORT}`)
    )
  )
  .catch((error) => console.error(error));
```

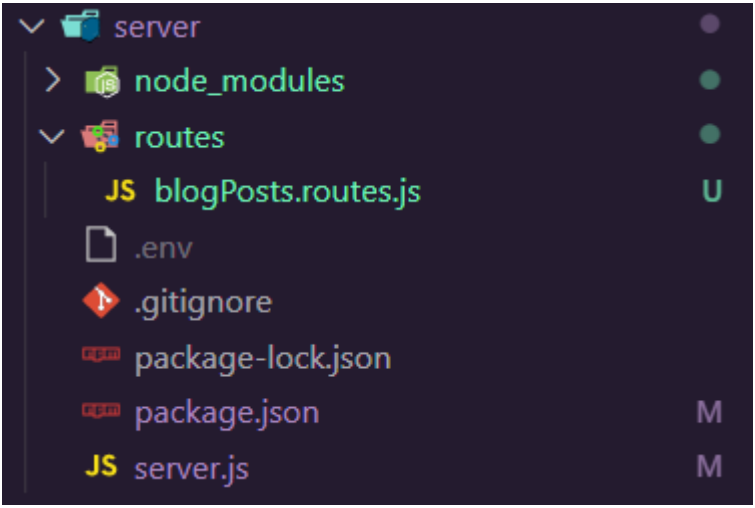
Insert mongodb+srv into the .env **file.**

```
PORT=4000
DATABASE_URL=mongodb+srv://admin:<password>@cluster0.ddtsa.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
```

That's it; we've successfully linked our server to the database.

Now that we've successfully connected to our database, let's get started on creating our routes for our backend application. To do so, we'll need to create a new folder inside the server called routes Within the routes folder, we will create a js file called blogPosts.routes.js.

This is what your folder structure should look like.



We are going to add all of the routes inside of blogPosts.routes.js, so first we must import express from "express" and also configure our router. Now we can begin adding our routes to it.

```
// routes/blogPosts.routes.js
import express from "express";

const router = express.Router();

router.get("/", (req, res) => {
  res.send("Awesome MERN BLOG");
});

export default router;
```

Let's get started on your server. js file and import the blogPost route. Now we can use express middleware to connect this blogPost to our application

```
// server.js
import express from "express";
import bodyParser from "body-parser";
import mongoose from "mongoose";
import cors from "cors";
import dotenv from "dotenv";

import blogPosts from "./routes/blogPosts.js";

dotenv.config();

const app = express();
```

```
app.use(bodyParser.json({ limit: "50mb", extended: true }));
app.use(bodyParser.urlencoded({ limit: "50mb", extended: true }));
app.use(cors());

// remember to add this after cors
app.use("/api/blogs", blogPosts);

const DB_CONNECTION = process.env.DATABASE_URL;
const PORT = process.env.PORT || 6000;

mongoose
  .connect(DB_CONNECTION, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() =>
    app.listen(PORT, () =>
      console.log(`Server is running at: http://localhost:${PORT}`)
    )
  )
  .catch((error) => console.log(error));
```

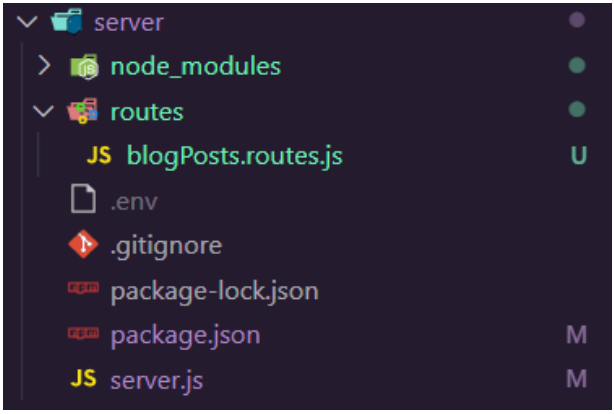
Before we go any further, let's create a folder structure for our backend applications that will allow them to be much more scalable. So let’s create a new folder called controllers inside the controllers’ folder we are also going to create a file called `blogPosts.controller.js` .So controllers is simply a file that contains route-specific logic.

so your blogPosts.routes.js and blogPosts.controller.js should resemble something like this.

```
//routes/blogPosts.routes.js
import express from 'express';

import { getAllBlogPosts } from
'../controllers/blogPosts.controller.js';

const router = express.Router();

router.get('/', getAllBlogPosts);

export default router;
blogPosts.controller.js

//controllers/blogPosts.controller.js
import express from "express";
import mongoose from "mongoose";

const router = express.Router();

export const getAllBlogPosts = (req, res) => {
  res.send("Awesome MERN BLOG");
};

export default router;
```

Let's make a new model for our blog posts, so make a folder called models and a file called blogs.js inside it.

The folder structure should resemble something like this




```
// models/blogs.js
import mongoose from "mongoose";

const blogSchema = mongoose.Schema({
  title: String,
  description: String,
  tags: [String],
  fileUpload: String,
  upvote: {
    type: Number,
    default: 0,
  },
  creator: String,
  createdAt: {
    type: Date,
    default: new Date(),
  },
});

var BlogPost = mongoose.model("BlogArticle", blogSchema);

export default BlogPost;
```

Let's start adding more routes now that our model is complete.

```
// routes/blogPosts.routes.js
import express from "express";

import {
  getAllBlogPosts,
  addBlogPost,
  getSinglePost,
  updateSingleBlogPost,
  removeSingleBlogPost,
  likeBlogPost,
} from "../controllers/blogPosts.controller.js";

const router = express.Router();

router.get("/", getAllBlogPosts);
router.post("/", addBlogPost);
router.get("/:id", getSinglePost);
router.patch("/:id", updateSingleBlogPost);
router.delete("/:id", removeSingleBlogPost);
router.patch("/:id/likeedBlogPost", likeBlogPost);

export default router;
```

Now, inside the controller's folder, add the following code to your blogPosts.controllers.js file.

getAllBlogPosts method fetches all the blogs information.

```
export const getAllBlogPosts = async (req, res) => {
  try {
    const blogPosts = await BlogPost.find();
    res.status(200).json(blogPosts);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
};
```

addBlogPost method adds/insert only one blog

```
export const addBlogPost = async (req, res) => {
  const { title, description, fileUpload, creator, tags } =
req.body;
  const createNewPost = new BlogPost({
    title,
    description,
    fileUpload,
    creator,
    tags,
  });
  try {
    await createNewPost.save();
    res.status(201).json(createNewPost);
  } catch (error) {
    res.status(409).json({ message: error.message });
  }
};
```

getSinglePost method fetches single blog post

```
export const getSinglePost = async (req, res) => {
  const { id } = req.params;

  try {
    const singlepost = await BlogPost.findById(id);

    res.status(200).json(singlepost);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
};
```

updateSingleBlogPost method updates single blog posts

```
export const updateSingleBlogPost = async (req, res) => {
  const { id } = req.params;
  const { title, description, creator, fileUpload, tags } =
req.body;
  if (!mongoose.Types.ObjectId.isValid(id))
    return res.status(404).send(`post ${id} not found`);
  const updatedBlogPost = {
    creator,
    title,
    description,
    tags,
    fileUpload,
    _id: id,
  };
  await BlogPost.findByIdAndUpdate(id, updatedBlogPost, { new:
true });
  res.json(updatedBlogPost);
};
```

removeSingleBlogPost method deletes single blog posts

```
export const removeSingleBlogPost = (req, res) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id))
    return res.status(404).send(`post ${id} not found`);

  await BlogPost.findByIdAndRemove(id);

  res.json({ message: "Successfully deleted" });
};
```

likeBlogPost method upvotes the posts

```
export const likeBlogPost = async (req, res) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id))
    return res.status(404).send(`No post with id: ${id}`);

  const post = await BlogPost.findById(id);

  const updatedBlogPost = await BlogPost.findByIdAndUpdate(
    id,
    { upvote: post.upvote + 1 },
    { new: true }
  );

  res.json(updatedBlogPost);
};
```

Your blogPosts.controller.js should resemble something like this

```
// blogPosts.controller.js
import express from "express";
import mongoose from "mongoose";

import BlogPost from "../models/blogs.js";

const router = express.Router();

export const getAllBlogPosts = async (req, res) => {
  try {
    const blogPosts = await BlogPost.find();
    res.status(200).json(blogPosts);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
};

export const addBlogPost = async (req, res) => {
  const { title, description, fileUpload, creator, tags } = req.body;

  const createNewPost = new BlogPost({
    title,
    description,
    fileUpload,
    creator,
    tags,
  });

  try {
    await createNewPost.save();
    res.status(201).json(createNewPost);
  } catch (error) {
    res.status(409).json({ message: error.message });
  }
};
```

```
export const getSinglePost = async (req, res) => {
  const { id } = req.params;

  try {
    const singlepost = await BlogPost.findById(id);

    res.status(200).json(singlepost);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
};

export const updateSingleBlogPost = async (req, res) => {
  const { id } = req.params;
  const { title, description, creator, fileUpload, tags } = req.body;

  if (!mongoose.Types.ObjectId.isValid(id))
    return res.status(404).send(`post ${id} not found`);

  const updatedBlogPost = {
    creator,
    title,
    description,
    tags,
    fileUpload,
    _id: id,
  };
  await BlogPost.findByIdAndUpdate(id, updatedBlogPost, { new: true
});
  res.json(updatedBlogPost);
};

export const likeBlogPost = async (req, res) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id))
    return res.status(404).send(`No post with id: ${id}`);

  const post = await BlogPost.findById(id);
```

```
  const updatedBlogPost = await BlogPost.findByIdAndUpdate(
    id,
    { upvote: post.upvote + 1 },
    { new: true }
  );

  res.json(updatedBlogPost);
};

export const removeSingleBlogPost = async (req, res) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id))
    return res.status(404).send(`post ${id} not found`);

  await BlogPost.findByIdAndRemove(id);

  res.json({ message: "Successfully deleted" });
};

export default router;
```

After restarting the server, you should see something similar to this:

```
PS D:\AviyelDemo\mern-awesome-blog\mern-awesome-blog\server> npm start

> server@1.0.0 start
> nodemon server.js

[nodemon] 2.0.14
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server is running at: http://localhost:4000
```

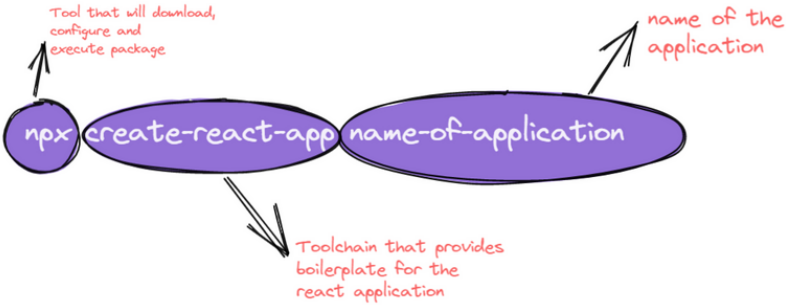
Configuring our Frontend

We'll start by setting up our frontend first using create-react-app .We will be creating the UI and its functionalities from the ground up. Let's get started on our application now.

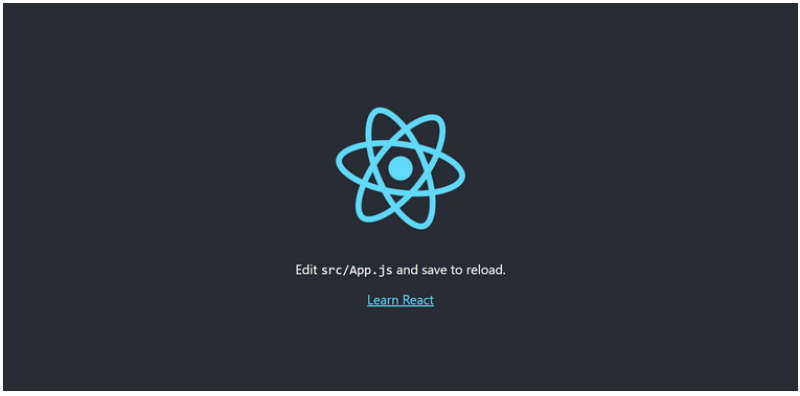
Installing react application

Let us begin with the frontend part and craft it using react. So, if Node.js isn't already installed on your system, the first thing you should do is install it. So, go to the official Node.js website and install the correct and appropriate version. We need node js so that we can use the node package manager, also known as NPM.

Now, open client folder inside the code editor of your choice. For this tutorial, I will be using VScode. Next step, let’s open the integrated terminal using ctrl + ` and type npx create-react-app . this command will create the app inside the current directory and that application will be named as client

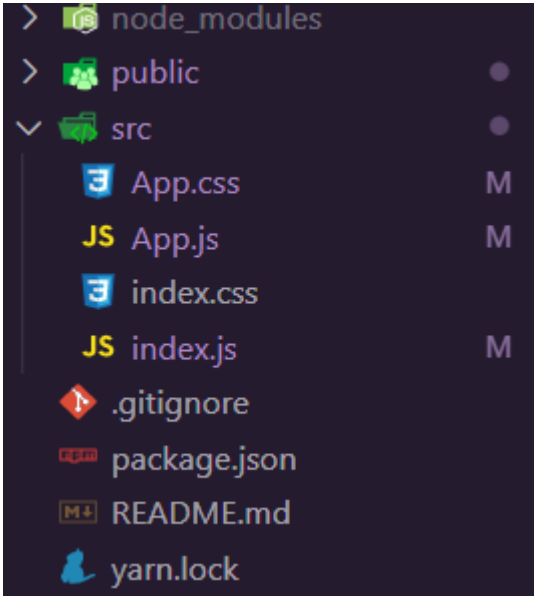


It usually takes only a few minutes to install. Normally, we would use npm to download packages into the project, but in this case, we are using npx, the package runner, which will download and configure everything for us so that we can start with an amazing template. It's now time to start our development server, so simply type npm start, and the browser will automatically open react-app.



React boilerplate cleanup

Before we begin building our projects, we must first clean them up by removing some of the files provided by create-react-app. Your src files should look like this after you've cleaned them up.

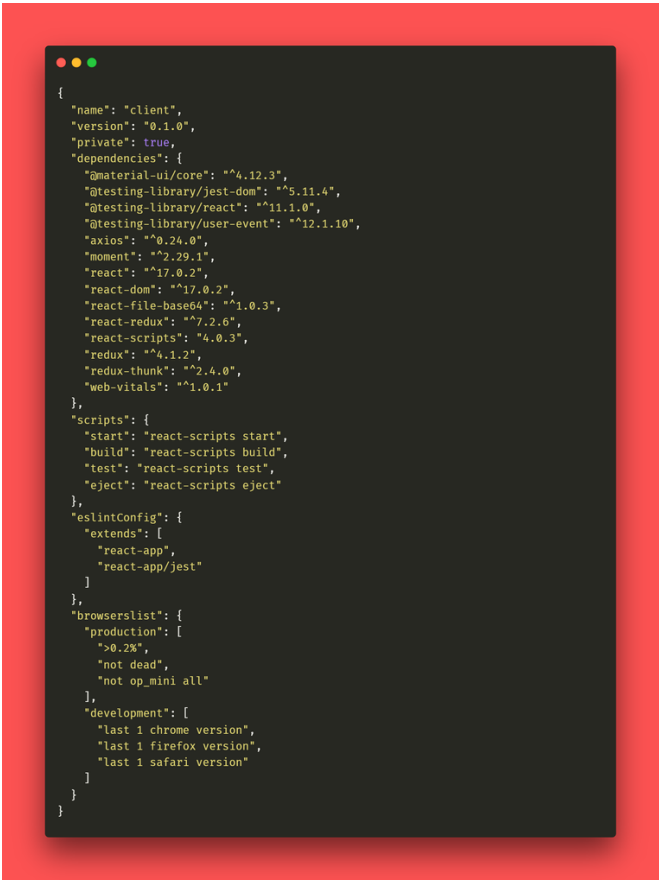


Installing some packages

We will need to install a few third-party packages for this project. so copy and paste the following command into your terminal

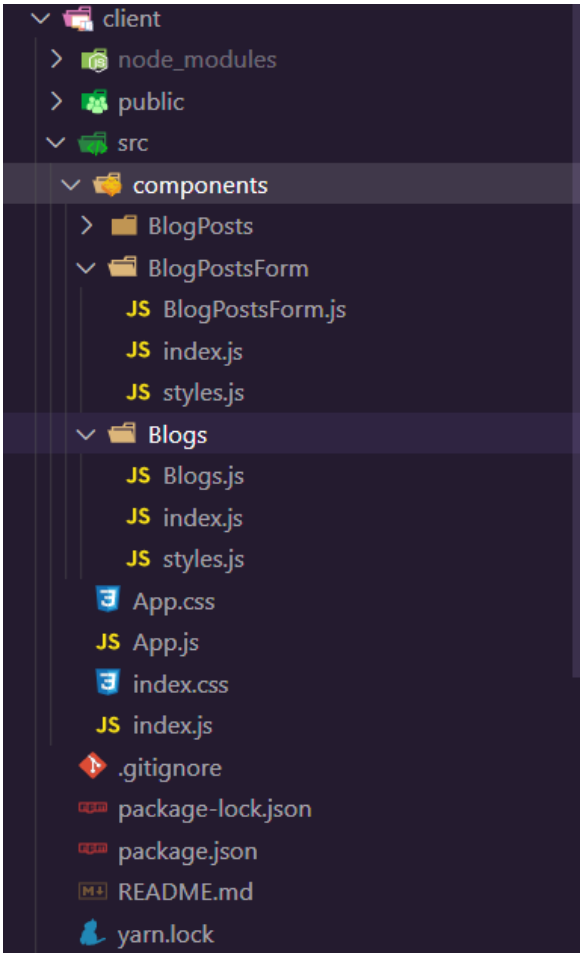
npm install @material-ui/core axios moment react-file-base64 redux react-redux redux-thunk

After installing all these packages your packge.json file should look like this:



```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@material-ui/core": "^4.12.3",
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "axios": "^0.24.0",
    "moment": "^2.29.1",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-file-base64": "^1.0.3",
    "react-redux": "^7.2.6",
    "react-scripts": "4.0.3",
    "redux": "^4.1.2",
    "redux-thunk": "^2.4.0",
    "web-vitals": "^1.0.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

After we've installed all of our project's dependencies, let's add two components to it and call them Blogs, BlogPosts and BlogPostsForm.



Now that we've got everything set up, let's go over to our App.js file and start writing some code but before that lets create one Assets folder inside our src folder and add the logo image of yor choice. After that, make another folder called styles, and inside it, make a file called app.styles.js, and paste the following code inside it.

```
// src/styles/app.styles.js
import { makeStyles } from "@material-ui/core/styles";
export default makeStyles(() => ({
  navigationBar: {
    borderRadius: 10,
    margin: "6px 0px",
    display: "flex",
    flexDirection: "row",
    justifyContent: "center",
    alignItems: "center",
  },
  title: {
    color: "#8661d1",
    fontFamily: "Poppins",
    fontStyle: "bold",
  },
  image: {
    marginRight: "25px",
  },
}));
```

Finally, go to App.js and import all of the necessary component files, styles, and components from the core material ui library, then implement it as follows.

```
//App.js
import React, { useState, useEffect } from "react";
import "./App.css";
import { Container, AppBar, Typography, Grow, Grid } from
"@material-ui/core";
import blogLogo from "./Assets/blogLogo.gif";
import BlogPosts from "./components/BlogPosts";
import BlogPostsForm from "./components/BlogPostsForm";
import useStyles from "./styles/app.styles.js";

function App() {
  const appStyles = useStyles();

  return (
    <div className="App">
      <Container maxWidth="xl">
        <AppBar
          className={appStyles.navigationBar}
          position="static"
          color="inherit"
        >
          <img
            className={appStyles.image}
            src={blogLogo}
            alt="icon"
            height="100"
          />
          <Typography className={appStyles.title} variant="h4"
align="center">
            Mern awesome blog
          </Typography>
        </AppBar>
        <Grow in>
          <Container>
            <Grid
              container
```

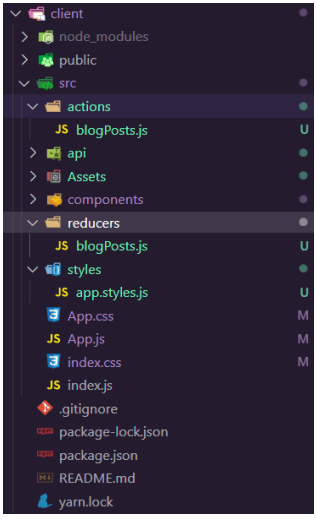
```
              justify="space-between"
              alignItems="stretch"
              spacing={2}
            >
              <Grid item xs={12} sm={7}>
                <BlogPostsForm />
              </Grid>
              <Grid item xs={12} sm={4}>
                <BlogPosts />
              </Grid>
            </Grid>
          </Container>
        </Grow>
      </Container>
    </div>
  );
}

export default App;
```

Now let’s finally connect our frontend with the backend. so for that let’s create a folder name API and inside it create a file name api.js. So let's import Axios to make API calls, then specify our backend server URL and write a function that simply fetches the post using Axios.

```
import axios from "axios";

const url = "http://localhost:4000/api/blogs";

export const fetchAllBlogPosts = () => axios.get(url);
```

Now, let's focus on adding redux functionality to our react application because all of our backend actions will be done with redux, so we need to dispatch those actions. To do that, let's create some files and folders to structure it so that our application can be scalable.So, inside our src folder, create a folder called actions as well as a folder called reducers, and inside both of those folders, create a file called blogPosts.js

Your folder structure should resemble something like this.



Before we proceed, let's fix our index.js file so that we can begin using redux from within it. Inside that file, let's import provider, which will keep track of the store, which is the global state, and which will allow us to access the store from anywhere within the application so that we don't have to be on the parent or even the child component so that we can easily access that state from anywhere, and after that, let's import createStore, applyMiddleware, and compose from the redux package, and finally, let's import thunk from redux-thunk and let's setup our index.js file accordingly.

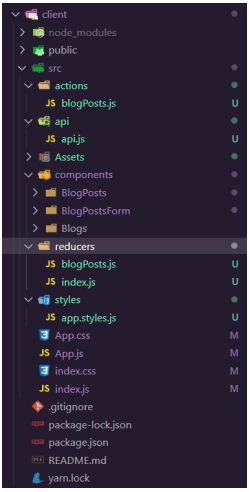
```
//index.js
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import { createStore, applyMiddleware, compose } from
"redux";
import thunk from "redux-thunk";
import { reducers } from "./reducers/blogPosts.js";
import App from "./App";
import "./index.css";

const store = createStore(reducers,
compose(applyMiddleware(thunk)));

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

If you run your application now, you may encounter the module not found error.

```
./src/index.js
Module not found: Can't resolve './reducers' in 'D:\AviyelDemo\mern-awesome-blog\mern-awesome-blog\client\src'
```

Let's go to our reducer folder and fix that error, so let's create an index.js file inside it and import combineReducers from the redux package, export and call that combineReducers as a function, and put an object inside of it. Now we can imlement all of the individual reducers that we have in this application case, which is only going to have blogPosts.

```
// reducers/index.js
import { combineReducers } from "redux";
import blogPosts from "./blogPosts";
export const reducers = combineReducers({ blogPosts });
```



If everything went well, your application should be running smoothly at this point. We now need to dispatch our action within our main App.js file.

```
// App.js
import React, { useEffect } from "react";
import "./App.css";
import { Container, AppBar, Typography, Grow, Grid } from
"@material-ui/core";
import blogLogo from "./Assets/blogLogo.gif";
import Blogs from "./components/Blogs";
import BlogPostsForm from "./components/BlogPostsForm";
import useStyles from "./styles/app.styles.js";
import { useDispatch } from "react-redux";
import { fetchAllBlogPosts } from "./actions/blogPosts";

function App() {
  const dispatch = useDispatch();
  const appStyles = useStyles();

  useEffect(() => {
    dispatch(fetchAllBlogPosts());
  }, [dispatch]);

  return (
    <div className="App">
      <Container maxWidth="xl">
        <AppBar
          className={appStyles.navigationBar}
          position="static"
          color="inherit"
        >
          <img
            className={appStyles.image}
            src={blogLogo}
            alt="icon"
            height="100"
          />
          <Typography className={appStyles.title} variant="h2"
```

```
align="center">
            Mern awesome blog
          </Typography>
        </AppBar>
        <Grow in>
          <Grid
            container
            justifyContent="space-between"
            alignItems="stretch"
            spacing={2}
          >
            <Grid item xs={12} sm={3}>
              <BlogPostsForm />
            </Grid>
            <Grid item xs={12} sm={9}>
              <Blogs />
            </Grid>
          </Grid>
        </Grow>
      </Container>
    </div>
  );
}

export default App;
```

Now, let's go over to our actions and import our api, and then create some Action creators, which are simply functions that return actions, so let's actually implement redux to pass or dispatch the function from data from our backend .

```
// actions/blogPosts.js
import * as api from "../api/api.js";

export const fetchAllBlogPosts = () => async (dispatch) => {
  try {
    const { data } = await api.fetchAllBlogPosts();

    dispatch({ type: GET_ALL_BLOGS, payload: data });
  } catch (error) {
    console.log(error.message);
  }
};
```

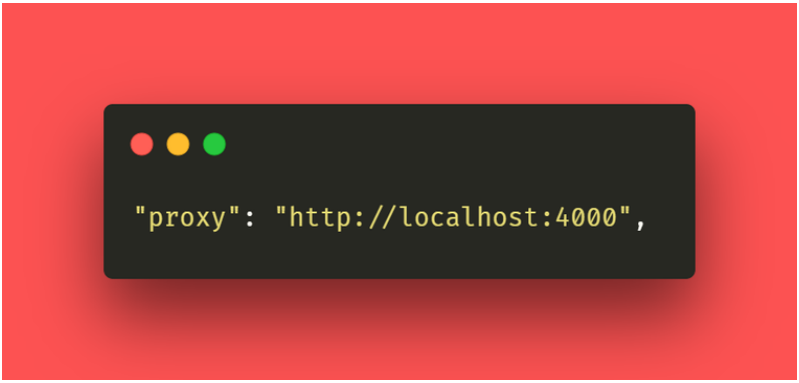
Finally, let us return to our reducers and handle the logic of getting and fetching all of the blog posts.

```
// reducers/blogPosts.js
export default (posts = [], action) => {
  switch (action.type) {
    case "GET_ALL_BLOGS":
      return action.payload;
    default:
      return posts;
  }
};
```

Now, let's actually retrieve these data from our child components, so let's go to our Blogs component and fetch the data from the global redux store we can do that by the help of useSelector

```
//components/Blogs
import React from "react";
import { Grid, CircularProgress } from
"@material-ui/core";
import { useSelector } from "react-redux";

import BlogPosts from "../BlogPosts";
import useStyles from "./styles";

const Blogs = () => {
  const posts = useSelector((state) =>
state.blogPosts);
  const classes = useStyles();

  console.log("this is post", posts);

  return (
    <>
      <BlogPosts />
    </>
  );
};

export default Blogs;
```

When you run your app, you may see an empty array and a network error; to fix this, simply include a proxy in your package.json file



So, if you still see that empty array, it means that the data was successfully fetched, and it is now time to implement the form so that we can make a post request to our database and actually add new posts to it.

So for that lets head over to the BlogPostsForm component and create a form. First step lets import all the component from material ui core library that we are going to implement in our form

```
// BlogPostsForm.js
import React, { useState, useEffect } from "react";
import { Paper, TextField, Typography, Button } from
"@material-ui/core";
import { useDispatch, useSelector } from "react-redux";
import FileBase from "react-file-base64";

import useStyles from "./styles";
import { addBlogPosts, editBlogPosts } from
"../../actions/blogPosts";

const BlogPostsForm = ({ blogPostId, setBlogPostId }) => {
  const [blogInfo, setBlogInfo] = useState({
    creator: "",
    title: "",
    description: "",
    tags: "",
    fileUpload: "",
  });
  const post = useSelector((state) =>
    blogPostId
      ? state.posts.find((message) => message._id ===
blogPostId)
      : null
  );
  const dispatch = useDispatch();
  const blogPostsStyles = useStyles();

  useEffect(() => {
    if (post) setBlogInfo(post);
  }, [post]);

  const handleSubmit = async (e) => {
    e.preventDefault();

    if (blogPostId === 0) {
```

```
      dispatch(addBlogPosts(blogInfo));
    } else {
      dispatch(editBlogPosts(blogInfo));
    }
  };

  return (
    <Paper className={blogPostsStyles.paper}>
      <form
        autoComplete="on"
        noValidate
        className={`${blogPostsStyles.root}
${blogPostsStyles.form}`}
        onSubmit={handleSubmit}
      >
        <Typography variant="h5">
          {blogPostId ? `Update "${post.title}"` : "✨ Create a
blog ✨"}
        </Typography>

        <div className={blogPostsStyles.chooseFile}>
          <Typography> 🖼️ Upload Blog Image</Typography>
          <FileBase
            type="file"
            multiple={false}
            onDone={({ base64 }) =>
              setBlogInfo({ ...blogInfo, fileUpload: base64 })
            }
          />
        </div>
        <TextField
          name="title"
          variant="outlined"
          label="🔥 Blog Title"
          fullWidth
          value={blogInfo.title}
```

```
          onChange={(e) => setBlogInfo({ ...blogInfo, title:
e.target.value })}
        />
        <TextField
          name="description"
          variant="outlined"
          label="📙 Blog Description"
          fullWidth
          multiline
          rows={7}
          value={blogInfo.description}
          onChange={(e) =>
            setBlogInfo({ ...blogInfo, description:
e.target.value })
          }
        />
        <TextField
          name="creator"
          variant="outlined"
          label="✍️ Author name"
          fullWidth
          value={blogInfo.creator}
          onChange={(e) =>
            setBlogInfo({ ...blogInfo, creator: e.target.value
})
          }
        />
        <Typography>Tags (5 max seperated by
comma)</Typography>
        <TextField
          name="tags"
          variant="outlined"
          label="🏷️ Tags"
          fullWidth
          value={blogInfo.tags}
          onChange={(e) =>
```

```
            setBlogInfo({ ...blogInfo, tags:
e.target.value.split(",") })
          }
        />

        <Button
          className={blogPostsStyles.publishButton}
          variant="contained"
          color="secondary"
          size="large"
          type="submit"
        >
          Publish 📝
        </Button>
      </form>
    </Paper>
  );
};

export default BlogPostsForm;
```

Also, don't forget to modify the blogPostForm styles within styles. js

```
// components/BlogPostsForm/styles.js
import { makeStyles } from "@material-ui/core/styles";

export default makeStyles((theme) => ({
  root: {
    "& .MuiTextField-root": {
      margin: theme.spacing(1),
    },
  },
  paper: {
```

```
    padding: theme.spacing(5),
  },
  chooseFile: {
    width: "95%",
    margin: "10px 0",
  },
  publishButton: {
    marginBottom: 10,
  },
  form: {
    display: "flex",
    flexWrap: "wrap",
    justifyContent: "center",
  },
}));
```

So before going any further lets fix our api first

```
// api/api.js
import axios from "axios";

const url = "http://localhost:4000/api/blogs";

export const fetchBlogPosts = () => axios.get(url);
export const addNewBlogPost = (newBlog) => axios.post(url,
newBlog);
export const editSingleBlogPost = (id, editedBlogPost) =>
  axios.patch(`${url}/${id}`, editedBlogPost);
```

After you've successfully added and exported the addNewBlogPost and editSingleBlogPost functions, let's actually implement them by creating some actions called addBlogPosts and editBlogPosts, respectively.

addBlogPosts action

```
export const addBlogPosts = (post) => async (dispatch) => {
  try {
    const { data } = await api.addNewBlogPost(post);

    dispatch({ type: "ADD_NEW_BLOG_POST", payload: data });
  } catch (error) {
    console.log(error.message);
  }
};
```

editBlogPosts action

```
export const editBlogPosts = (id, post) => async (dispatch) => {
  try {
    const { data } = await api.editSingleBlogPost(id, post);

    dispatch({ type: "EDIT_SINGLE_BLOG_POST", payload: data });
  } catch (error) {
    console.log(error.message);
  }
};
```

Your blogPosts.js actions should look something like this.

```
 // actions/blogPosts.js
import * as api from "../api/api.js";

export const fetchAllBlogPosts = () => async (dispatch) => {
  try {
    const { data } = await api.fetchBlogPosts();

    dispatch({ type: "GET_ALL_BLOG_POST", payload: data });
  } catch (error) {
    console.log(error.message);
  }
};

export const addBlogPosts = (post) => async (dispatch) => {
  try {
    const { data } = await api.addNewBlogPost(post);

    dispatch({ type: "ADD_NEW_BLOG_POST", payload: data });
  } catch (error) {
    console.log(error.message);
  }
};

export const editBlogPosts = (id, post) => async (dispatch) =>
{
  try {
    const { data } = await api.editSingleBlogPost(id, post);

    dispatch({ type: "EDIT_SINGLE_BLOG_POST", payload: data
});
  } catch (error) {
    console.log(error.message);
  }
};
```

After that, let's update the reducers section.

```
export default (posts = [], action) => {
    switch (action.type) {
        case "GET_ALL_BLOG_POST":
            return action.payload;
        case "ADD_NEW_BLOG_POST":
            return [...posts, action.payload];
        case "EDIT_SINGLE_BLOG_POST":
            return posts.map((post) =>
                post._id === action.payload._id ?
action.payload : post
            );
        default:
            return posts;
    }
};
```

Finally, let's update our App.js to include the blogPostId state, which we'll pass as a prop to our BlogPostsForm and Blogs components.

```
//App.js
import React, { useState, useEffect } from "react";
import "./App.css";
import { Container, AppBar, Typography, Grow, Grid } from
"@material-ui/core";
import blogLogo from "./Assets/blogLogo.gif";
import Blogs from "./components/Blogs";
import BlogPostsForm from "./components/BlogPostsForm";
import useStyles from "./styles/app.styles.js";
import { useDispatch } from "react-redux";
import { fetchAllBlogPosts } from "./actions/blogPosts";
```

```
function App() {
  const [blogPostId, setBlogPostId] = useState(0);
  const dispatch = useDispatch();
  const appStyles = useStyles();

  useEffect(() => {
    dispatch(fetchAllBlogPosts());
  }, [blogPostId, dispatch]);

  return (
    <div className="App">
      <Container maxWidth="xl">
        <AppBar
          className={appStyles.navigationBar}
          position="static"
          color="inherit"
        >
          <img
            className={appStyles.image}
            src={blogLogo}
            alt="icon"
            height="100"
          />
          <Typography className={appStyles.title} variant="h2"
align="center">
            Mern awesome blog
          </Typography>
        </AppBar>
        <Grow in>
          <Grid
            container
            justifyContent="space-between"
            alignItems="stretch"
            spacing={2}
          >
```

```
            <Grid item xs={12} sm={3}>
              <BlogPostsForm
                blogPostId={blogPostId}
                setBlogPostId={setBlogPostId}
              />
            </Grid>
            <Grid item xs={12} sm={9}>
              <Blogs setBlogPostId={setBlogPostId} />
            </Grid>
          </Grid>
        </Grow>
      </Container>
    </div>
  );
}

export default App;
```

After we've fixed App.js, we'll move on to our Blogs component and use the passed props within it and also drill it to the BlogPosts components

```
// components/Blogs.js
import React from "react";
import { Grid, CircularProgress } from "@material-ui/core";
import { useSelector } from "react-redux";

import BlogPosts from "../BlogPosts";
import useStyles from "./styles";

const Blogs = ({ setBlogPostId }) => {
  const posts = useSelector((state) => state.posts);
  const classes = useStyles();
  console.log("this is post", posts);
```

```
  return !posts.length ? (
    <CircularProgress />
  ) : (
    <Grid
      className={classes.container}
      container
      alignItems="stretch"
      spacing={4}
    >
      {posts.map((post) => (
        <Grid key={post._id} item xs={12} sm={12}>
          <BlogPosts post={post} setBlogPostId={setBlogPostId}
/>
        </Grid>
      ))}
    </Grid>
  );
};

export default Blogs;
```

Now that we've completed almost everything, it's time to work on the individual blog posts. To do so, go to the BlogPosts components and install material UI icons first, then import several components from the material UI core library, and finally copy and paste the following code inside it.

```
// components/BlogPosts.js
import React from "react";
import {
  Typography,
  CardMedia,
  Button,
  Card,
  CardActions,
  CardContent,
} from "@material-ui/core/";
import ArrowUpwardIcon from "@material-ui/icons/ArrowUpward";
import DeleteIcon from "@material-ui/icons/Delete";
import EditIcon from "@material-ui/icons/Edit";
import moment from "moment";
import { useDispatch } from "react-redux";
import blogImageLogo from "../../Assets/blogLogo.gif";

import { upvoteBlogPosts, removeBlogPosts } from
"../../actions/blogPosts";
import useStyles from "./styles";

const BlogPosts = ({ post, setCurrentId }) => {
  const dispatch = useDispatch();
  const blogPostStyles = useStyles();

  return (
    <>
      <Card className={blogPostStyles.blogContainer}>
        <CardMedia
          className={blogPostStyles.imageContainer}
          image={post.fileUpload || blogImageLogo}
          title={post.title}
        />{" "}
        <div className={blogPostStyles.nameOverlay}>
          <Typography variant="h6"> {post.creator}
</Typography>{" "}
```

```
          <Typography variant="body2">
            {" "}
            {moment(post.createdAt).fromNow()}{" "}
          </Typography>{" "}
        </div>{" "}
        <div className={blogPostStyles.editOverlay}>
          <Button
            style={{
              color: "white",
            }}
            size="small"
            onClick={() => setCurrentId(post._id)}
          >
            <EditIcon fontSize="default" />
          </Button>{" "}
        </div>{" "}
        <div className={blogPostStyles.tagSection}>
          <Typography variant="body2" color="textSecondary"
component="h2">
            {" "}
            {post.tags.map((tag) => `#${tag} `)}{" "}
          </Typography>{" "}
        </div>{" "}
        <Typography
          className={blogPostStyles.titleSection}
          gutterBottom
          variant="h5"
          component="h2"
        >
          {post.title}{" "}
        </Typography>{" "}
        <CardContent>
          <Typography variant="body2" color="textSecondary"
component="p">
            {" "}
            {post.description}{" "}
```

```
          </Typography>{" "}
        </CardContent>{" "}
        <CardActions className={blogPostStyles.cardActions}>
          <Button
            size="small"
            color="primary"
            onClick={() => dispatch(upvoteBlogPosts(post._id))}
          >
            <ArrowUpwardIcon fontSize="small" />
{post.likeCount}{" "}
          </Button>{" "}
          <Button
            size="small"
            color="primary"
            onClick={() => dispatch(removeBlogPosts(post._id))}
          >
            <DeleteIcon fontSize="big" />
          </Button>{" "}
        </CardActions>{" "}
      </Card>{" "}
    </>
  );
};

export default BlogPosts;
```

Finally, let's create an action to actually upvote and remove the blog post. First, create a function inside API and name it upvoteSingleBlogPost and removeBlogPost, then export it.

```
// api/api.js
import axios from "axios";

const url = "http://localhost:4000/api/blogs";

export const fetchBlogPosts = () => axios.get(url);
export const addNewBlogPost = (newBlog) => axios.post(url, newBlog);
export const editSingleBlogPost = (id, editedBlogPost) =>
    axios.patch(`${url}/${id}`, editedBlogPost);
export const upvoteSingleBlogPost = (id) =>
    axios.patch(`${url}/${id}/likedBlogPost`);
export const removeBlogPost = (id) => axios.delete(`${url}/${id}`);
```

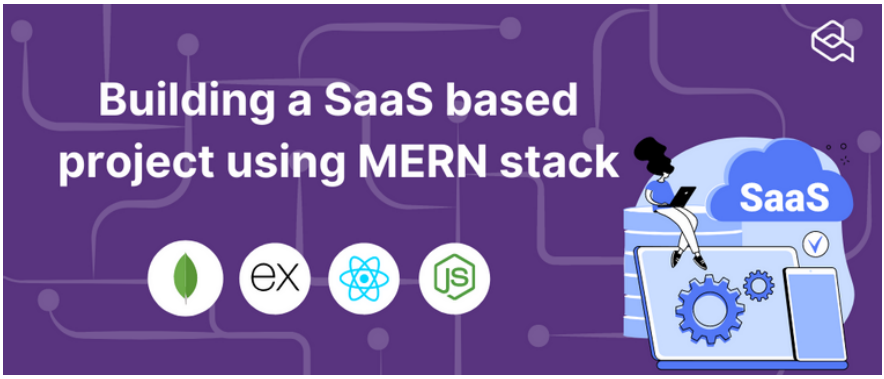
Finally, let us fix our reducers and we will be finished with our application.

```
// reducers/blogPosts.js
export default (posts = [], action) => {
    switch (action.type) {
        case "GET_ALL_BLOG_POST":
            return action.payload;
        case "ADD_NEW_BLOG_POST":
            return [...posts, action.payload];
        case "EDIT_SINGLE_BLOG_POST":
            return posts.map((post) =>
                post._id === action.payload._id ? action.payload : post
            );
        case "UPVOTE_SINGLE_BLOG_POST":
            return posts.map((post) =>
                post._id === action.payload._id ? action.payload : post
            );
        case "DELETE_SINGLE_BLOG_POST":
            return posts.filter((post) => post._id !== action.payload);
        default:
            return posts;
    }
};
```

We've gone over quite a lot of material to give you the knowledge you need to build a fully-fledged MERN stack application from absolute scratch.

The complete source code can be found here.

https://github.com/pramit-marattha/MERN-awesome-blog

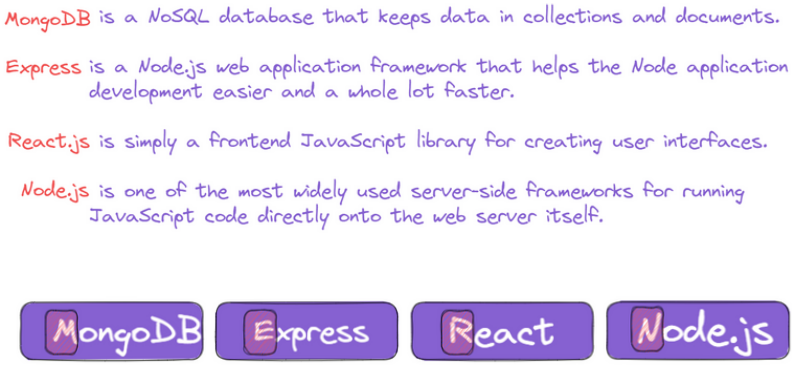


This blog article concentrates on the most significant tasks and ideas to assist you in better understanding and building MERN stack applications from the absolute ground up. It is intended for folks who are really curious about the MERN stack and want to focus on what they really need to know.

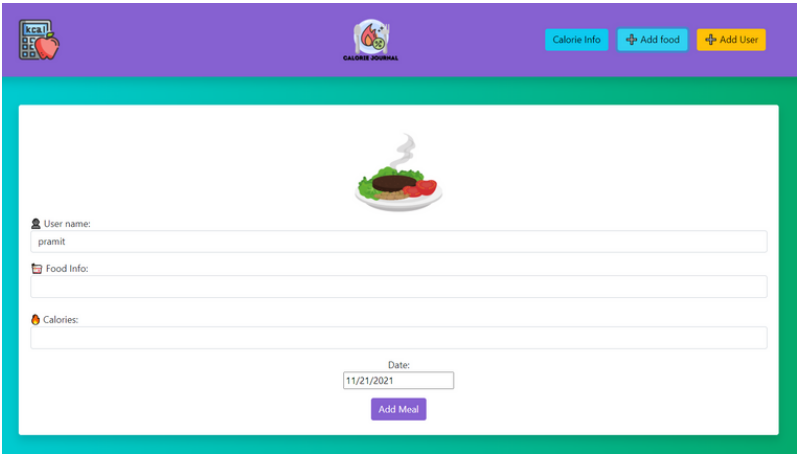
So, What is the MERN stack?

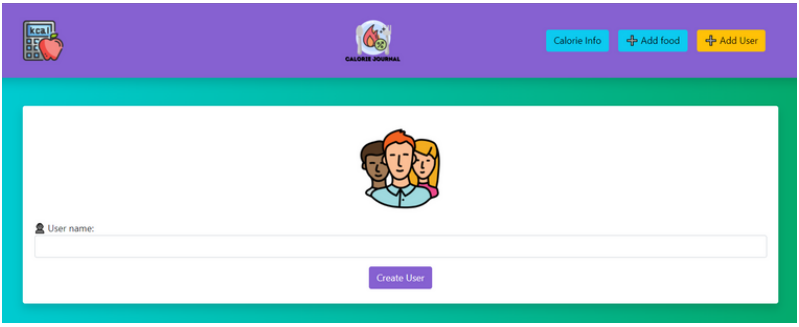
The MERN stack is a popular technology stack for creating modern Single Page Applications also known as SPA in short. MongoDB, Express, React, and Node.js are the acronyms for the “MERN” stack.MERN is a variant of the very popular MEAN stack (MongoDB, Express, Angular, Node), with React replacing Angular as the frontend UI framework. The MEVN (MongoDB, Express, Vue, Node), which uses Vue as the frontend UI framework, is another very popular option. These frontends tech stack helps for building Single Page Applications (SPAs) which helps to

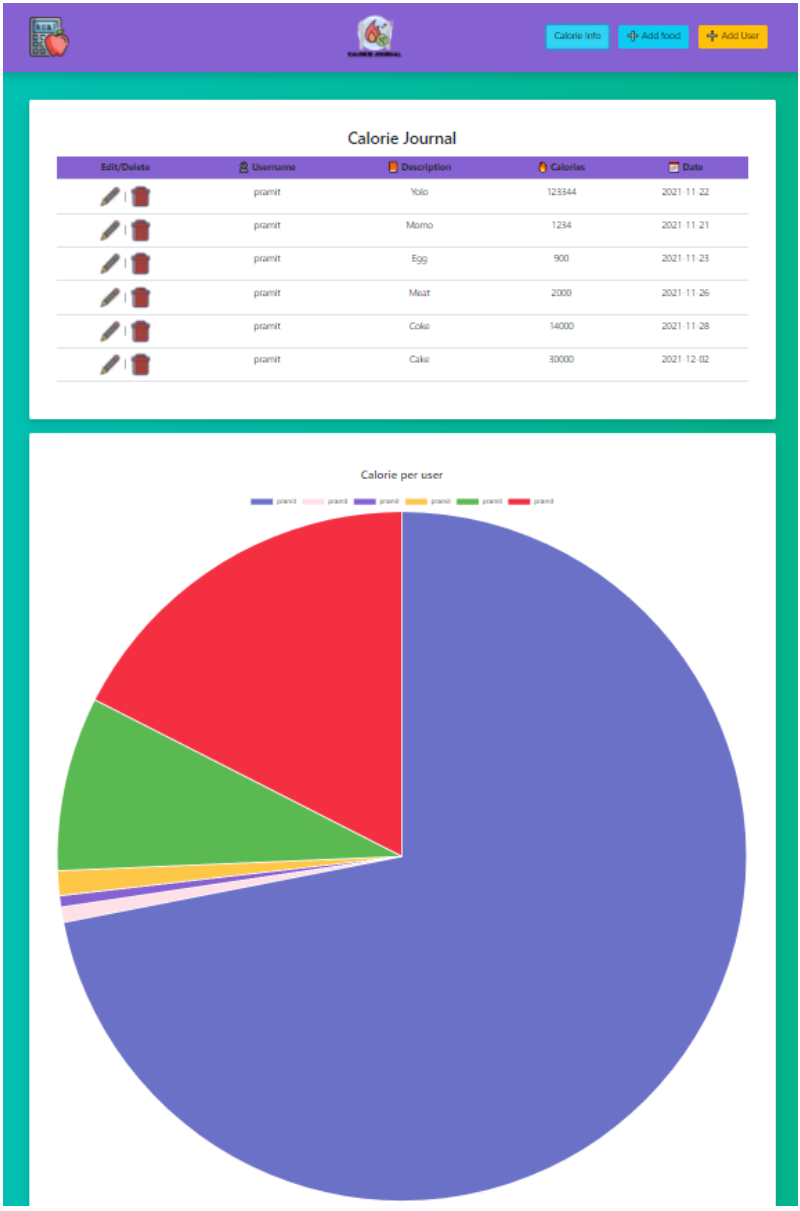
avoid reloading the entire page and only fetch relevant pieces of information of the page from the server and displays freshly and newly updated stuff.



In this blog article, we'll build a full-stack calorie tracker application that users can use to keep track of users food habits and are able to track their entire calorie count by utilizing the absolute power of the MERN stack only. This tutorial should help you understand the fundamentals as well as advanced concepts and operations of the MERN stack technology. Here is our application's final sneak peek.


CALORIE JOURNAL
Calorie Info
Add food
Add User
User name:
pramit
Food Info:
Calories:
Date:
11/21/2021
Add Meal


CALORIE JOURNAL
Calorie Info
Add food
Add User
User name:
Create User


Calorie Info
Add food
Add User
Calorie Journal
Edit/Delete
Username
Description
Calories
Date
pramit
Yolo
123344
2021-11-22
pramit
Momo
1234
2021-11-21
pramit
Egg
900
2021-11-23
pramit
Meat
2000
2021-11-26
pramit
Coke
14000
2021-11-28
pramit
Cake
30000
2021-12-02
Calorie per user

Configuring our folder structure

Create a two folder name client and server inside your project directory, then open it inside the Visual Studio Code or any code editor of your choice.

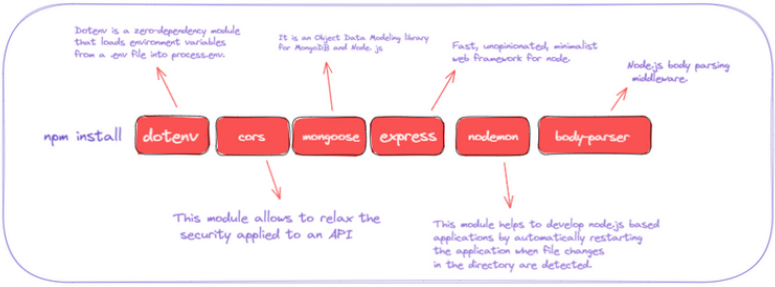
Now we'll set up our backend with npm and install the required packages, then configure a MongoDB database, set up a server with Node and Express, establish a database schema to describe our calorie tracker application, and set up API routes to create, read, update, and delete data and information from the database. So, using a command prompt, navigate to your server's directory and run the code below.

```
init -y
```

Configuring and updating our package.json file

Execute the following commands in the terminal to install the dependencies.

```
npm install cors dotenv express mongoose nodemon
body-parser
```



```
PS D:\AviyelDemo\MERN-saas\server> npm i cors dotenv express mongoose nodemon body-parser

up to date, audited 197 packages in 2s

19 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

The "**package.json**" file should look like this after the dependencies have been installed.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "start": "nodemon app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "cors": "^2.8.5",
    "dotenv": "^10.0.0",
    "express": "^4.17.1",
    "mongoose": "^6.0.13",
    "nodemon": "^2.0.15"
  }
}
```

Also, don't forget to update the scripts.

Now go to your server directory and make a app.js file there.

The structure of your folders and file should resemble this.

Setting up app.js

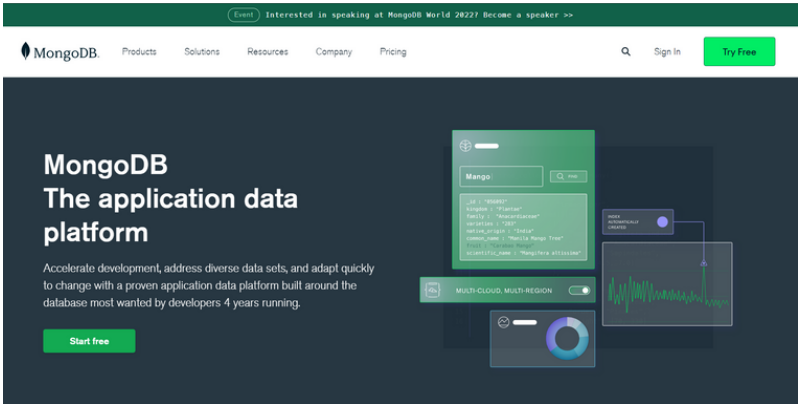
- Import express module.
- Import mongoose module
- Import and configure dotenv module
- Import CORS module
- Use express() to start our app.

```
//app.js
const express = require("express");
const cors = require("cors");
const mongoose = require("mongoose");

require("dotenv").config();

const app = express();
On that app instance, we can now use all of the different
methods. Let's start with some basic setup. Don't forget to
configure the port as well.

// app.js
const express = require("express");
const cors = require("cors");
const mongoose = require("mongoose");

require("dotenv").config();

const app = express();
const port = process.env.PORT || 5000;

app.use(cors());
app.use(express.json());
```

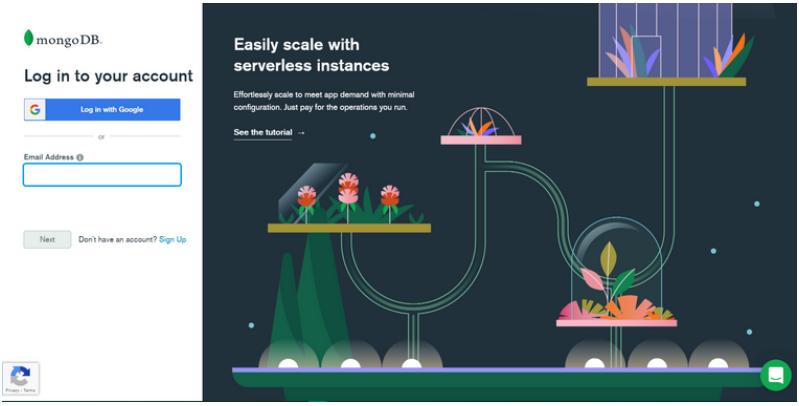
Setting up MongoDB cloud cluster

MongoDB is a document-oriented database that is open source and cross-platform. MongoDB is a NoSQL database that stores data in JSON-like documents with optional schemas. Versions prior to October 16, 2018 are released under the AGPL license. All versions released after October 16, 2018, including bug fixes for previous versions, are distributed under the SSPL license v1.

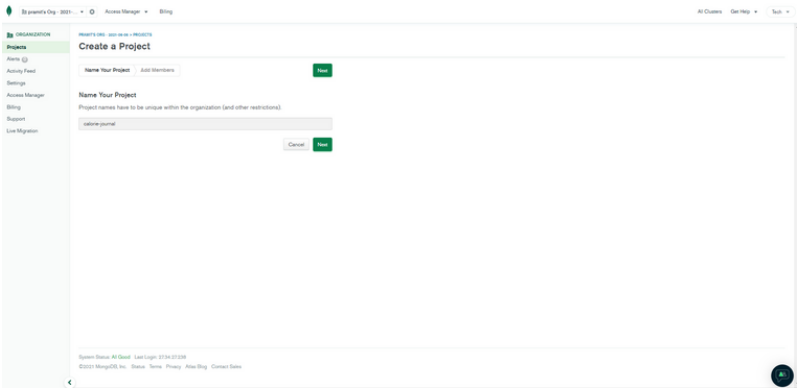
Official MongoDB website



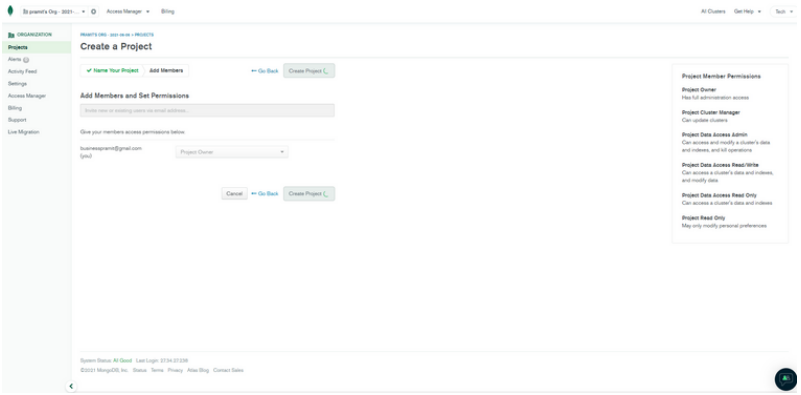
Sign in to MongoDB



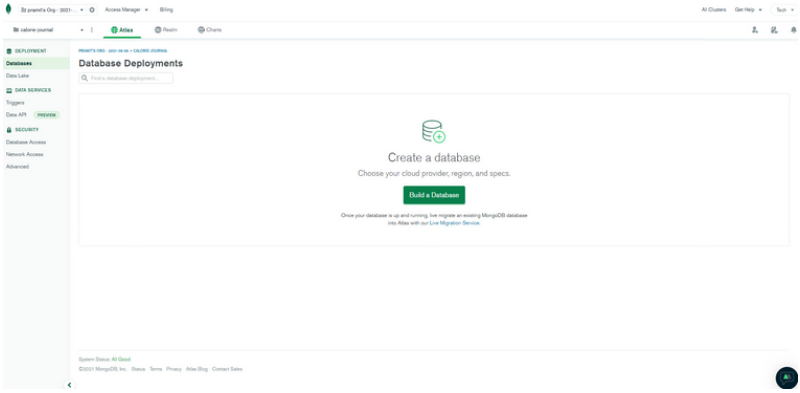
Create a Project



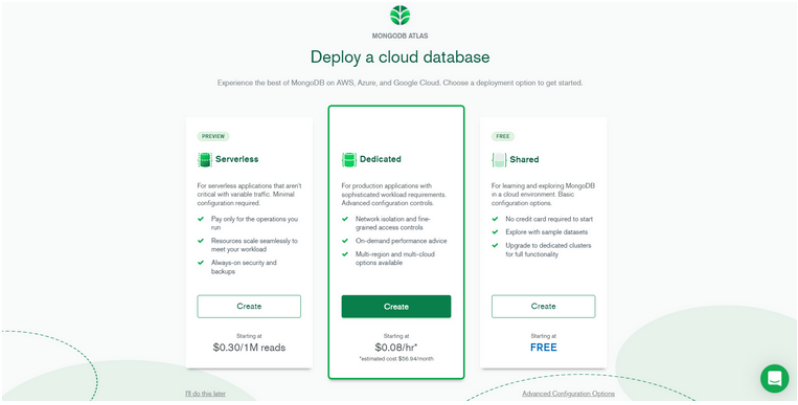
Adding members



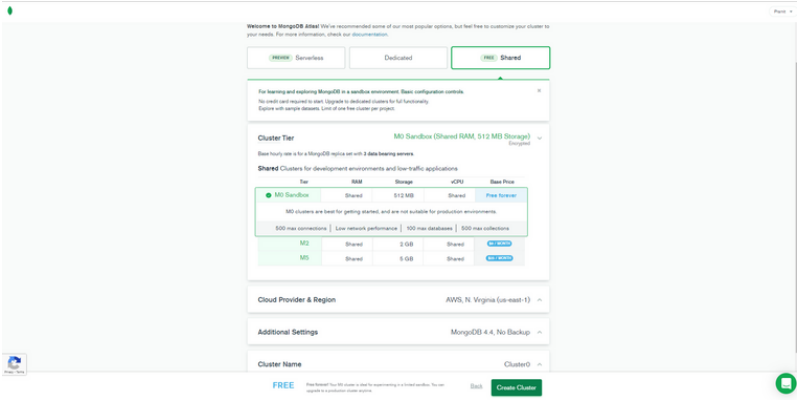
Creating a database



Creating a cluster

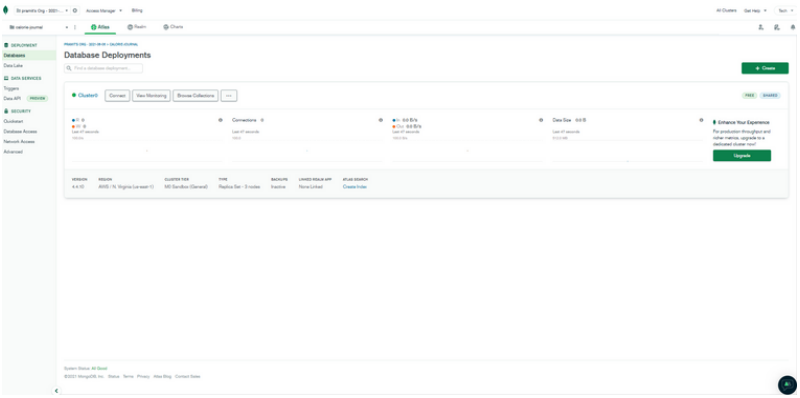


Selecting a cloud service provider

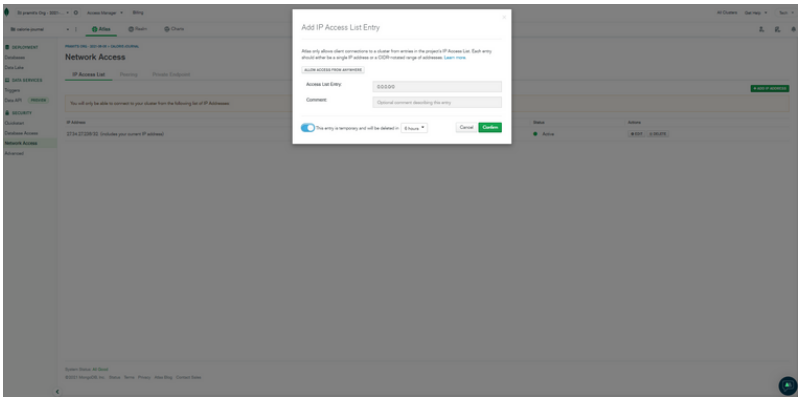


Security Quickstart

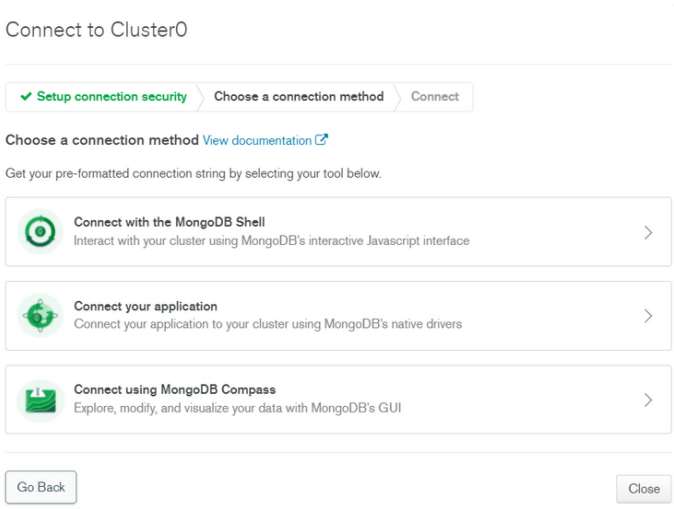
Finish and close to make a cluster and wait for the cluster to be built before proceeding (usually takes around 5 -10 minutes)



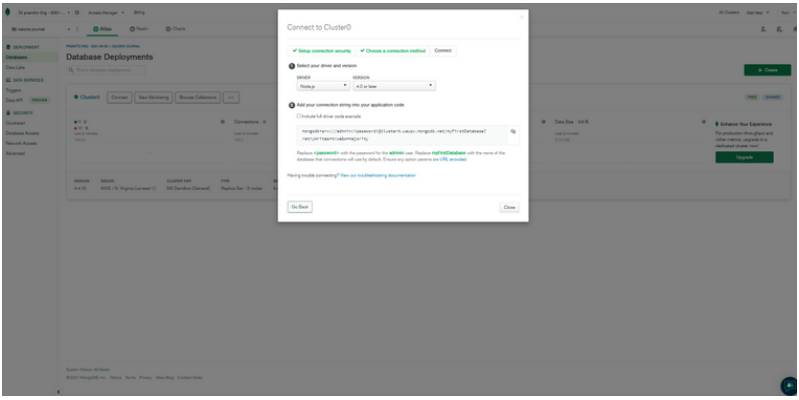
Navigate to the network access tab and select "Add IP address."



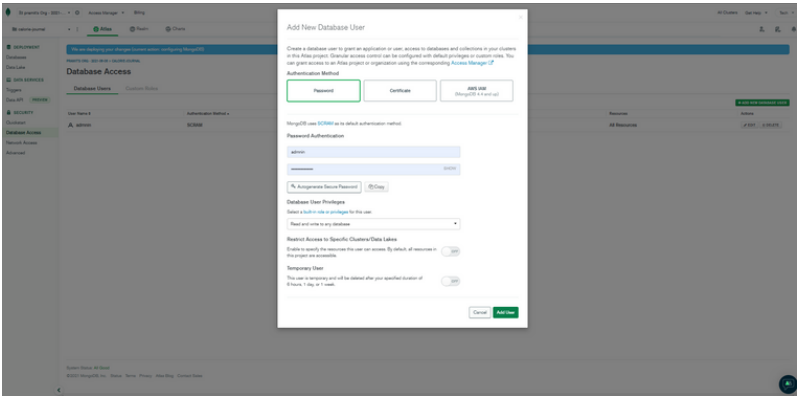
Now, select the Choose a connection method.



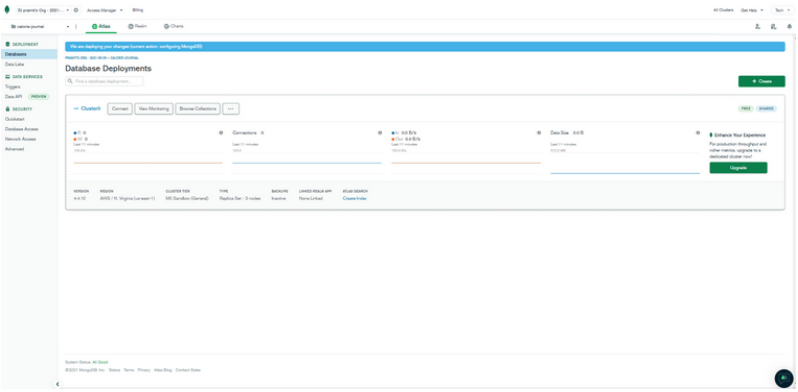
Connect your application by clicking on it and finally select the correct driver and version.



In the database, create a user. You'll need the username and password for the MongoDB URI and finally, create a database user.



Cloud atlas up and running



Now, inside app.js create a new variable and name it DATABASE_CONNECTION. Inside it, create a string and simply paste the copied mongo DB connection URL or simply paste the link for the environment variables. Now, inside the link of Mongo Sb cloud atlas URL , enter your username and password, making sure to remove all the brackets and enter your own credentials. The second thing we need is a PORT, so simply enter the port number, for now, 6000, and finally, we will use mongoose to connect to our database, so enter mongoose.connect() which is a function with two different parameters. The first will be the DATABASE_CONNECTION, and the second will be an object with two different options. The first is useNewUrlParser, which we will set to true, and the second is useUnifiedTopology, which we will also set to true. These objects are not required, but we will see some errors or warnings on our console. Following that, let's chain

a.then() and.catch() because this will return a promise, so inside .then() will call the app and invoke listen, which has two parameters, the first of which is PORT and the second of which is the callback function that will be executed if our application is successfully connected and finally, if the connection to the database is not successful we will simply console log our error message.

```
// app.js
const express = require("express");
const cors = require("cors");
const mongoose = require("mongoose");
require("dotenv").config();
const app = express();
// app config
app.use(cors());
app.use(express.json());
// port and DB config
const DATABASE_CONNECTION = process.env.DATABASE_URL;
const PORT = process.env.PORT || 5000;
// mongoose connection
mongoose
  .connect(DATABASE_CONNECTION, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() =>
    app.listen(PORT, () =>
      console.log(`Server is running at :
http://localhost:${PORT}`)
   ) )
  .catch((error) => console.error(error));
```

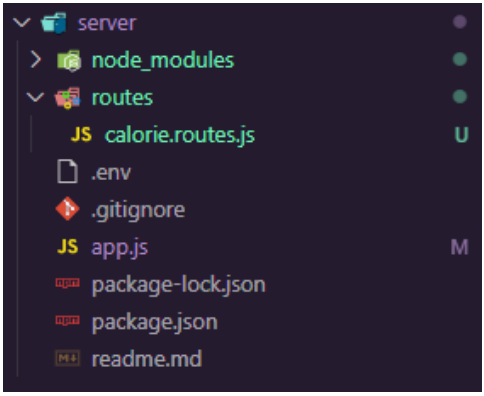
Insert mongodb+srv into the .env **file.**

```
PORT=6000
DATABASE_URL=mongodb+srv://pramit:<password>@cluster0.uauqv.mo
ngodb.net/myFirstDatabase?retryWrites=true&w=majority
```

That's all there is to it; we've successfully connected our server to the database.

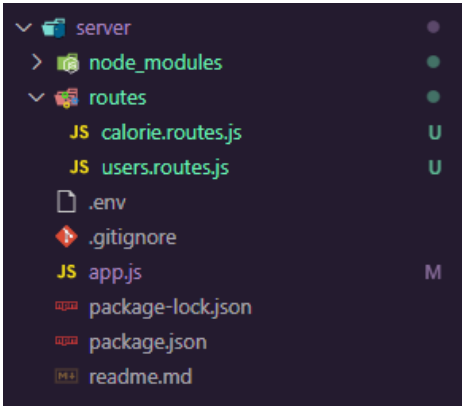
Now that we've successfully connected to our database, let's get started on building our backend application's routes. To do so, we'll need to create a new folder called routes on the server directory. We will create a file called calorie.routes.js within the routes folder.

This is how your folders should be organized.



After that, create a new route for the user by using the same steps as before, by navigating to the routes folder and creating a user.routes.js file within it.

Now, your folder structure should look something like this

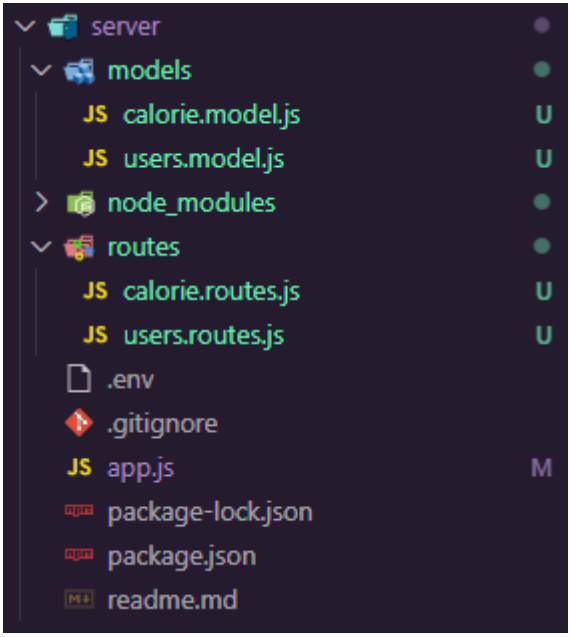


Let's get started by importing the calorie and user routes into your app.js file . We can now connect calorie and user to our application using express middleware. Finally, your app.js file should like the following.

```
//app.js
const express = require("express");
const cors = require("cors");
const mongoose = require("mongoose");

require("dotenv").config();

const app = express();

// app config
app.use(cors());
app.use(express.json());

// port and DB config
const DATABASE_CONNECTION = process.env.DATABASE_URL;
const PORT = process.env.PORT || 5000;

// mongoose connection
mongoose
    .connect(DATABASE_CONNECTION, {
        useNewUrlParser: true,
        useUnifiedTopology: true,
    })
    .then(() =>
        app.listen(PORT, () =>
            console.log(`Server is running at :
http://localhost:${PORT}`)
        )
    )
    .catch((error) => console.error(error));

// routers
const calorie = require("./routes/calorie.routes.js");
const users = require("./routes/users.routes.js");

app.use("/calorie", calorie);
app.use("/users", users);
```

We are going to add all of the routes as well as its controllers inside of calorie.routes.js and user.routes.js , so first we must import express from "express" and also configure our router. But first, let's make a model for our users and calorie.So, create a folder named models, and inside that folder, create two files called calorie.model.js and users.model.js, and paste the following code into each of them.

Now, your folder structure should look something like this



```
//models/calorie.model.js
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const calorieSchema = new Schema({
    username: {
        type: String,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    calories: {
        type: Number,
        required: true
    },
    date: {
        type: Date,
        required: true
    },
}, {
    timestamps: true,
});

const Calorie = mongoose.model("CalorieJournal",
calorieSchema);
module.exports = Calorie;
```

and

```
//models/users.model.js
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const userSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        minlength: 2,
    },
}, {
    timestamps: true,
});

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

module.exports = User;
```

Now we can begin adding our routes to it.

```
//routes/calorie.routes.js
const router = require("express").Router();
let Calorie = require("../models/calorie.model.js");

router.route("/").get((req, res) => {
    Calorie.find()
        .then((meals) => res.json(meals))
        .catch((err) => res.status(400).json("Error: " +
err));
```

```
});

router.route("/add").post((req, res) => {
    const username = req.body.username;
    const description = req.body.description;
    const calories = Number(req.body.calories);
    const date = Date.parse(req.body.date);

    const addCalorie = new Calorie({
        username,
        description,
        calories,
        date,
    });

    addCalorie
        .save()
        .then(() => res.json("Calories Added Successfully"))
        .catch((err) => res.status(400).json("Error: " +
err));
});
```

Fetching all the calorie information.

```
router.route("/:id").get((req, res) => {
    Calorie.findById(req.params.id)
        .then((calories) => res.json(calories))
        .catch((err) => res.status(400).json("Error: " + err));
});
```

Deleting single calorie information.

```
router.route("/:id").delete((req, res) => {
    Calorie.findByIdAndDelete(req.params.id)
        .then(() => res.json("Calories is deleted Successfully"))
        .catch((err) => res.status(400).json("Error: " + err));
});
```

Updating single calorie information.

```
router.route("/update/:id").post((req, res) => {
    Calorie.findById(req.params.id)
        .then((calories) => {
            calories.username = req.body.username;
            calories.description = req.body.description;
            calories.calories = Number(req.body.calories);
            calories.date = Date.parse(req.body.date);
            calories
                .save()
                .then(() => res.json("Calorie Updated Successfully"))
                .catch((err) => res.status(400).json("Err: " + err));
        })
        .catch((err) => res.status(400).json("Err: " + err));
});
```

Finally, export the router

```
module.exports = router;
```

Your calorie.route.js file should look like this.

```
//models/calorie.model.js
const router = require("express").Router();
let Calorie = require("../models/calorie.model.js");

router.route("/").get((req, res) => {
    Calorie.find()
        .then((meals) => res.json(meals))
        .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/add").post((req, res) => {
    const username = req.body.username;
    const description = req.body.description;
    const calories = Number(req.body.calories);
    const date = Date.parse(req.body.date);

    const addCalorie = new Calorie({
        username,
        description,
        calories,
        date,
    });

    addCalorie
        .save()
        .then(() => res.json("Calories Added Successfully"))
        .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/:id").get((req, res) => {
    Calorie.findById(req.params.id)
        .then((calories) => res.json(calories))
```

```
        .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/:id").delete((req, res) => {
    Calorie.findByIdAndDelete(req.params.id)
        .then(() => res.json("Calories is deleted
Successfully"))
        .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/update/:id").post((req, res) => {
    Calorie.findById(req.params.id)
        .then((calories) => {
            calories.username = req.body.username;
            calories.description = req.body.description;
            calories.calories = Number(req.body.calories);
            calories.date = Date.parse(req.body.date);
            calories
                .save()
                .then(() => res.json("Calorie Updated
Successfully"))
                .catch((err) => res.status(400).json("Err: " +
err));
        })
        .catch((err) => res.status(400).json("Err: " + err));
});

module.exports = router;
```

Now let's update user routes.

```
 //routes/user.routes.js
const router = require("express").Router();
let User = require("../models/users.model.js");
```

Fetch the user info

```
router.route("/").get((req, res) => {
    User.find()
        .then((users) => res.json(users))
        .catch((err) => res.status(400).json("Error: " + err));
});
```

Adding the user info

```
router.route("/add").post((req, res) => {
    const username = req.body.username;

    const newUser = new User({
        username
    });

    newUser
        .save()
        .then(() => res.json("User added Successfully"))
        .catch((err) => res.status(400).json("Error: " + err));
});
```

Finally, export the router

```
module.exports = router;
```

Your users.route.js file should look like this.

```
//routes/user.routes.js
const router = require("express").Router();
let User = require("../models/users.model.js");

// get user
router.route("/").get((req, res) => {
    User.find()
        .then((users) => res.json(users))
        .catch((err) => res.status(400).json("Error: " + err));
});

// add user
router.route("/add").post((req, res) => {
    const username = req.body.username;

    const newUser = new User({
        username
    });

    newUser
        .save()
        .then(() => res.json("User added Successfully"))
        .catch((err) => res.status(400).json("Error: " + err));
});

module.exports = router;
```

You should see something like this after restarting the server:

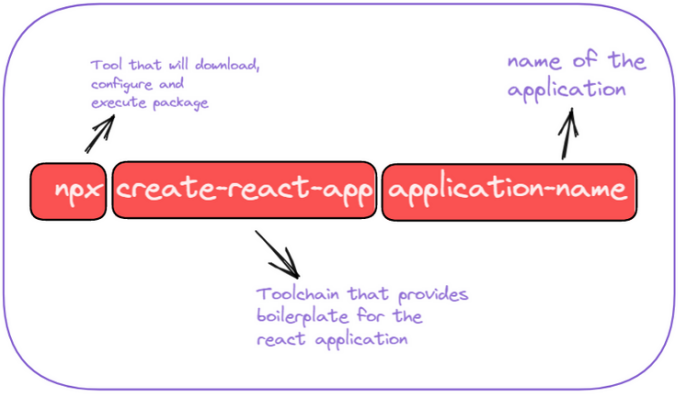
```
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Server is running at : http://localhost:5000
```

Configuring our Frontend

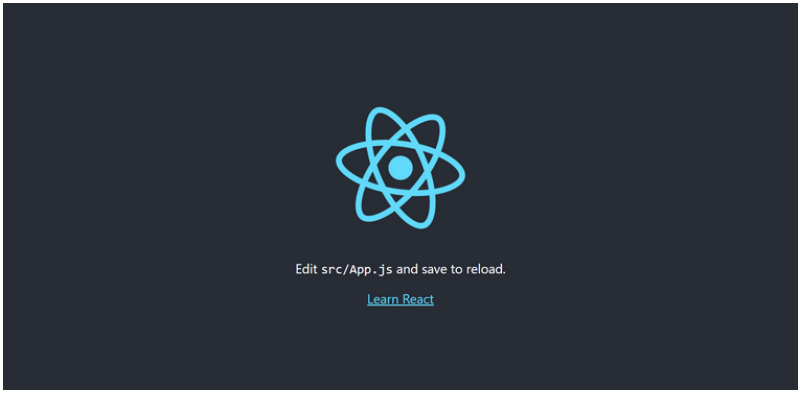
We'll begin by using create-react-app to set up our frontend. We'll build the user interface and its features from the ground up. Let's go to work on our application right away.

Setting up react application bootstarpped using CRA

Let's start with the frontend and build it with react. The first thing you need to do is install Node.js if it isn't already installed on your PC. So, head over to the official Node.js website and download the latest version. Node js is required in order to use the node package manager, generally known as NPM. Now open the client folder in your preferred code editor. I'll be using VScode . Next, open the integrated terminal and type npx create-react-app . This command will create a client application in the current directory, using the name client.

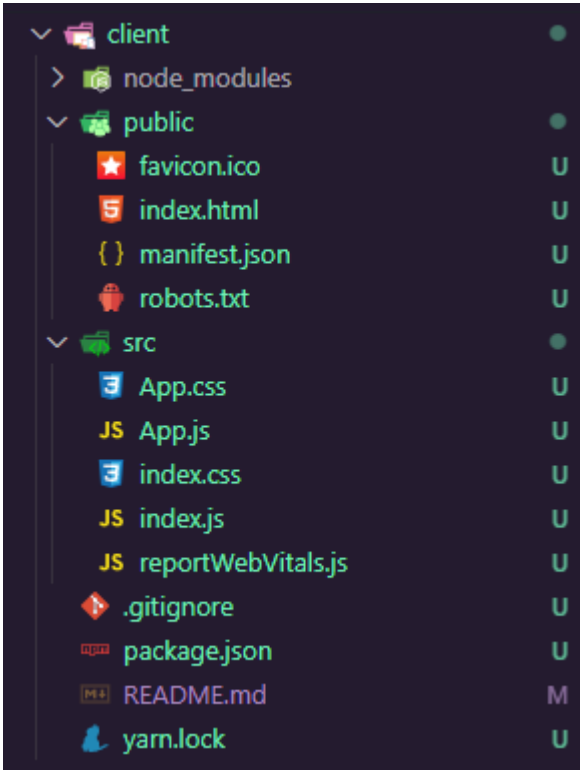


It normally only takes a few minutes to set up. Normally, we would use npm to get packages into a project, but in this case, we'll use npx, the package runner, which will download and configure everything for us so that we can get started with an excellent template right away. It's time to start our development server, so run npm start and the browser will open react-app instantly.



React boilerplate files cleanup

We must first tidy up our projects by eliminating some of the files provided by create-react-app before we can begin creating them. After you've cleaned up your files and folder , they should look like this.

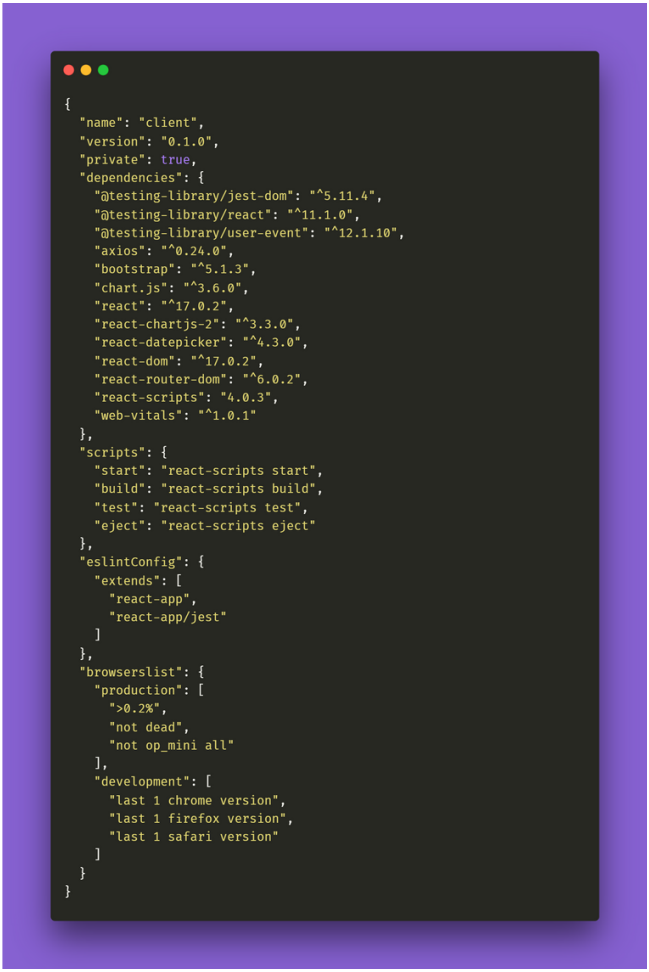


Adding and Installing some packages

We will need to install a few third-party packages for this project. so copy and paste the following command into your terminal

```
npm install bootstrap react-chartjs-2 chart.js
axios react-datepicker react-router-dom
```

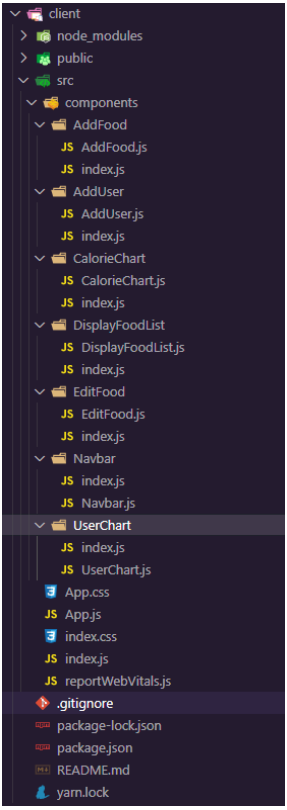
After installing all these packages your **packge.json** file of client should look like this:



```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "axios": "^0.24.0",
    "bootstrap": "^5.1.3",
    "chart.js": "^3.6.0",
    "react": "^17.0.2",
    "react-chartjs-2": "^3.3.0",
    "react-datepicker": "^4.3.0",
    "react-dom": "^17.0.2",
    "react-router-dom": "^6.0.2",
    "react-scripts": "4.0.3",
    "web-vitals": "^1.0.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Let's construct seven separate folders / component inside the components folder after we've installed all of our project's dependencies and name it as Navbar, CalorieChart, UserChart, AddFood , AddUser , EditFood and DisplayFoodList .

Your file and folder structure should look something like this once you've added all of your components.



Now go to your app.js file and import the routers from react-router-dom and styles, as well as the bootstrap css file, also all the components as well and make the necessary changes to the code as follows.


```
// app.js
import { BrowserRouter as Router, Routes, Route } from
"react-router-dom";
import "./App.css";
import "bootstrap/dist/css/bootstrap.min.css";

import Navbar from "./components/Navbar";
import DisplayFoodList from "./components/DisplayFoodList";
import EditFood from "./components/EditFood";
import AddFood from "./components/AddFood";
import AddUser from "./components/AddUser";
function App() {
  return (
    <>
      <Router>
        <Navbar />
        <br />
        <Routes>
          <Route path="/" exact element={<DisplayFoodList />}
/>
```

```
          <Route path="/edit/:id" element={<EditFood />} />
          <Route path="/create" element={<AddFood />} />
          <Route path="/user" element={<AddUser />} />
        </Routes>
      </Router>
    </>
  );
}

export default App;
```

then go to the navbar component and paste the code below into it.


```
//components/Navbar/Navbar.js
import React from "react";
import { Link } from "react-router-dom";

const Navbar = () => {
  return (
    <nav
      className="navbar navbar-expand-lg navbar-light
static-top mb-0 shadow"
      style={{ backgroundColor: "#8661d1" }}
    >
      <div className="container">
        <Link to="/">
```

```
          <img
            alt="Calorie Journal Logo"

src="https://user-images.githubusercontent.com/37651620/142762093-45207811-0c6e-4b62-9cb2-8d0009efb4ea.png"
            width="70"
            height="70"
            className="d-inline-block align-top"
          />
        </Link>
        <Link
          className="navbar-brand"
          to="/"
          className="navbar-brand"
          style={{
            color: "white",
            fontSize: "1.5rem",
            marginRight: "15rem",
            marginLeft: "30rem",
          }}
        >
          <img

src="https://user-images.githubusercontent.com/37651620/142764762-fef8f764-4cd5-44c6-8b9a-cffcfab2ccf8.png"
            alt="calorie journal"
            style={{ height: "100px" }}
          />
        </Link>

        <div className="collapse navbar-collapse">
          <ul className="navbar-nav ml-auto">
            <li className="nav-item">
              <Link
                className="nav-link"
                to="/"
```

```
            className="nav-link"
            style={{
              fontSize: "0.2rem",
              color: "white",
            }}
          >
            <button type="button" className="btn btn-info">
              Calorie Info
            </button>
          </Link>
        </li>
        <li className="nav-item active">
          <Link
            className="nav-link"
            to="/create"
            className="nav-link"
            style={{
              fontSize: "0.2rem",
              color: "white",
            }}
          >
            <button type="button" className="btn btn-info">
              ➕ Add food
            </button>
          </Link>
        </li>
        <li className="nav-item">
          <Link
            className="nav-link"
            to="/user"
            className="nav-link"
            style={{
              fontSize: "0.2rem",
              color: "white",
            }}
          >
```

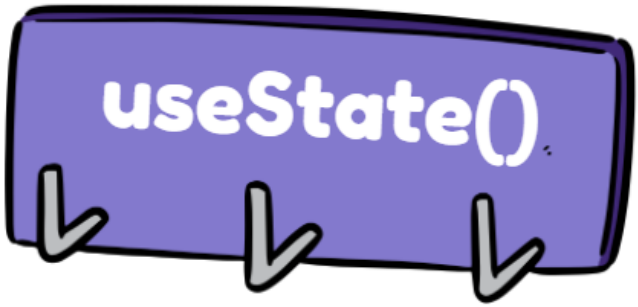
```
                <button type="button" className="btn
btn-warning">
                  ➕ Add User
                </button>
              </Link>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  );
};

export default Navbar;
```

It's time to define our AddFood component now that we've successfully introduced the navbar component to our application.

```
import React,{useState,useEffect,useRef} from "react";
import axios from "axios";
import DatePicker from "react-datepicker";
import "react-datepicker/dist/react-datepicker.css";
```

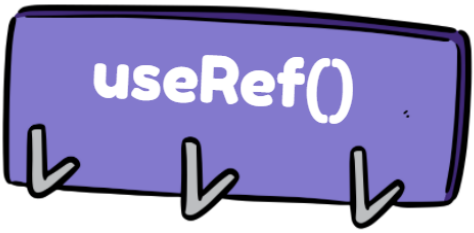
In AddFood component, add a useState() hook, which will allow us to incorporate the state into our functional component. useState() does not operate with object values, unlike state in class components. We can use primitives directly to build multiple react hooks for multiple variables if necessary.

`const [state, setState] = useState(initialState);`



Hooks must always be declared at the beginning of a function in React. This also aids in the component's state maintenance as well as preservation between renderings.

```
const [username,setUsername] = useState("");
const [description,setDdescription] = useState("");
const [calories,setCalories] = useState("");
const [date,setDate] = useState(new Date());
const [users,setUsers] = useState([]);
```



what is useRef() hook ?

This hook simply returns a mutable ref object with the passed argument as its.current property (initialValue). The returned

object will be retained for the duration of the component's lifetime.

```
const refContainer = useRef(initialValue);
```

Let us jump right back into the code and implement useRef functionality

```
const userInputRef = useRef("userInput");
```

Let's have a look at the useEffect() hook. You notify React that your component needs to perform something after it renders by using this Hook. After completing the DOM modifications, React will remember the function you gave (which we'll refer to as our "effect"). We set the document title to achieve this, but we could alternatively perform data fetching or call another imperative API. Using useEffect() within the component allows us to directly access the count state variable (or any props) from the effect. It's already in the function scope, so we don't need a new API to read it. Hooks make use of JavaScript closures rather than providing React-specific APIs where JavaScript already provides it. useEffect() The hook is comparable to the life-cycle methods for class components that we are familiar with. It executes after each component render, including the initial render. As a result, componentDidMount, componentDidUpdate, and componentWillUnmount can all be thought of as a single

component. We can pass dependencies to the effect to determine the behavior of when the effect should execute (just on initial render, or only when a specific state variable changes). This hook also has a clean-up option, which allows resources to be cleaned up before the component is destroyed. useEffect(didUpdate) is the effect's fundamental syntax.

Let's make a function that fetches all the user information

```
useEffect(() => {
  axios
    .get("http://localhost:5000/users/")
    .then((response) => {
      if (response.data.length > 0) {
        setUsers(response.data.map((user) =>
user.username));
        setUsername(response.data[0].username);
      }
```

```
    })
    .catch((error) => {
      console.log(error);
    });
}, []);
```

Now, create five function or handlers and name it as handleUsername, handlDescription, handleCalories, handleDate and handleSubmit

```
function handleUsername(e) {
  setUsername(e.target.value);
}
function handleDescription(e) {
  setDescription(e.target.value);
}
function handleCalories(e) {
  setCalories(e.target.value);
}
function handleDate(date) {
  setDate(date);
}
function handleSubmit(e) {
  e.preventDefault();
  const meal = {
    username,
    description,
    calories,
    date,
  };
```

```
  console.log(meal);

  axios
    .post("http://localhost:5000/calorie/add", meal)
    .then((res) => console.log(res.data));

  window.location = "/";
}
```

Finally, your AddFood component should look something like this


```
//components/AddFood
import React, { useState, useEffect, useRef } from "react";
import axios from "axios";
import DatePicker from "react-datepicker";
import "react-datepicker/dist/react-datepicker.css";

const AddFood = () => {
  const [username, setUsername] = useState("");
  const [description, setDescription] = useState("");
  const [calories, setCalories] = useState("");
  const [date, setDate] = useState(new Date());
```

```
const [users, setUsers] = useState([]);
const userInputRef = useRef("userInput");

useEffect(() => {
  axios
    .get("http://localhost:5000/users/")
    .then((response) => {
      if (response.data.length > 0) {
        setUsers(response.data.map((user) => user.username));
        setUsername(response.data[0].username);
      }
    })
    .catch((error) => {
      console.log(error);
    });
}, []);

function handleUsername(e) {
  setUsername(e.target.value);
}

function handleDescription(e) {
  setDescription(e.target.value);
}

function handleCalories(e) {
  setCalories(e.target.value);
}

function handleDate(date) {
  setDate(date);
}

function handleSubmit(e) {
  e.preventDefault();
```

```
    const meal = {
      username,
      description,
      calories,
      date,
    };

    console.log(meal);

    axios
      .post("http://localhost:5000/calorie/add", meal)
      .then((res) => console.log(res.data));

    window.location = "/";
  }
  return (
    <>
      <div className="container">
        <div className="card border-0 shadow my-4">
          <div className="card-body p-3"></div>
          <div>
            <h3 style={{ textAlign: "center" }}>
              <img

src="https://user-images.githubusercontent.com/37651620/142764215-78f5b75f-4871-451e-9a4d-dd77cc667fc5.png"
                alt="Food"
                style={{ height: "150px" }}
              />{" "}
            </h3>
            <form onSubmit={handleSubmit}>
              <div
                className="form-group"
                style={{
                  marginLeft: "20px",
                  marginBottom: "15px",
```

```
      marginRight: "20px",
    }}
  >
    <label>👤 User name: </label>
    <select
      ref={userInputRef}
      required
      className="form-control"
      value={username}
      onChange={handleUsername}
    >
      {users.map(function (user) {
        return (
          <option key={user} value={user}>
            {user}
          </option>
        );
      })}
    </select>
  </div>
  <div
    className="form-group"
    style={{
      marginLeft: "20px",
      marginBottom: "25px",
      marginRight: "20px",
    }}
  >
    <label>🥡 Food Info: </label>
    <input
      type="text"
      required
      className="form-control"
      value={description}
      onChange={handleDescription}
    />
```

```
            </div>
            <div
              className="form-group"
              style={{
                marginLeft: "20px",
                marginBottom: "15px",
                marginRight: "20px",
              }}
            >
              <label>🔥 Calories: </label>
              <input
                type="text"
                className="form-control"
                value={calories}
                onChange={handleCalories}
              />
            </div>
            <div
              className="form-group"
              style={{
                marginLeft: "20px",
                marginBottom: "15px",
                marginRight: "20px",
              }}
            >
              <div style={{ textAlign: "center", cursor:
"pointer" }}>
                <label>Date: </label>
                <div>
                  <DatePicker selected={date}
onChange={handleDate} />
                </div>
              </div>
            </div>

            <div className="form-group" style={{ textAlign:
```

```
"center" }}>
                <input
                  type="submit"
                  value="Add Meal"
                  className="btn"
                  style={{
                    color: "white",
                    backgroundColor: "#8661d1",
                    marginBottom: "25px",
                  }}
                />
              </div>
            </form>
          </div>
        </div>
      </div>
    </>
  );
};

export default AddFood;
```

Now, It's time to define our AddUser component now that we've successfully introduced the AddFood component to our application. Copy the following code and paste it inside the AddUser component.


```
//components/AddUser
import React, { useState } from "react";
import axios from "axios";

const AddUser = () => {
  const [username, setUsername] = useState("");

  function handleUsername(e) {
    setUsername(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault();
    const user = {
      username,
    };
    console.log(user);
    axios
      .post("http://localhost:5000/users/add", user)
      .then((res) => console.log(res.data));
```

```
    setUsername("");
  }

  return (
    <>
      <div class="container">
        <div class="card border-0 shadow my-4">
          <div class="card-body p-3"></div>
          <div>
            <h3 style={{ textAlign: "center",
marginBottom: "15px" }}>
              <img

src="https://user-images.githubusercontent.com/37651620/1
42767072-ff777861-7ee9-4355-b48e-a624e8de085b.png"
                alt="Logo"
                style={{ height: "150px" }}
              />
            </h3>
            <form onSubmit={handleSubmit}>
              <div
                className="form-group"
                style={{
                  marginLeft: "20px",
                  marginBottom: "15px",
                  marginRight: "20px",
                }}
              >
                <label>👤 User name:</label>
                <input
                  type="text"
                  required
```

```
                  className="form-control"
                  value={username}
                  onChange={handleUsername}
                />
              </div>
              <div
                className="form-group"
                style={{
                  textAlign: "center",
                }}
              >
                <input
                  type="submit"
                  value="Create User"
                  className="btn "
                  style={{
                    color: "white",
                    marginBottom: "25px",
                    backgroundColor: "#8661d1",
                  }}
                />
              </div>
            </form>
          </div>
        </div>
      </div>
    </>
  );
};

export default AddUser;
```

Now that we've completed the AddUser component, it's time to construct a feature that allows us to change our data, therefore we'll make an EditFood component.


```
//components/EditFood
import React, { useState, useEffect, useRef } from "react";
import axios from "axios";
import DatePicker from "react-datepicker";
import "react-datepicker/dist/react-datepicker.css";

const EditFood = (props) => {
  const [username, setUsername] = useState("");
  const [description, setDescription] = useState("");
  const [calories, setCalories] = useState("");
  const [date, setDate] = useState(new Date());
  const [users, setUsers] = useState([]);
  const userInputRef = useRef("userInput");

  useEffect(() => {
    axios
      .get("http://localhost:5000/calorie/" +
props.match.params.id)
      .then((response) => {
        setUsername(response.data.username);
        setDescription(response.data.description);
        setCalories(response.data.calories);
        setDate(new Date(response.data.date));
      })
      .catch((error) => {
```

```
        console.log(error);
      });

    axios
      .get("http://localhost:5000/users/")
      .then((response) => {
        if (response.data.length > 0) {
          setUsers(response.data.map((user) =>
user.username));
          setUsername(response.data[0].username);
        }
      })
      .catch((error) => {
        console.log(error);
      });
  }, [props.match.params.id]);

  function handleUsername(e) {
    setUsername(e.target.value);
  }

  function handleDescription(e) {
    setDescription(e.target.value);
  }

  function handleCalories(e) {
    setCalories(e.target.value);
  }

  function handleDate(date) {
    setDate(date);
  }

  function handleSubmit(e) {
    e.preventDefault();
```

```
    const food = {
      username,
      description,
      calories,
      date,
    };

    console.log(food);

    axios
      .post("http://localhost:5000/calorie/update", food)
      .then((res) => console.log(res.data));

    window.location = "/";
  }
  return (
    <>
      <div className="container">
        <div className="card border-0 shadow my-4">
          <div className="card-body p-3"></div>
          <div>
            <h3 style={{ textAlign: "center" }}>
              <img

src="https://user-images.githubusercontent.com/37651620/142764
215-78f5b75f-4871-451e-9a4d-dd77cc667fc5.png"
                alt="Food"
                style={{ height: "150px" }}
              />{" "}
            </h3>
            <form onSubmit={handleSubmit}>
              <div
                className="form-group"
                style={{
                  marginLeft: "20px",
                  marginBottom: "15px",
```

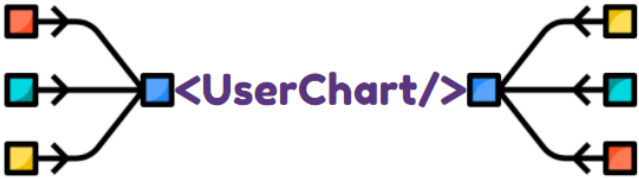
```
        marginRight: "20px",
      }}
    >
      <label>👤 User name: </label>
      <select
        ref={userInputRef}
        required
        className="form-control"
        value={username}
        onChange={handleUsername}
      >
        {users.map(function (user) {
          return (
            <option key={user} value={user}>
              {user}
            </option>
          );
        })}
      </select>
    </div>
    <div
      className="form-group"
      style={{
        marginLeft: "20px",
        marginBottom: "25px",
        marginRight: "20px",
      }}
    >
      <label>📦 Food Info: </label>
      <input
        type="text"
        required
        className="form-control"
        value={description}
        onChange={handleDescription}
      />
```

```
              </div>
              <div
                className="form-group"
                style={{
                  marginLeft: "20px",
                  marginBottom: "15px",
                  marginRight: "20px",
                }}
              >
                <label>🔥 Calories: </label>
                <input
                  type="text"
                  className="form-control"
                  value={calories}
                  onChange={handleCalories}
                />
              </div>
              <div
                className="form-group"
                style={{
                  marginLeft: "20px",
                  marginBottom: "15px",
                  marginRight: "20px",
                }}
              >
                <div style={{ textAlign: "center", cursor:
"pointer" }}>
                  <label>Date: </label>
                  <div>
                    <DatePicker selected={date}
onChange={handleDate} />
                  </div>
                </div>
              </div>

              <div className="form-group" style={{ textAlign:
```

```
"center" }}>
                <input
                  type="submit"
                  value="Add Meal"
                  className="btn"
                  style={{
                    color: "white",
                    backgroundColor: "#8661d1",
                    marginBottom: "25px",
                  }}
                />
              </div>
            </form>
          </div>
        </div>
      </div>
    </>
  );
};

export default EditFood;
```

Let's concentrate on visualizing the fetched data into charts using the react-chartjs-2 library before we start fetching and showing the whole information on our home page.

So let's make two distinct components, one for a bar graph and the other for a pie chart, and once you've done that, copy the following code into each component.




```
//components/UserChart.js
import React, { useEffect, useState } from "react";
import { Pie } from "react-chartjs-2";
import axios from "axios";

const Delayed = ({ children, waitBeforeShow = 4500 }) =>
{
  const [isShown, setIsShown] = useState(false);

  useEffect(() => {
    setTimeout(() => {
      setIsShown(true);
    }, waitBeforeShow);
  }, [waitBeforeShow]);

  return isShown ? children : null;
};

const UserChart = () => {
  const [chartData, setChartData] = useState({});

  async function getData() {
    let username = [];
    let calories = [];
    await axios
      .get("http://localhost:5000/calorie/")
      .then((res) => {
```

```
      console.log(res);
      for (const dataObj of res.data) {
        username.push(dataObj.username);
        calories.push(parseInt(dataObj.calories));
        console.log(username, calories);
      }
      setChartData({
        labels: username,
        datasets: [
          {
            label: "Calories",
            data: calories,
            backgroundColor: [
              "#f42f42",
              "#5ab950",
              "#fe812a",
              "#ffc748",
              "#6b71c7",
              "#8661d1",
              "#8a2cba",
            ],
            borderColor: [
              "#f42f42",
              "#5ab950",
              "#fe812a",
              "#ffc748",
              "#6b71c7",
              "#8661d1",
              "#8a2cba",
            ],
            borderWidth: 2,
          },
```

```
          ],
        });
      })
      .catch((err) => {
        console.log(err);
      });
    console.log(username, calories);
}

useEffect(() => {
  getData();
}, []);

return (
  <div className="App">
    <div>
      <h5
        style={{
          fontSize: "20",
          textAlign: "center",
          marginTop: "1em",
          marginBottom: "1em",
        }}
      >
        Calorie per user
      </h5>
      <Delayed>
        <Pie
          data={chartData}
          options={{
            title: {
              text: "Calorie per User",
```

```
                fontSize: 10,
                fontColor: "#212529",
              },
              maintainAspectRatio: true,
            }}
          />
        </Delayed>
      </div>
    </div>
  );
};

export default UserChart;
```


```
//components/CalorieChart
import React, { useEffect, useState } from "react";
import { Bar } from "react-chartjs-2";
import axios from "axios";

const Delayed = ({ children, waitBeforeShow = 4500 }) =>
{
  const [isShown, setIsShown] = useState(false);

  useEffect(() => {
    setTimeout(() => {
      setIsShown(true);
    }, waitBeforeShow);
  }, [waitBeforeShow]);

  return isShown ? children : null;
};

const CalorieChart = () => {
  const [chartData, setChartData] = useState({});

  async function getData() {
    let foodCal = [];
    let caloriesCal = [];
    await axios
```

```
      .get("http://localhost:5000/calorie/")
      .then((res) => {
        console.log(res);
        for (let dataObj of res.data) {
          foodCal.push(dataObj.description);

caloriesCal.push(parseInt(dataObj.caloriesCal));
          console.log("foodCal, caloriesCal", foodCal,
caloriesCal);
        }
        setChartData({
          labels: foodCal,
          datasets: [
            {
              label: "Cal",
              data: caloriesCal,
              backgroundColor: [
                "#f42f42",
                "#5ab950",
                "#fe812a",
                "#ffc748",
                "#6b71c7",
                "#8661d1",
                "#8a2cba",
              ],
            },
          ],
        });
      })
      .catch((err) => {
        console.log(err);
      });
```

```
}

useEffect(() => {
  getData();
}, []);

return (
  <div className="App">
    <h4>Food Analytics</h4>

    <h5
      style={{
        fontSize: "20",
        textAlign: "center",

        marginBottom: "1em",
      }}
    >
      Calorie Intake per each Food
    </h5>
    <div>
      <Delayed>
        <Bar
          data={chartData}
          options={{
            responsive: true,
            title: {
              text: "Calorie Per Food ",
              fontSize: 20,
              fontColor: "#212529",
            },
            scales: {
```

```
                yAxes: [
                  {
                    ticks: {
                      autoSkip: true,
                      maxTicksLimit: 10,
                      beginAtZero: true,
                    },
                    gridLines: {
                      // display: true,
                    },
                  },
                ],
                xAxes: [
                  {
                    gridLines: {
                      display: true,
                    },
                  },
                ],
              },
            }}
          />
        </Delayed>
      </div>
    </div>
  );
};

export default CalorieChart;
```

Finally, let's work on the DisplayFoodList component, so first import the link from react-router, then import the axios package, then import the two previously created chart components, then create a FoodTrack component inside the DisplayFoodList file and add the following code, and finally create three functions named DisplayFoodList, deleteMeal, and malList, and finally use all the imported data inside the return statement and don't forget to invoke the mailList function inside the tbody. Finally, if you followed all the steps correctly then your DisplayFoodList component should resemble the following.


```
//components/DisplayFoodList
import React, { useState, useEffect } from "react";
import { Link } from "react-router-dom";
import axios from "axios";
import CalorieChart from "../CalorieChart";
import UserChart from "../UserChart";

const FoodTrack = (props) => (
  <tr>
    <td>
      <Link to={"/edit/" + props.meal._id} style={{ color: "
#a04949" }}>
```

```
        <img

src="https://user-images.githubusercontent.com/37651620/142769
270-6128d45e-3650-4b66-bc0b-a76e3991fa1f.png"
          alt="edit"
          style={{ height: "40px" }}
        />
      </Link>{" "}
      |{" "}
      <a
        href="#"
        onClick={() => {
          props.deleteMeal(props.meal._id);
          window.location.reload(false);
        }}
        style={{ color: " #a04949" }}
      >
        <img

src="https://user-images.githubusercontent.com/37651620/142769
328-23d55107-8bed-4fa0-92b8-cca7df931083.png"
          alt="edit"
          style={{ height: "40px" }}
        />
      </a>
    </td>
    <td>{props.meal.username}</td>
    <td>{props.meal.description}</td>
    <td>{props.meal.calories}</td>
    <td>{props.meal.date.substring(0, 10)}</td>
  </tr>
);

const DisplayFoodList = () => {
  const [foods, setFoods] = useState([]);
```

```
  useEffect(() => {
    axios
      .get("http://localhost:5000/calorie/")
      .then((response) => {
        setFoods(response.data);
      })
      .catch((error) => {
        console.log(error);
      });
  }, []);

  function deleteMeal(id) {
    axios.delete("http://localhost:5000/calorie/" +
id).then((response) => {
      console.log(response.data);
    });
    setFoods(foods.filter((el) => el._id !== id));
  }

  const mealList = () => {
    return foods.map((currentmeal) => {
      return (
        <FoodTrack
          meal={currentmeal}
          deleteMeal={deleteMeal}
          key={currentmeal._id}
        />
      );
    });
  };



  return (
    <>
        <div className="container">
```

```
          <div className="card border-0 shadow my-4">
            <div className="card-body p-5">
              <h3 style={{ textAlign: "center", marginBottom:
"15px" }}>
                Calorie Journal
              </h3>
              <table className="table" style={{ textAlign:
"center" }}>
                <thead className="thead" style={{
backgroundColor: "#8661d1" }}>
                  <tr>
                    <th>Edit/Delete</th>
                    <th>👤 Username</th>
                    <th>📙 Description</th>
                    <th>🔥 Calories</th>
                    <th>📅 Date</th>
                  </tr>
                </thead>
                <tbody>{mealList()}</tbody>
              </table>
            </div>
          </div>
        </div>
        <div className="container">
          <div
            className="card border-0 shadow my-2"
            style={{ padding: "2rem" }}
          ><div className="card-body p-1"></div>
            <UserChart />
            <CalorieChart /></div></div>
      </>
  )};
export default DisplayFoodList;
```

We've covered a lot of ground to provide you with the information you'll need to create a full-fledged MERN stack application from the ground up.

You can find the whole source code here.

https://github.com/pramit-marattha/MERN-saas-project

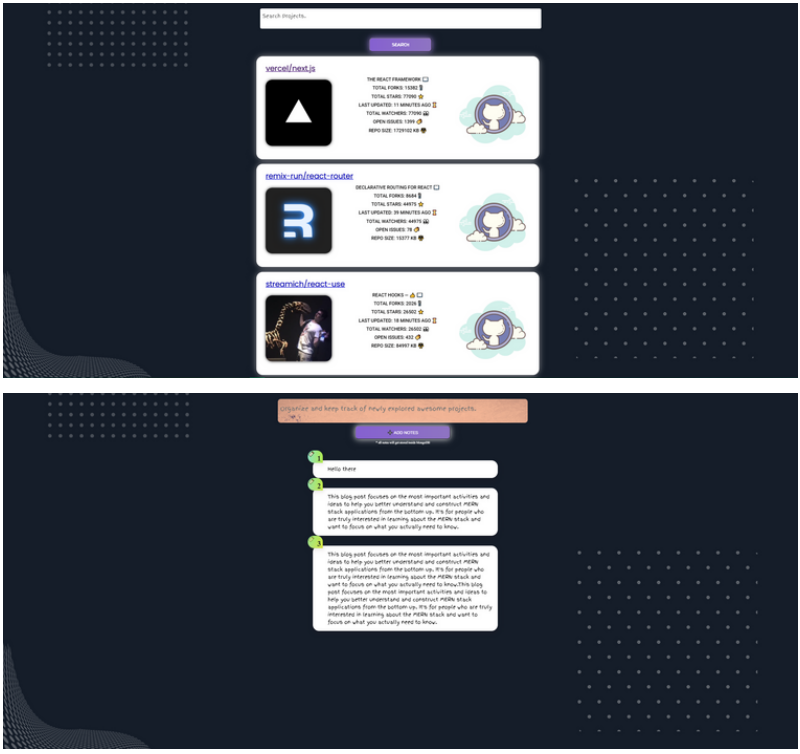
This section focuses on the most important activities and ideas to help you better understand and construct MERN stack applications from the bottom up. It's for people who are truly interested in learning about the MERN stack and want to focus on what you actually need to know.

There is a separate article where you may learn about MERN stack in great detail.

https://aviyel.com/post/1323/building-a-calorie-journal-saas-based-project-using-mern-stack

In this section, we'll construct a full-stack project case study application that users can use to keep track of and make notes on GitHub projects, as well as search the entire GitHub project using the GitHub API and the MERN stack alone. This blog lesson should help you learn the MERN stack technology's fundamentals as well as advanced concepts and operations.

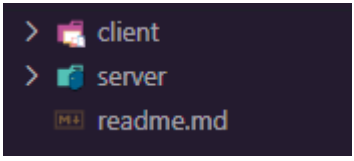
Here's a look at the final version of our application.



Setting up the folder structure

Create a two folder name client and server inside your project directory, then open it inside the Visual Studio Code or any code editor of your choice.

```
mkdir client
mkdir server
```



Now, we'll construct a MongoDB database, set up a server with Node and Express, create a database schema to represent our project case study application, and set up API routes to create, read, update, and delete data and information from the database using npm and the appropriate packages. So, open a command prompt and navigate to the directory on your server, then run the code below.

```
npm init -y
```

Setting up our package.json file

Execute the following commands in the terminal to install the dependencies.

```
npm install cors dotenv express mongoose
nodemon body-parser
```

The "**package.json**" file should look like this after the dependencies have been installed.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "take notes while exploring projects",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js"
  },
  "keywords": [
    "mern"
  ],
  "author": "pramit marattha",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "cors": "^2.8.5",
    "dotenv": "^8.6.0",
    "express": "^4.17.1",
    "mongoose": "^5.13.13",
    "nodemon": "^2.0.15",
    "pusher": "^4.0.2"
  }
}
```

And also, remember to update the scripts as well.

Now go to your server directory and make a index.js file there.

Setting up index.js

- Import express module.
- Import and configure dotenv module
- Import CORS module
- Use express() to start our app.

```
//index.js
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')

// dotenv config
require('dotenv').config();

// app config
const app = express();
```

We may now utilize all of the other methods on that app instance. Let's start with the fundamentals and very basic setups. Don't forget to set up the port, too.

```
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')

// dotenv config
require('dotenv').config();

// app and port config
const app = express();
const port = process.env.PORT || 4000;

// middlewares
app.use(bodyParser.urlencoded({ extended: true }));
app.use(cors());
app.use(bodyParser.json());
```

Setting up MongoDB cloud cluster

MongoDB is an open-source, cross-platform document-oriented database. MongoDB is a NoSQL database with optional schemas that stores data as JSON-like documents. Prior to October 16, 2018, all versions were distributed under the AGPL license. The SSPL license v1 applies to all versions issued after October 16, 2018, including bug fixes for older versions.

To set up and start your MongoDB cluster, follow the exact same steps outlined in the article mentioned below.

https://aviyel.com/post/1304/building-a-mern-stack-blog-site-from-absolute-scratch

Now create a separate database folder and inside it create another index.js file. Inside it, import the mongoose library and create a string and simply paste the copied mongo DB connection URL or simply paste the link for the environment variables. Now, inside the link of Mongo DB cloud atlas URL, enter your username and password, making sure to remove all the brackets and enter your own credentials. finally, we will use mongoose to connect to our database, so enter mongoose. connect() which is a function with two different parameters. The first will be the MONGO_DB_URL, and the second will be an object with two different options. The first is useNewUrlParser, which we will set to true, and the second is useUnifiedTopology, which we will also set to true. These objects are not required, but we will see some errors or warnings on our console. Following that, let's chain a.then() and.catch() because this will return a promise, so inside .then() will call the app and invoke listen, which has two parameters, the first of which is PORT and the second of which is the callback function that will be executed if our application is successfully connected and finally, if the connection to the database is not successful we will simply console log our error message and finally export that database.

```
const mongoose = require('mongoose');
require('dotenv').config();


mongoose.connect(process.env.MONGO_DB_URL, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
}).catch(e => {
    console.error('Error while connecting to the database',
e.message)
})

const Database = mongoose.connection

module.exports = Database;
```

Insert mongodb+srv into the .env **file.**

```
PORT=4000
MONGO_DB_URL=mongodb+srv://pramit:<password>@cl
uster0.yxjll.mongodb.net/TakeNote?retryWrites=t
rue&w=majority
```

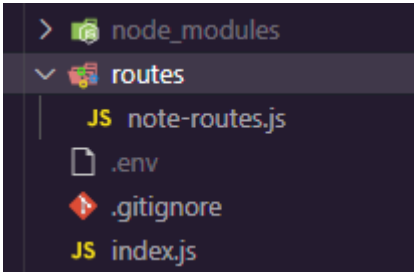
That's all there is to it; we've successfully created our database. so, let’s import it into our main root index.js file and actually connect our database with the server.

```
const express = require('express')
const bodyParser = require('body-parser')
```

```
const cors = require('cors')

// importing database
const database = require('./database')

// dotenv config
require('dotenv').config();

// app and port config
const app = express();
const port = process.env.PORT || 4000;

// middlewares
app.use(bodyParser.urlencoded({ extended: true
}));
app.use(cors());
app.use(bodyParser.json());

// DB connection
database.on('error',
console.error.bind(console, 'MongoDB failed to
connect'))

// listening to port
app.listen(port, () => console.log(`Currently
server is running at
http://localhost:${port}`))
```

We now have successfully connected our server to the database.

Now that we've successfully connected to our database, let's get started on building our backend application's routes. To do so, we'll need to create a new folder called routes on the server directory. We will create a file called notes-routes.js within the routes folder.



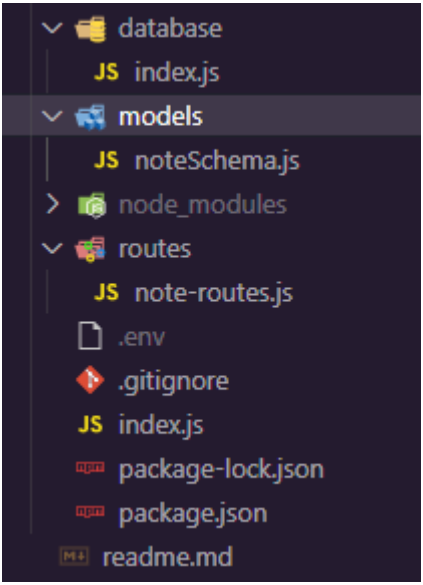
Let's get started by importing the note routes into your index.js file. We can now connect notes to our application using express middleware. Finally, your root index.js file should be like the following.

```
// index.js
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')

// importing database
const database = require('./database')
// importing routes
```

```
const noteRouter = require('./routes/note-routes')

// dotenv config
require('dotenv').config();

// app and port config
const app = express();
const port = process.env.PORT || 4000;

// middlewares
app.use(bodyParser.urlencoded({
    extended: true
}));
app.use(cors());
app.use(bodyParser.json());

// DB connection
database.on('error', console.error.bind(console, 'MongoDB
failed to connect'))

app.use('/', noteRouter)

// listening to port
app.listen(port, () => console.log(`Currently server is
running at http://localhost:${port}`))
```

We'll segregate our routes and controllers folder. But first, let's create a note model. Create a folder called models, and inside it, create one files called noteSchema.js and with the following code pasted into each.

The structure of your folders should now look like this.




```
// models/noteSchema.js
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const Note = new Schema({
    note: {
        type: String,
        required: true
    },
}, {
    timestamps: true
}, )

module.exports = mongoose.model('notes', Note)
```

Now we can begin adding our routes and conterollers.

```
// routes/note-rotes.js
const express = require('express')

const noteController =
require('../controllers/noteControllers')

const router = express.Router()

router.post('/', noteController.createItem)
router.get('/', noteController.getNotes)

module.exports = router;
```

Now create a controller folder and inside it create a file called noteControllers and inside it create a two controller named as createItem and getNotes

Import note schema

```
const Note = require('../models/noteSchema')
```

Creating a note

```
createItem = (req, res) => {

        const body = req.body

        if (!body) {
                return res.status(400).json({
                        success: false,
                        error: 'Please!! enter a item',
                })
        }
        const note = new Note(body)

        if (!note) {
                return res.status(400).json({
                        success: false,
                        error: err
                })
        }
        note.save().then(() => {
                        return res.status(200).json({
                                success: true,
                                id: note._id,
                                message: 'Cheers!! Note is
Created',
                        })
                })
                .catch(error => {
                        return res.status(400).json({
                                error,
                                message: 'Error!! while creating
note',
                        });
                });
};
```

Fetching all the notes

```
getNotes = async(req, res) => {
	await Note.find({}, (err, notes) => {
		if (err) {
			return res.status(400).json({
				success: false,
				error: err
			})
		}
		if (!notes.length) {
			return res
				.status(404)
				.json({
					success: false,
					error: `Sorry, Item not
found`
				})
		}
		return res.status(200).json({
			success: true,
			data: notes
		})
	}).catch(err => console.log(err))
}
```

Finally, your controllers should resemble something like this

```
  //controllers/noteControllers.js
const Note = require('../models/noteSchema')

createItem = (req, res) => {

    const body = req.body

    if (!body) {
        return res.status(400).json({
            success: false,
            error: 'Please!! enter a item',
        })
    }
    const note = new Note(body)

    if (!note) {
        return res.status(400).json({
            success: false,
            error: err
        })
    }
    note.save().then(() => {
            return res.status(200).json({
                success: true,
                id: note._id,
                message: 'Cheers!! Note is Created',
            })
        })
        .catch(error => {
            return res.status(400).json({
                error,
                message: 'Error!! while creating note',
            });
        });
};


getNotes = async (req, res) => {
    await Note.find({}, (err, notes) => {
        if (err) {
            return res.status(400).json({
                success: false,
                error: err
```

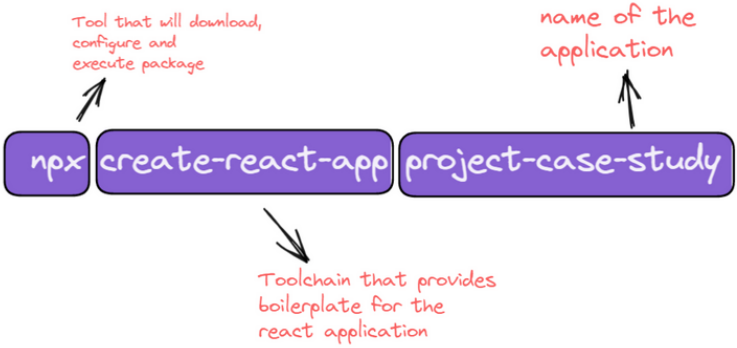
```
            })
        }
        if (!notes.length) {
            return res
                .status(404)
                .json({
                    success: false,
                    error: `Sorry, Item not found`
                })
        }
        return res.status(200).json({
            success: true,
            data: notes
        })
    }).catch(err => console.log(err))
}

module.exports = {
    createItem,
    getNotes
}
```

After restarting the server, you should see something similar to this:

```
[nodemon] 2.0.14
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server is running at: http://localhost:4000
```

Setting up the frontend with react

Let's start with the frontend and use react to construct it. If Node.js isn't currently installed on your computer, the first thing you need to do is install it. So, go to the official Node.js website and get the most recent version. In order to use the node package manager, also known as NPM, you'll need Node js. Now open your favourite code editor and navigate to the client folder. I'll be utilizing Visual Studio Code. Next, input npx create-react-app into the integrated terminal. This command will construct a client application with the name client in the current directory.

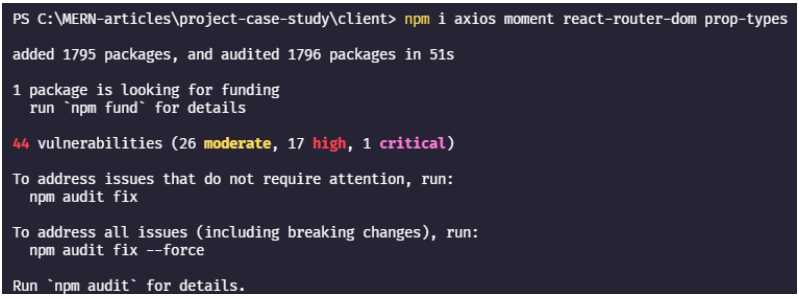


There is a separate article where you may learn everything there is to know about react.js

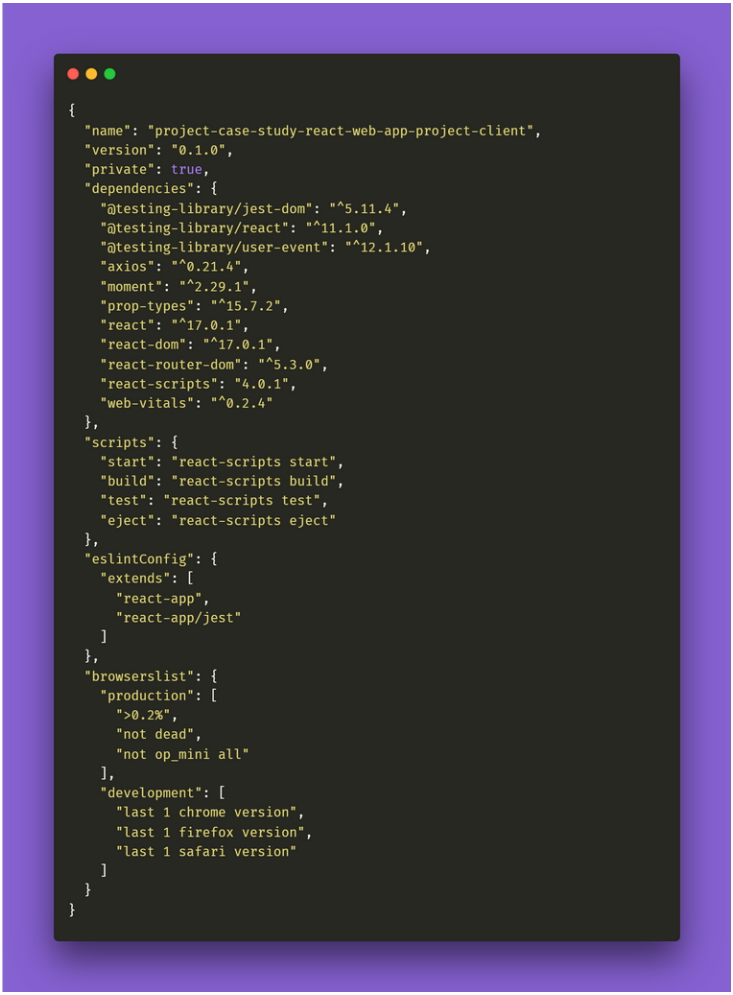
https://aviyel.com/post/1190/building-a-react-application-from-absolute-scratch

It's time to install some packages within react-boilerplate now that you've installed and cleaned it. so copy and paste the following command into your terminal.

```
npm i axios moment react-router-dom prop-types
```



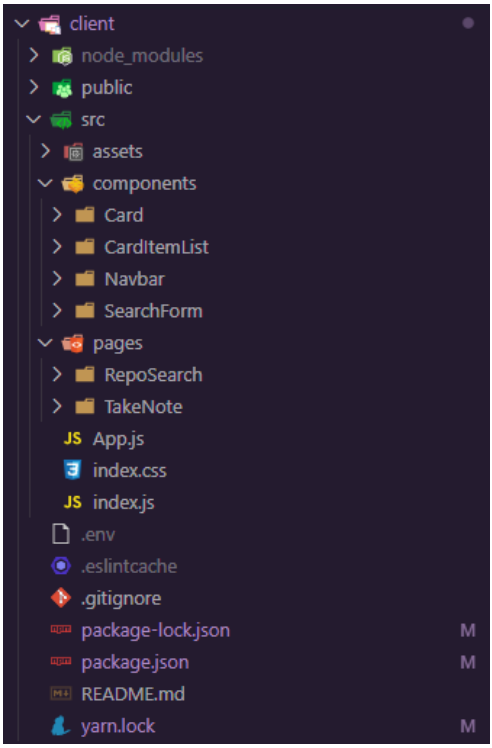
After installing all these packages your **packge.json** file of the client should look like this:



```
{
  "name": "project-case-study-react-web-app-project-client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "axios": "^0.21.4",
    "moment": "^2.29.1",
    "prop-types": "^15.7.2",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-router-dom": "^5.3.0",
    "react-scripts": "4.0.1",
    "web-vitals": "^0.2.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Let's construct four separate folders/component inside the components folder after we've installed all of our project's dependencies and name it as Card , CardItemList , Navbar and SearchForm and also let’s create one folder called pages and create two older ones inside it and name it as RepoSearch and TakeNote respectively.

Your file and folder structure should look something like this once you've added all of your components and pages.



Now go to your app.js file and import the routers from react-router-dom and styles, also all the components as well and make the necessary changes to the code as follows.

```
// app.js
import React from "react";
import { BrowserRouter, Route } from
"react-router-dom";

import Navigation from "./components/Navbar";
import RepoSearch from
"./pages/RepoSearch/RepoSearch";
import TakeNote from
"./pages/TakeNote/TakeNote";
const App = () => {
  return (
    <BrowserRouter>
      <Navigation />
      <Route path="/" exact
component={RepoSearch} />
      <Route path="/note" exact
component={TakeNote} />
    </BrowserRouter>
  );
};

export default App;
```

then go to the card component and create a card like structure for each fetched github projects.

```
// components/card
import React from "react";
import "./Card.css";
import moment from "moment";

const Card = (props) => {
  return (
    <div className="container__cardbox">
      <div className="card-body">
        <div className="card-title">
          <a href={props.link} target="_blank"
rel="noreferrer">
            {props.title}
          </a>
        </div>
        <div className="card-description">{props.description}
📖</div>
        <div className="card-description">Total Forks:
{props.forks} 🍴 </div>
        <div className="card-description">Total Stars:
{props.stars} ⭐</div>
        <div className="card-description">
          Last Updated:
{moment(`${props.updatedAt}`).fromNow()} ⏳
        </div>
        {/* <div className="card-description">License Name:
{(props.licenseName === "Other" & null ) ? "Other License" :
props.licenseName} 📜</div> */}
        <div className="card-description">
          Total Watchers: {props.watchCount} 👀
        </div>
```

```
        <div className="card-description">
          Open Issues: {props.openIssuesCount} 🏷️
        </div>
        <div className="card-description">
          Repo Size: {props.repoSize} KB ⚖️
        </div>
        <img className="card-image" src={props.image}
alt={props.title} />
      </div>
    </div>
  );
};

export default Card;
```

and also don’t forget to create a card.css file inside it and add the following style to it

```
.container__cardbox {
  flex: 1;
  /* flex-grow: 4; */
  flex-basis: 15%;
  margin: 15px;
  /* border: solid 2px #383636; */
  border-radius: 25px;
  /* display: flex; */
  flex-flow: row wrap;
}
```

```css
.card-body {
  padding: 10px;
  border-radius: 20px;
  background: white;
}

.card-title {
  font-size: 25px;
  text-align: left;
}

.card-description {
  font-size: 12px;
  margin: 4px;
  text-align: center;
}

.card-image {
  width: 20%;
  margin-top: -130px;
}
```

Then, under the CardItemList, import the Card component and provide it all of the appropriate props so that it can display all of the elements within the card component itself.

```
//components/CardItemList
import React from "react";
import Card from "../Card";
const CardItemList = (props) => {
  return (
    <div className="container__carditemlist">
      {props.items.map((item) => (
        <Card
          key={item.id}
          link={item.html_url}
          title={item.full_name}
          description={item.description}
          image={item.owner.avatar_url}
          forks={item.forks_count}
          stars={item.stargazers_count}
          updatedAt={item.updated_at}
          watchCount={item.watchers_count}
        openIssuesCount={item.open_issues_count}
          repoSize={item.size}
        />
      ))}
    </div>
)
};
export default CardItemList;
```

and again don’t forget to create a CardItemList.css file inside it and add the following style to it

```
.container__carditemlist {
  display: flex;
  flex-wrap: wrap;
}
```

So, before we develop a search form, let's work on the navbar section. Go to the navbar component and paste the code below into it.

```
 // components/Navbar
import React from "react";
import { Link } from "react-router-dom";
import "./Navbar.css";

const Navbar = () => {
  return (
    <div className="container__navbar">
      <div className="navbar-title">Github Search</div>
      <ul className="navbar-menu">
        <li>
          <Link to="/">Search-Projects</Link>
        </li>
        <li>
          <Link to="/note">Take-Note</Link>
        </li>
      </ul>
      <div className="navbar-menu"></div>
    </div>
  )
};
export default Navbar;
```

Remember to create a Navbar.css file inside it and apply the following style to it.

```
@import
url("https://fonts.googleapis.com/css?family=Ra
leway:400,400i,800");
 .container__navbar {
     display: flexbox;
     align-items: center;
     background:
url("../../assets/gifs/navBack.gif") no-repeat
center center fixed;
     -webkit-background-size: cover;
     -moz-background-size: cover;
     -o-background-size: cover;
     background-size: cover;
    /* background-color: transparent;
     */
     padding: 25px;
     width: 100%;
     margin-bottom: 20px;
}
 @media only screen and (max-width: 900px) {
     .container__navbar {
         display: flexbox;
         align-items: center;
         background:
```

```
url("../../assets/gifs/navBack.gif") no-repeat
center center fixed;
          -webkit-background-size: cover;
          -moz-background-size: cover;
          -o-background-size: cover;
          background-size: cover;
         /* background-color: transparent;
          */
          padding: 25px;
          width: 100rem;
          margin-bottom: 20px;
     }
}
 .navbar-title {
      color: transparent;
      font-size: 28px;
      margin-bottom: -50px;
      text-align: right;
}
 .navbar-menu {
      border-radius: 25px;
      height: -webkit-fit-content;
      height: -moz-fit-content;
      height: fit-content;
      display: inline-flex;
      background-color: rgba(0, 0, 0, 0.4);
      -webkit-backdrop-filter: blur(10px);
```

```
    backdrop-filter: blur(10px);
    align-items: center;
    padding: 0 20px;
    margin: 50px 0 0 0;
}
 .navbar-menu li {
    list-style: none;
    color: white;
    font-family: sans-serif;
    font-weight: bold;
    padding: 12px 60px;
    margin: 0 8px;
    position: relative;
    cursor: pointer;
    white-space: nowrap;
}
 .navbar-menu li::before {
    content: " ";
    position: absolute;
    top: 0;
    left: 0;
    height: 100%;
    width: 100%;
    z-index: -1;
    transition: 0.6s;
    border-radius: 25px;
}
```

```
 .navbar-menu li:hover {
     color: black;
}
 .navbar-menu li:hover::before {
     background: linear-gradient(to bottom,
#e8edec, #d2d1d3);
     box-shadow: 0px 3px 20px 0px black;
     transform: scale(1.2);
}
 @media only screen and (max-width: 1000px) {
     .navbar-menu {
         border-radius: 25px;
         height: -webkit-fit-content;
         height: -moz-fit-content;
         height: fit-content;
         display: inline-flex;
         background-color: rgba(0, 0, 0, 0.4);
         -webkit-backdrop-filter: blur(10px);
         backdrop-filter: blur(10px);
         align-items: center;
         padding: 0 0px;
         margin: 50px 0 0 0;
    }
     .navbar-menu li {
         list-style: none;
         color: white;
         font-family: sans-serif;
```

```
        font-weight: bold;
        padding: 12px 10px;
        margin: 0 1px;
        position: relative;
        cursor: pointer;
        white-space: nowrap;
    }
     .navbar-menu li::before {
        content: " ";
        position: absolute;
        top: 0;
        left: 0;
        height: 100%;
        width: 100%;
        z-index: -1;
        transition: 0.6s;
        border-radius: 25px;
    }
     .navbar-menu li:hover {
        color: black;
    }
     .navbar-menu li:hover::before {
        background: linear-gradient(to bottom,
#e8edec, #d2d1d3);
        box-shadow: 0px 3px 20px 0px black;
        transform: scale(1.2);
    }}
```

Finally, let's go to work on the components of the SearchForm

```
// components/SearchForm
import React, { useState } from "react";
import "./SearchForm.css";

const SearchForm = (props) => {
  const [value, setValue] = useState("");

  const submitSearchValue = () => {
    setValue("");
    props.onSubmit(value);
  };

  return (
    <div>
      <input
        className="search-input"
        type="text"
        placeholder={props.placeholder}
        value={value}
        onChange={(event) =>
setValue(event.target.value)}
      />
      <label htmlFor="name"
className="search-label">
```

```
        Search Project
      </label>
      <button
        className="search-button"
        type="submit"
        onClick={() => submitSearchValue()}
      >
        {props.buttonText}
      </button>
    </div>
  );
};

export default SearchForm;
```

Remember to include the following style in the SearchForm.css file

```
@import
url("https://fonts.googleapis.com/css2?family=Finger+Paint&display=swap");

.search-button {
  background-image: linear-gradient(
    to right,
    #02aab0 0%,
    #00cdac 51%,
    #02aab0 100%
```

```
  );
}
.search-button {
  margin: 0 auto;
  padding: 10px 100px;
  margin-top: 0px;
  text-align: center;
  text-transform: uppercase;
  transition: 0.5s;
  background-size: 200% auto;
  color: white;
  box-shadow: 0 0 20px #eee;
  border-radius: 10px;
  display: block;
  outline: none;
}

.search-button:hover {
  background-position: right center; /* change the direction of
the change here */
  color: #fff;
  text-decoration: none;
}

.search-label {
  color: white;
  font-family: "Finger Paint", cursive;
  font-size: 1.2rem;
  margin-left: 2rem;
  margin-top: 0.2rem;
  display: block;
  transition: all 0.3s;
  transform: translateY(0rem);
}
```

```
.search-input {
  color: #333;
  font-size: 1.2rem;
  margin: 0 auto;
  padding: 1rem 0.5rem;
  border-radius: 0.6rem;
  background-color: rgb(255, 255, 255);
  border: none;
  width: 50rem;
  display: block;
  border-bottom: 1rem solid transparent;
  transition: all 0.3s;
  outline:none;
}
@media only screen and (max-width: 900px) {
  .search-input {
    color: #333;
    font-size: 1.2rem;
    margin: 0 auto;
    padding: 1rem 0.5rem;
    border-radius: 0.6rem;
    background-color: rgb(255, 255, 255);
    border: none;
    width: 100%;
    display: block;
    border-bottom: 1rem solid transparent;
    transition: all 0.3s;
  }
}
.search-input:placeholder-shown + .search-label {
  opacity: 0;
  color: white;
  visibility: hidden;
  -webkit-transform: translateY(-4rem);
  transform: translateY(-4rem);
}
```

Now that we've successfully integrated the component into our application, it's time to specify our pages. So, inside our pages directory, create a RepoSearch folder and two files, RepoSearch.js and RepoSearch.css. So simply import SearchForm and CardItemList components into the RepoSearch page, then within that Reposearch construct one useState hook named repos with an empty array as the initial value.

This hook will enable us to integrate the state into our functional component. useState(), unlike state in class components, does not work with object values. If necessary, we can use primitives directly and create multiple react hooks for multiple variables. const [state, setState] = useState(initialState);

and also remember that hooks in React must always be declared at the top of a function. This also aids in the preservation of state between all rendering for the component. Finally, let's develop the searchRepository function, which uses the free github api to get all project information and simply returns with the SearchForm and CardItemList components, passing the searchRepository function as a prop on a SearchForm component and repos in CardItemLists components.

```
//pages/RepoSearch
import React, { useState } from "react";
import axios from "axios";

import SearchForm from "../../components/SearchForm";
import CardItemList from "../../components/CardItemList";

import "./RepoSearch.css";

const RepoSearch = () => {
  const [repos, setRepos] = useState([]);

  const searchRepository = (searchQuery) => {
    setRepos([]);
    axios
      .get(

`https://api.github.com/search/repositories?q=${searchQuery}{&p
age,per_page,sort,order}`
      )
      .then((result) => setRepos(result.data.items));
  };

  return (
    <div className="container__RepoSearch">
      <SearchForm
        placeholder="Search Projects."
        buttonText="Search"
        onSubmit={(value) => searchRepository(value)}
      />
      <CardItemList items={repos} />
    </div>
  );
};

export default RepoSearch;
```

Don't forget to apply the following style to it as well.

```
.container__RepoSearch {
  display: flex;
  flex-direction: column;
  align-items: center;
}
```

Finally, let's build a functionality to generate notes in our application. To do so, construct two state notes and items,

```
const [notes,setNotes] = useState([]);
 const [items,setItems] = useState("");
```

Then return to the code and implement the useEffect functionality. By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. To this effect, we set the document title, but we could also perform data fetching or call some other imperative API. Placing useEffect() inside the component lets us access the count state variable (or any props) right from the effect. We don't need a special API to read it — it's already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript

already provides a solution.useEffect() hook is somewhat similar to the life-cycle methods that we are aware of for class components. It runs after every render of the component including the initial render. Hence it can be thought of as a combination of componentDidMount, componentDidUpdate, and componentWillUnmount.If we want to control the behavior of when the effect should run (only on initial render, or only when a particular state variable changes), we can pass in dependencies to the effect to do so. This hook also provides a clean-up option to allow cleaning up of resources before the component is destroyed. basic syntax of the effect:useEffect(didUpdate);

Here, didUpdate is a function that performs mutations, subscriptions, timers, logging, etc. It will get triggered after the component is rendered to the screen as well as on every subsequently completed render.

```
useEffect(() => {
  axios.get("http://localhost:4000").then((response) => {
    let data = [];
    for (var i = 0; i < response.data.data.length; i++) {
      data.push(response.data.data[i].note);
    }
    setNotes(data);
  });
}, []);
```

and after implementing useEffect functionality create two function called clickHandler and changeHandler .

```
const changeHandler = (e) => {
  setItems(e.target.value);
};
```

and

```
const clickHandler = async (e) => {
  axios({
    method: "post",
    url: "http://localhost:4000",
    data: {
      note: items,
    },
  })
    .then(() => {
      setItems("");
    })
    .then(() => {
      window.location.reload(false);
    });
};
```

After you've built everything you need, simply return the following statement.

```
return (
  <div className="conatiner__back">
    <input
      className="todo-input"
      placeholder="Organize and keep track of newly
explored awesome projects."
      type="text"
      onChange={changeHandler}
    />
    <button className="todo-button" type="submit"
onClick={clickHandler}>
      ➕ Add Notes
    </button>
    <small style={{ color: "white", fontSize: "10px" }}>
      * all notes will get stored inside MongoDB
    </small>
    <div className="notes__layout">
      <ol className="gradient-list">
        {notes.map((note) => (
          <li key={note._id}>{note}</li>
        ))}
      </ol>
    </div>
  </div>
);
```

This is how your final code should look like.

```
 // pages/TakeNote.js
import React, { useState, useEffect } from "react";
import axios from "axios";
import "./TakeNote.css";

const TakeNote = () => {
  const [notes, setNotes] = useState([]);
  const [items, setItems] = useState("");
  const changeHandler = (e) => {
    setItems(e.target.value);
  };
  const clickHandler = async (e) => {
    axios({
      method: "post",
      url: "http://localhost:4000",
      data: {
        note: items,
      },
    })
      .then(() => {
        setItems("");
      })
      .then(() => {
        window.location.reload(false);
      });
  };

  useEffect(() => {
    axios.get("http://localhost:4000").then((response) => {
      let data = [];
      for (var i = 0; i < response.data.data.length; i++) {
        data.push(response.data.data[i].note);
      }
      setNotes(data);
```

```
    });
  }, []);

  return (
    <div className="conatiner__back">
      <input
        className="todo-input"
        placeholder="Organize and keep track of newly explored
awesome projects."
        type="text"
        onChange={changeHandler}
      />
      <button className="todo-button" type="submit"
onClick={clickHandler}>
        ➕ Add Notes
      </button>
      <small style={{ color: "white", fontSize: "10px" }}>
        * all notes will get stored inside MongoDB
      </small>
      <div className="notes__layout">
        <ol className="gradient-list">
          {notes.map((note) => (
            <li key={note._id}>{note}</li>
          ))}
        </ol>
      </div>
    </div>
  );
};

export default TakeNote;
```

Lastly, make a TakeNote.css file inside the TakeNote and add the styles listed below to it.

```
// TakeNote.css
@import
url("https://fonts.googleapis.com/css2?family=Finger+Paint&disp
lay=swap");

.conatiner__back {
  text-align: center;
  background-color: transparent;
}

.todo-button {
  background-image: linear-gradient(
    to right,
    #02aab0 0%,
    #00cdac 51%,
    #02aab0 100%
  );
}
.todo-button {
  margin: 0 auto;
  padding: 10px 100px;
  margin-top: 10px;
  text-align: center;
  text-transform: uppercase;
  transition: 0.5s;
  background-size: 200% auto;
  color: white;
  box-shadow: 0 0 20px #eee;
  border-radius: 10px;
  display: block;
  outline: none;
}
```

```
.todo-button:hover {
  background-position: right center;
  color: #fff;
  text-decoration: none;
}

.todo-input {
  color: white;
  font-size: 1.2rem;
  font-family: "Finger Paint", cursive;
  margin: 0 auto;
  padding: 1rem 0.5rem;
  border-radius: 0.6rem;
  /* background-color: rgb(255, 255, 255); */
  background: url("../../assets/gifs/inputBack.gif");
  -webkit-background-size: cover;
  -moz-background-size: cover;
  -o-background-size: cover;
  background-size: cover;
  border: none;
  width: 50rem;
  display: block;
  border-bottom: 1rem solid transparent;
  transition: all 0.3s;
  outline: none;
}

@media only screen and (max-width: 900px) {
  .todo-input {
    color: #333;
    font-size: 1.2rem;
    margin: 0 auto;
    padding: 1rem 0.5rem;
    border-radius: 0.6rem;
    background-color: rgb(255, 255, 255);
```

```
    border: none;
    width: 100%;
    display: block;
    border-bottom: 1rem solid transparent;
    transition: all 0.3s;
  }
}

/* ------------------------------------------------ */

ol.gradient-list > li::before,
ol.gradient-list > li {
  box-shadow: 0.25rem 0.25rem 0.6rem rgba(0, 0, 0, 0.05),
    0 0.5rem 1.125rem rgba(75, 0, 0, 0.05);
}

/*** STYLE ***/
*,
*:before,
*:after {
  box-sizing: border-box;
}

.notes__layout {
  display: block;
  margin: 0 auto;
  max-width: 40rem;
  padding: 1rem;
}

ol.gradient-list {
  list-style: none;
  margin: 1.75rem 0;
  padding-left: 1rem;
}
ol.gradient-list > li {
```

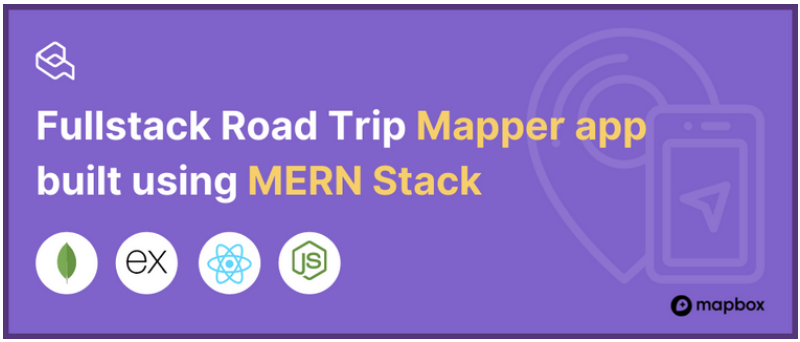
```
  background: white;
  text-align: left;
  font-family: "Finger Paint", cursive;
  border-radius: 0 0.5rem 0.5rem 0.5rem;
  counter-increment: gradient-counter;
  margin-top: 2rem;
  min-height: 3rem;
  border-radius: 20px;
  padding: 1rem 1rem 1rem 3rem;
  position: relative;
}
ol.gradient-list > li::before,
ol.gradient-list > li::after {
  background: linear-gradient(90deg, #83e4e2 0%, #a2ed56 100%);
  border-radius: 5rem 5rem 0 5rem;
  content: "🔖";
  height: 2.5rem;
  left: -1rem;
  overflow: hidden;
  position: absolute;
  top: -2rem;
  width: 3rem;
}
ol.gradient-list > li::before {
  align-items: flex-end;
  content: counter(gradient-counter);
  color: #1d1f20;
  display: flex;
  font: 1000 1.5em/1 "Montserrat";
  justify-content: center;

  justify-content: flex-end;
  padding: 0.125em 0.25em;
  z-index: 1;
}
ol.gradient-list > li:nth-child(10n + 1):before {
```

```
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.2) 0%,
    rgba(253, 220, 50, 0.2) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 2):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.4) 0%,
    rgba(253, 220, 50, 0.4) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 3):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.6) 0%,
    rgba(253, 220, 50, 0.6) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 4):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.8) 0%,
    rgba(253, 220, 50, 0.8) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 5):before {
  background: linear-gradient(135deg, #a2ed56 0%, #fddc32
100%);
}
ol.gradient-list > li:nth-child(10n + 6):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.8) 0%,
    rgba(253, 220, 50, 0.8) 100%
```

```
  );
}
ol.gradient-list > li:nth-child(10n + 7):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.6) 0%,
    rgba(253, 220, 50, 0.6) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 8):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.4) 0%,
    rgba(253, 220, 50, 0.4) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 9):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0.2) 0%,
    rgba(253, 220, 50, 0.2) 100%
  );
}
ol.gradient-list > li:nth-child(10n + 10):before {
  background: linear-gradient(
    135deg,
    rgba(162, 237, 86, 0) 0%,
    rgba(253, 220, 50, 0) 100%
  );
}
ol.gradient-list > li + li {
  margin-top: 2rem;
}
```

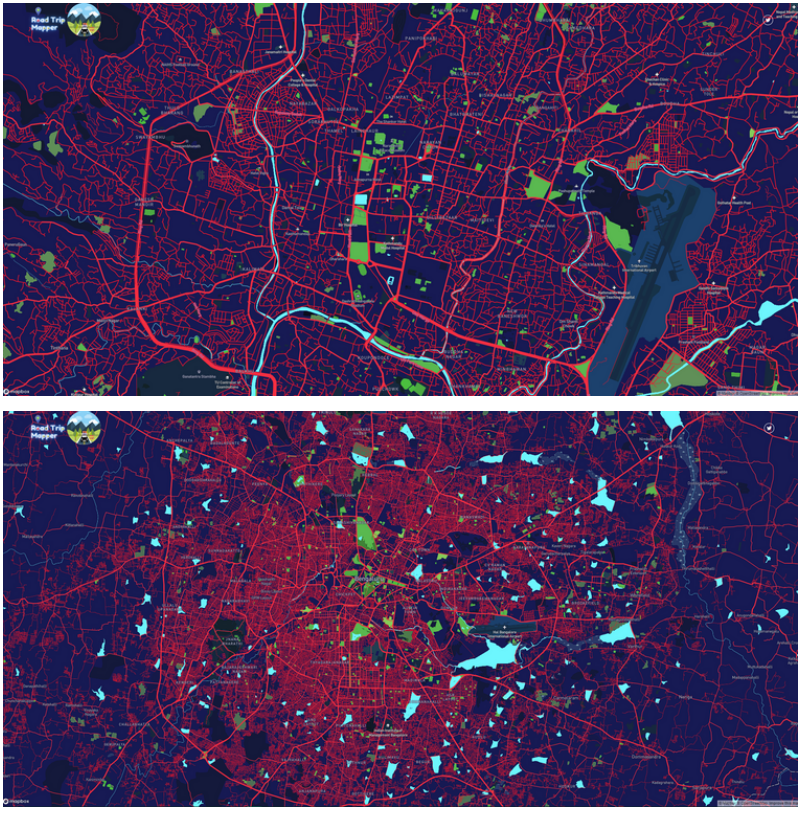
The application's full source code is available here.

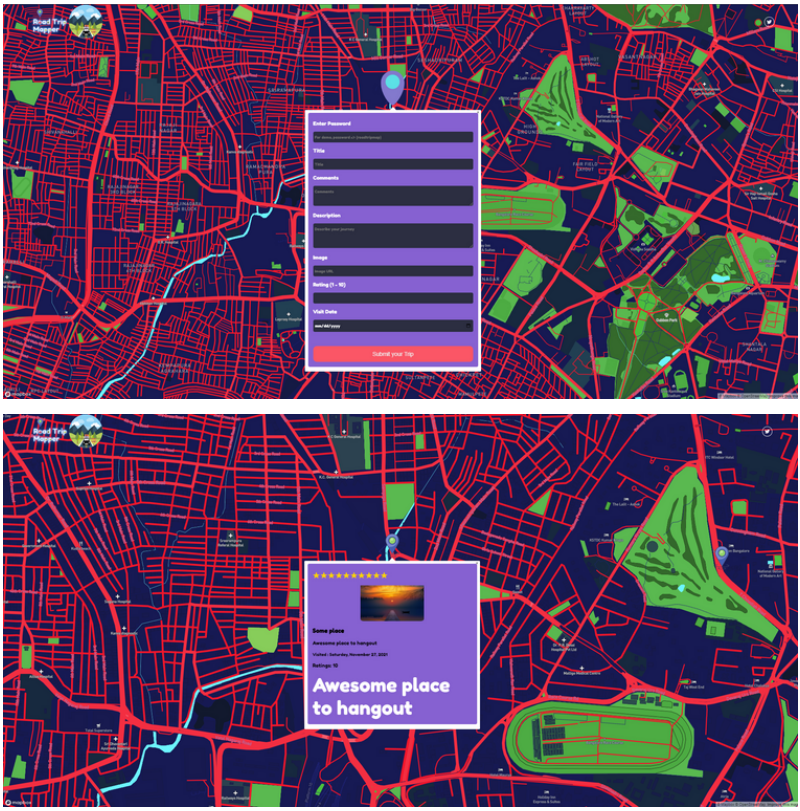
https://github.com/pramit-marattha/project-case-study-mern-app



This section concentrates on the most critical tasks and concepts for better understanding and building MERN stack applications from the ground up. It's for folks who are serious about learning about the MERN stack and want to concentrate on the essentials. We'll build a full-stack road trip mapper application where users can pin and map locations and view the sites pinned by other users, all using the MERN stack and leveraging the power of the Mapbox API. This blog session will teach you the fundamentals of MERN stack technology as well as advanced concepts and operations.

Here's a quick preview of our application's final version:



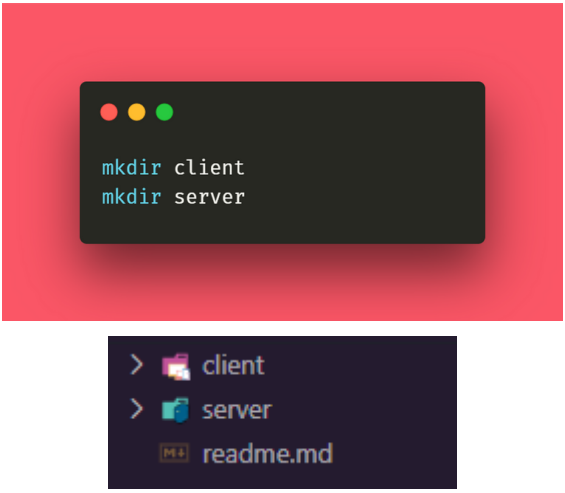


There is a separate article where you may learn about the MERN stack in very great detail.

https://aviyel.com/post/1278/crafting-a-stunning-crud-application-with-mern-stack

Setting up the folder structure

Create two folders inside your project directory called client and server, then open them in Visual Studio Code or any other code editor of your choice.



Now, we'll create a MongoDB database, a Node and Express server, a database schema to represent our project case study application, and API routes to create, read, update, and delete data and information from the database using npm and the appropriate packages. So, open a command prompt, navigate to your server's directory, and then run the code below.

```
npm init -yes
```

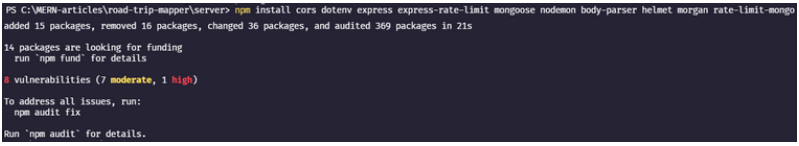
Configuring package.json file

Execute the following commands in the terminal to install the dependencies.

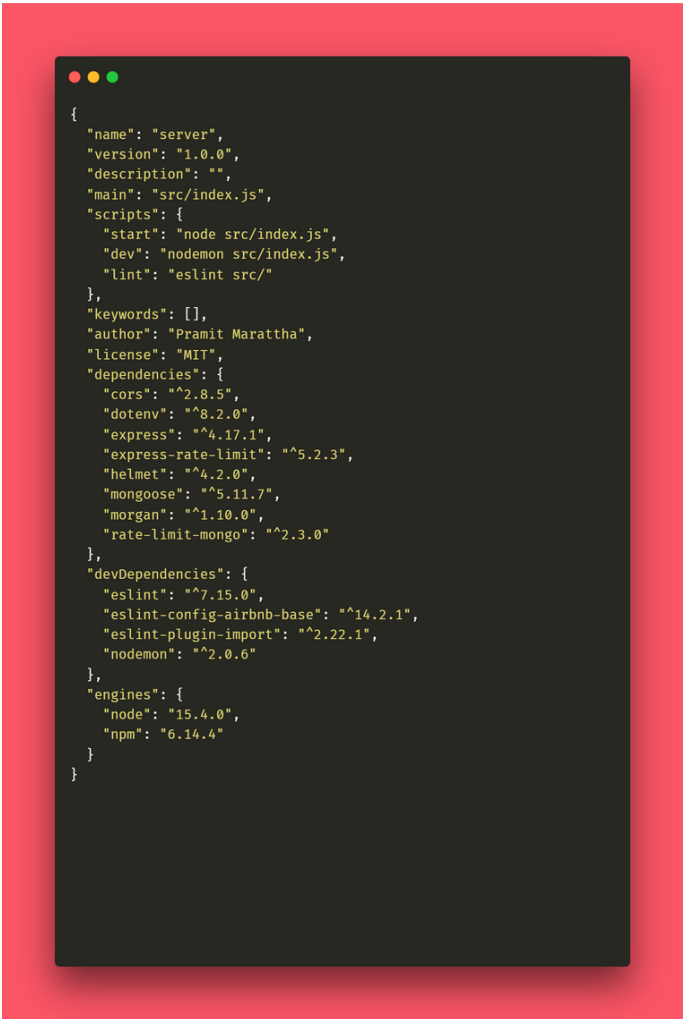
```
 npm install cors dotenv express express-rate-limit mongoose
nodemon body-parser helmet morgan rate-limit-mongo
```

- **Dotenv:** Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env
- **cors:** This module allows to relax the security applied to an API
- **express:** Fast, unopinionated, minimalist web framework for node.
- **express-rate-limit:** Basic IP rate-limiting middleware for Express. It is used to limit repeated requests to public APIs and/or endpoints such as password reset.
- **mongoose:** It is an Object Data Modeling library for MongoDB and Node. js

- **nodemon:** This module helps to develop node.js based applications by automatically restarting the application when file changes in the directory are detected.
- **body-parser:** Node.js body parsing middleware.
- **helmet:** Helmet.js fills in the gap between Node.js and Express.js by securing HTTP headers that are returned by Express applications.
- **morgan :** HTTP request logger middleware for node.js
- **rate-limit-mongo :** MongoDB store for the express-rate-limit middleware.



The "**package.json**" file should look like this after the dependencies have been installed.



```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "src/index.js",
  "scripts": {
    "start": "node src/index.js",
    "dev": "nodemon src/index.js",
    "lint": "eslint src/"
  },
  "keywords": [],
  "author": "Pramit Marattha",
  "license": "MIT",
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^8.2.0",
    "express": "^4.17.1",
    "express-rate-limit": "^5.2.3",
    "helmet": "^4.2.0",
    "mongoose": "^5.11.7",
    "morgan": "^1.10.0",
    "rate-limit-mongo": "^2.3.0"
  },
  "devDependencies": {
    "eslint": "^7.15.0",
    "eslint-config-airbnb-base": "^14.2.1",
    "eslint-plugin-import": "^2.22.1",
    "nodemon": "^2.0.6"
  },
  "engines": {
    "node": "15.4.0",
    "npm": "6.14.4"
  }
}
```

And also, remember to update the scripts as well.

Now go to your server directory, create an src folder, and an index.js file there.

Setting up index.js

- Import express module.
- Import and configure dotenv module
- Import helmet module.
- Import morgan module.
- Import CORS module
- Use express() to initialize our app.

```
//src/index.js
const express = require('express');
// NOTE morgan is a logger
const morgan = require('morgan');
const helmet = require('helmet');
const cors = require('cors');
const mongoose = require('mongoose');
require('dotenv').config();
// app config
const app = express();
```

We may now utilize all of the other methods on that app instance. Let's start with the fundamentals and very basic setups. Don't forget to set up the port and cors, too.

```
const express = require('express');
// NOTE morgan is a logger
const morgan = require('morgan');
const helmet = require('helmet');
const cors = require('cors');
const mongoose = require('mongoose');
require('dotenv').config();
const app = express();
const port = process.env.PORT || 4000;
app.use(morgan('common'));
app.use(helmet());
app.use(cors({
  origin: process.env.CORS_ORIGIN,
}));
app.use(express.json());
app.get('/', (req, res) => {
  res.json({
    message: 'Hello There',
  })});
```

Now it's time to connect our server application to a real database. Here we'll use the MongoDB database, specifically the MongoDB cloud Atlas version, which means our database will be hosted on their cloud.

Setting up MongoDB Atlas cloud cluster

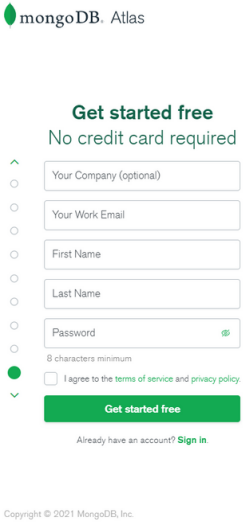
MongoDB is a document-oriented database that is open-source and cross-platform. MongoDB is a NoSQL database that stores data in JSON-like documents and has optional schemas. All versions were given under the AGPL license prior to October 16, 2018. All versions released after October 16, 2018, including bug fixes for prior versions, are covered by the SSPL license v1. You can also learn more about MongoDB setup and configuration from the following article.

https://aviyel.com/post/1323/building-a-calorie-journal-saas-based-project-using-mern-stack

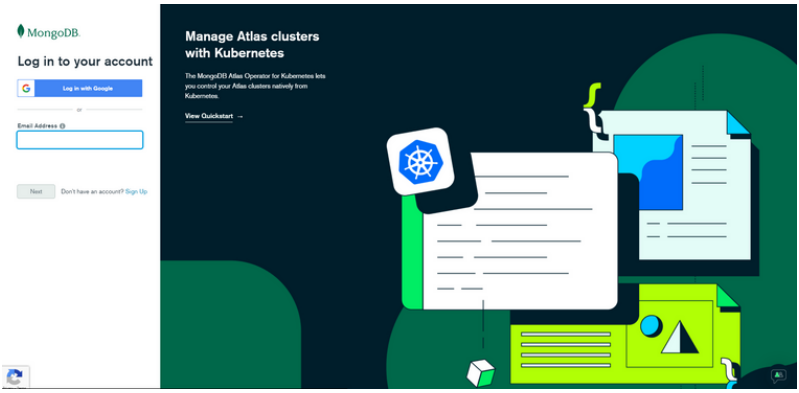
To set up and start your MongoDB cluster, follow the exact same steps mentioned below.

Official MongoDB website

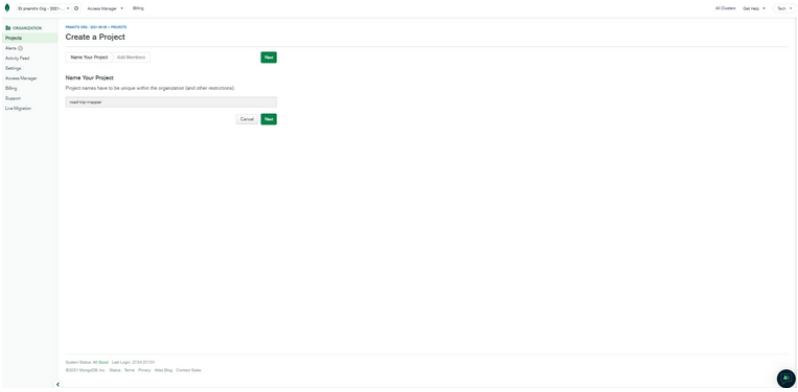
Sign up to MongoDB



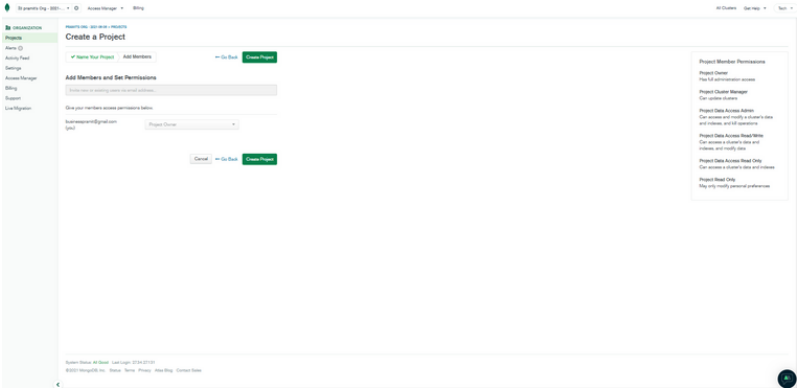
Sign in to MongoDB



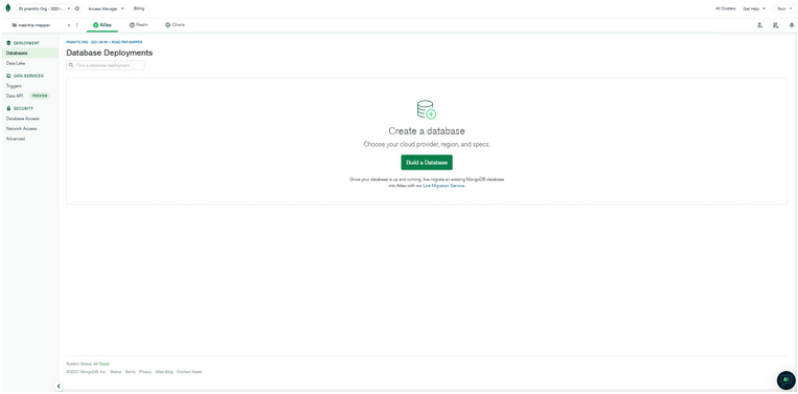
Create a project



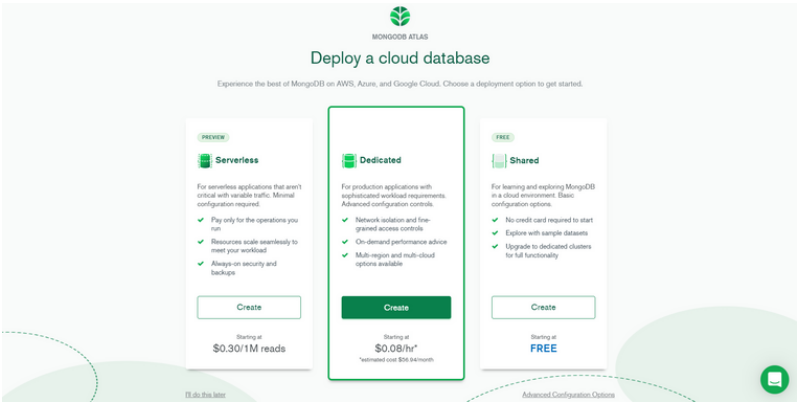
Adding members



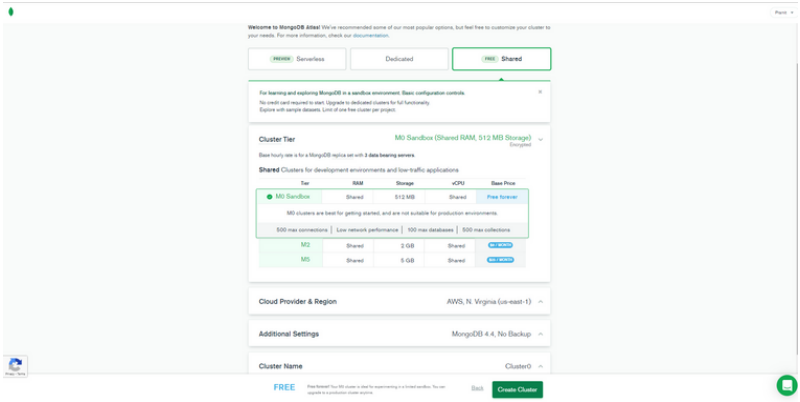
Building a database



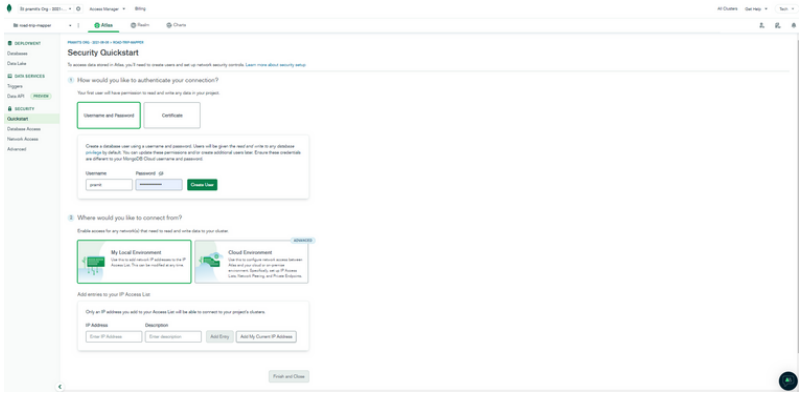
Creating a cluster



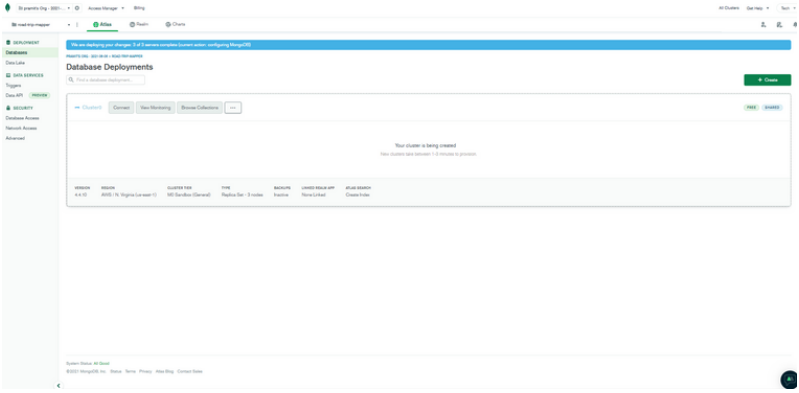
Selecting a cloud service provider



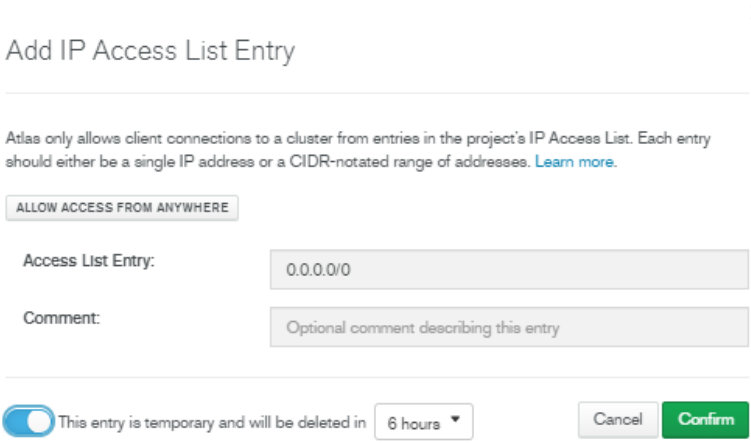
Configuring security



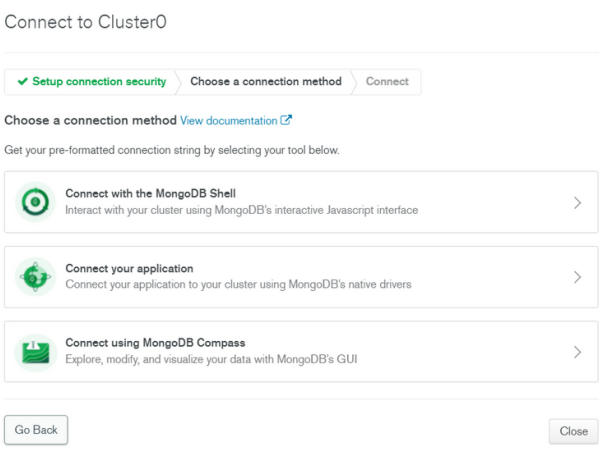
Database deployment to the cloud.



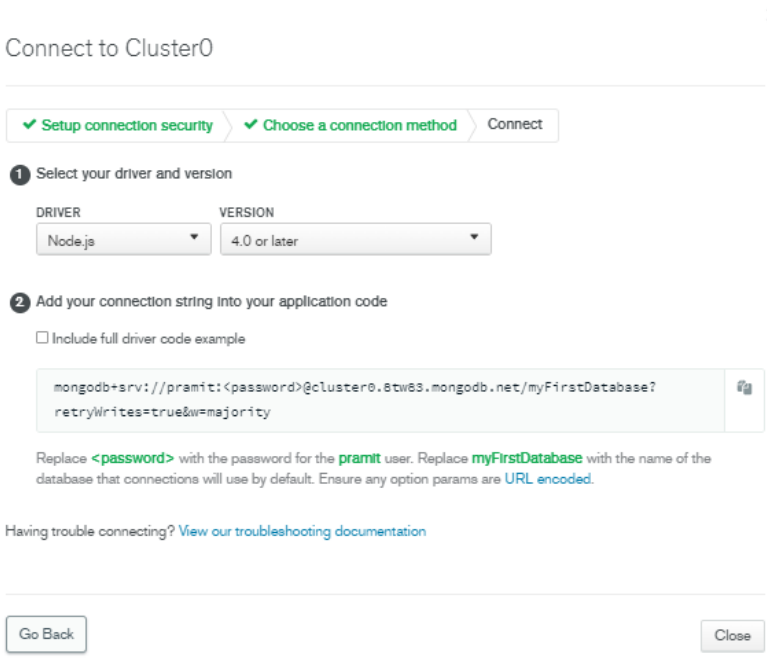
Navigate to the network access tab and select "Add IP address."



Now, select the Choose a connection method.



Connecting to cluster



Create a new variable called DATABASE_CONNECTION inside index.js . Create a string and paste the copied mongo DB connection URL into it. Now, inside it, type your username and password, removing any brackets and entering your own credentials. We'll create environmental variables to safeguard the credential later, but for now, let's add it this way. The second thing we'll need is a PORT, so just type in 4000 for now. Finally, we'll use mongoose to connect to our database, so type in mongoose. connect(), which is a function with two parameters. The DATABASE_CONNECTION will be the first, and the object

with two choices will be the second. The first is useNewUrlParser, which we'll enable, and the second is useUnifiedTopology, which we'll enable as well. These objects are optional, but we will see some errors or warnings on our console. Let's chain it with .then() and .catch() inside,then() function. This will simply call the app and invoke listen, leading to two parameters: PORT and the callback function that will be executed if our application is successfully connected to the database. Finally, if the connection to the database is unsuccessful, we will simply console log our error message. Your index.js file should now look something like this.

```
//src/index.js
const express = require('express');
// NOTE morgan is a logger
const morgan = require('morgan');
const helmet = require('helmet');
const cors = require('cors');
const mongoose = require('mongoose');

require('dotenv').config();

const app = express();

const DATABASE_CONNECTION = process.env.DATABASE_URL;




mongoose.connect(DATABASE_CONNECTION, {
  useNewUrlParser: true,
```

```
  newUnifiedTopology: true,
});

app.use(morgan('common'));
app.use(helmet());
app.use(cors({
  origin: process.env.CORS_ORIGIN,
}));

app.use(express.json());

app.get('/', (req, res) => {
  res.json({
    message: 'Hello There',
  });
});

const port = process.env.PORT || 4000;
app.listen(port, () => {
  console.log(`Currently Listening at
http://localhost:${port}`);
});
```

Insert mongodb+srv into the .env **file.**

```
PORT=4000
DATABASE_URL=mongodb+srv://pramit:<password>@cluster0.8tw83.mon
godb.net/myFirstDatabase?retryWrites=true&w=majority
CORS_ORIGIN=http://localhost:3000
```

We now have successfully connected our server to the database, let's create middleware first before we get started on building our backend application's routes and database schema. To do so, we'll need to create a new file called

middlewares.js and within that file, we will create a two function called notFound and errorHandler

and export those functions. So let create notFound middleware so typically this middleware should be the last middleware that is registered so this middleware takes in req, res, and next. Basically, if a request ever made it here, it means we didn't locate the route users were searching for, so we'll create a variable and send them a message, and then we'll pass that to our next middleware, which is errorHander Middleware but before that don't forget to pass the response status of 404 as well. Now let's make our errorHandler middleware, which has four parameters instead of three, so we'll have (error,req, res, next). The first thing we'll do is set a status code and check whether it's 200 or use the status code that's already been specified, and then we'll simply set the status code, and then we'll respond with some JSON that will display the error message.

```
//middlewares.js
const notFound = (req, res, next) => {
  const error = new Error(`Not Found - ${req.originalUrl}`);
  res.status(404);
  next(error);
};


const errorHandler = (error, req, res, next) => {
  const statusCode = res.statusCode === 200 ? 500 :
res.statusCode;
```

```
  res.status(statusCode);
  res.json({
    message: error.message,
    stack: process.env.NODE_ENV === "production" ? "nope" :
error.stack,
  });
};

module.exports = {
  notFound,
  errorHandler,
};
```

So, after modifying the middlewares.js file, import and use the middleware as needed in the index.js file.

```
//src/index.js
const express = require("express");
// NOTE morgan is a logger
const morgan = require("morgan");
const helmet = require("helmet");
const cors = require("cors");
const mongoose = require("mongoose");

require("dotenv").config();

const middlewares = require("./middlewares");
const app = express();

const DATABASE_CONNECTION = process.env.DATABASE_URL;

mongoose.connect(DATABASE_CONNECTION, {
  useNewUrlParser: true,
  newUnifiedTopology: true,
```

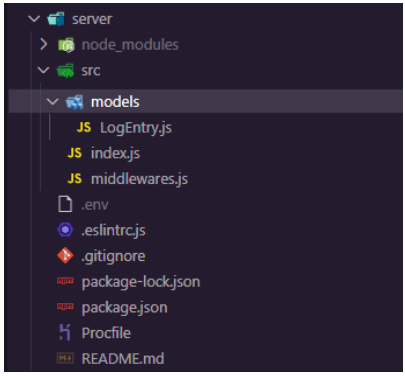
```
});

app.use(morgan("common"));
app.use(helmet());
app.use(
  cors({
    origin: process.env.CORS_ORIGIN,
  })
);

app.use(express.json());

app.get("/", (req, res) => {
  res.json({
    message: "Hello There",
  });
});

app.use(middlewares.notFound);
app.use(middlewares.errorHandler);

const port = process.env.PORT || 4000;
app.listen(port, () => {
  console.log(`Currently Listening at
http://localhost:${port}`);
});
```

let's create a LogEntry model. Create a folder called models and inside it, create one file called LogEntry.model.js and within that following file structure your DB schema by defining title, description, comments, image, ratings, latitude and longitude as shown below.

```
//models/LogEntry.model.js
const mongoose = require("mongoose");
const { Schema } = mongoose;

const requiredNumber = {
  type: Number,
  required: true,
};

const logEntrySchema = new Schema(
  {
    title: {
      type: String,
      required: true,
    },
    description: String,
    comments: String,
    image: String,
    rating: {
      type: Number,
      min: 0,
      max: 10,
      default: 0,
    },
    latitude: {
      ...requiredNumber,
      min: -90,
      max: 90,
    },
    longitude: {
      ...requiredNumber,
      min: -180,
```

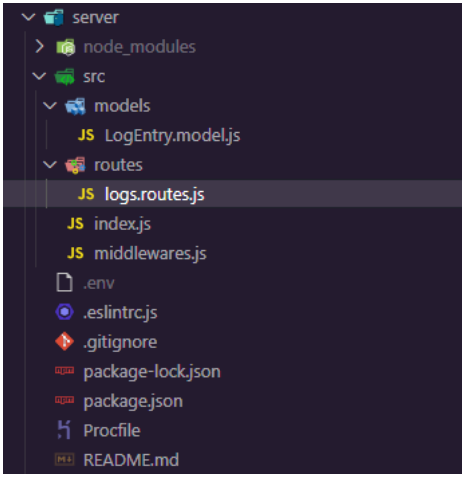
```
      max: 180,
    },
    visitDate: {
      required: true,
      type: Date,
    },
  },
  {
    timestamps: true,
  }
);
const LogEntry = mongoose.model("collections",
logEntrySchema);
module.exports = LogEntry;
```

The structure of your files and folders should now look like this.



Now that we've successfully created our DB Schema, let's get started on creating our routes for our backend application. To do so, we'll need to create a new folder inside the src directory

and name it as routes Within the routes folder, we will create a js file called logs.routes.js.so first we must import express from "express" and also configure our router and import our recently created DB schema. Now we can begin adding our routes to it.




```
const { Router } = require("express");

const LogEntry = require("../models/LogEntry.model.js");

const { API_KEY } = process.env;

const router = Router();
```

fetches all the pinned location information.

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

```
  try {
    const entries = await LogEntry.find();
    res.json(entries);
  } catch (error) {
    next(error);
  }
});
```

Insert/add a pinned location with authorized access

```
router.post("/", async (req, res, next) => {
  try {
    if (req.get("X-API-KEY") !== API_KEY) {
      res.status(401);
      throw new Error("Unauthorized Access");
    }
    const logEntry = new LogEntry(req.body);
    const createdEntry = await logEntry.save();
    res.json(createdEntry);
  } catch (error) {
    if (error.name === "ValidationError") {
      res.status(422);
    }
    next(error);
  }
});
```

exporting the router

```
module.exports = router;
```

Your logs.routes.js should resemble something like this

```
//src/routes/logs.routes.js
const { Router } = require("express");
const LogEntry = require("../models/LogEntry.model.js");
const { API_KEY } = process.env;
const router = Router();
router.get("/", async (req, res, next) => {
  try {
    const entries = await LogEntry.find();
    res.json(entries);
  } catch (error) {
    next(error);
  }
});
router.post("/", async (req, res, next) => {
  try {
    if (req.get("X-API-KEY") !== API_KEY) {
      res.status(401);
      throw new Error("Unauthorized Access")}
    const logEntry = new LogEntry(req.body);
    const createdEntry = await logEntry.save();
    res.json(createdEntry);
  } catch (error) {
    if (error.name === "ValidationError") {
      res.status(422);
    }
    next(error);
  }});
module.exports = router;
```

Now, update your .env **file**

```
NODE_ENV=production
PORT=4000
DATABASE_URL=mongodb+srv://pramit:<password>@cluster0.8tw83.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
CORS_ORIGIN=http://localhost:3000
API_KEY=roadtripmapper
```

Let's get started by importing the logs routes into your index.js file. We can now connect map pinned log info to our application using express middleware. Finally, your root index.js file should be like the following.

```
//src/index.js
const express = require("express");
// NOTE morgan is a logger
const morgan = require("morgan");
const helmet = require("helmet");
const cors = require("cors");
const mongoose = require("mongoose");

require("dotenv").config();

const middlewares = require("./middlewares");
const logs = require("./routes/logs.routes.js");
const app = express();

const DATABASE_CONNECTION = process.env.DATABASE_URL;


mongoose.connect(DATABASE_CONNECTION, {
  useNewUrlParser: true,
  newUnifiedTopology: true,
```

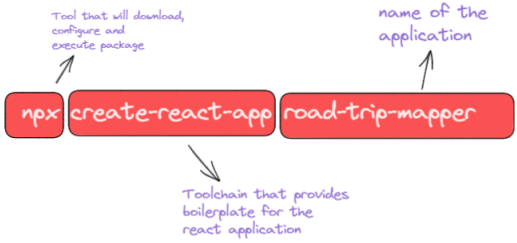
```
});

app.use(morgan("common"));
app.use(helmet());
app.use(
  cors({
    origin: process.env.CORS_ORIGIN,
  })
);

app.use(express.json());

app.get("/", (req, res) => {
  res.json({
    message: "Hello There",
  });
});

app.use("/api/logs", logs);

app.use(middlewares.notFound);
app.use(middlewares.errorHandler);

const port = process.env.PORT || 4000;
app.listen(port, () => {
            console.log(`Currently      Listening      at
http://localhost:${port}`);
});
```

After restarting the server, you should see something like this:

```
PS C:\MERN-articles\road-trip-mapper\server> npm start

> server@1.0.0 start
> node src/index.js

Currently Listening at http://localhost:4000
```

Setting up the frontend with react

In the next step let's start with the frontend and build it with react. The first thing you need to do is install Node.js if it isn't already installed on your machine. So, go to the Node.js official website and download the most recent version. You'll require Node js to utilize the node package manager, generally known as NPM. Now navigate to the client folder in your favorite code editor. Visual Studio Code will be my tool of choice. Then, in the integrated terminal, type npx create-react-app. This command will create a client application in the current directory with the name client.



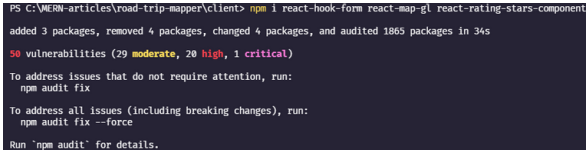
There is a separate article where you may learn everything there is to know about cleaning up boilerplate react projects.

https://aviyel.com/post/1190/building-a-react-application-from-absolute-scratch

It's time to install some packages within react-boilerplate now that you've installed and cleaned it. so copy and paste the following command into your terminal.

```
npm i react-hook-form react-map-gl react-rating-stars-component
react-responsive-animate-navbar
```

- **react-hook-form:** Performant, flexible, and extensible forms library for React Hooks.
- **react-map-gl:** react-map-gl is a suite of React components designed to provide a React API for Mapbox GL JS-compatible libraries
- **react-rating-stars-component:** Simple star rating component for your React projects.
- **react-responsive-animate-navbar :** simple, flexible & completely customisable responsive navigation bar component.

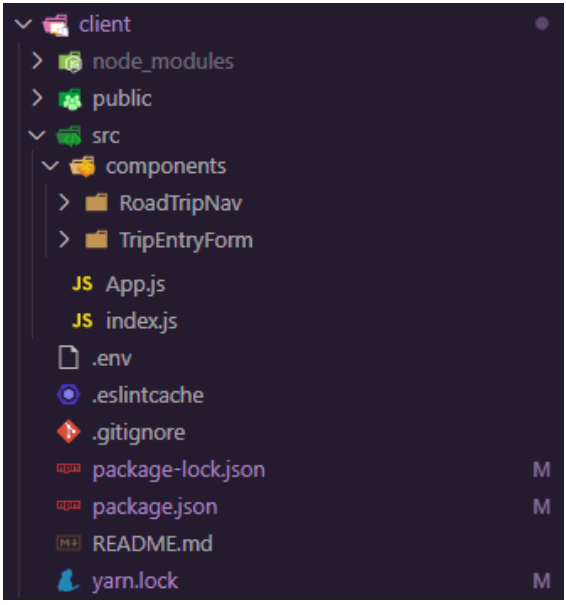


After installing all these packages your **packge.json** file of the client should look like this:

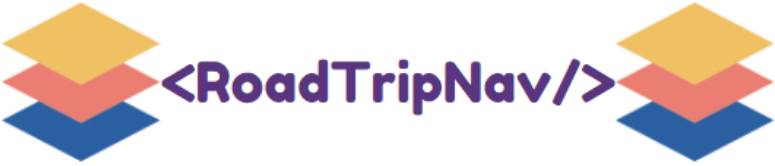
```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-hook-form": "^6.15.8",
    "react-map-gl": "^5.3.17",
    "react-rating-stars-component": "^2.2.0",
    "react-responsive-animate-navbar": "^1.1.8",
    "react-scripts": "4.0.1",
    "web-vitals": "^0.2.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Let's construct two separate folders /components inside the components folder after we've installed all of our project's dependencies and name it as RoadTripNav and TripEntryForm

Your file and folder structure should look something like this once you've added all of your components.

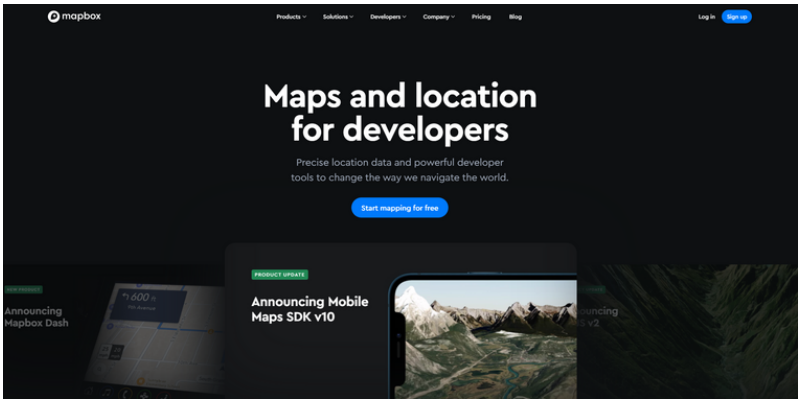


Now that you have all of the project's components set up, it's time to start coding. First, import the ReactNavbar from "react-responsive-animate-navbar" and customize the color of your navbar, add the logo to the public folder and import it directly, and don't forget to add some social links as well. The following is an example of how the code should appear.



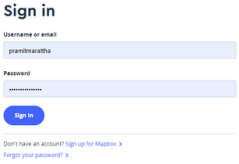
```
// components/RoadTripNav
import React from "react";
import * as ReactNavbar from
"react-responsive-animate-navbar";
// import roadTripSvg from "../../assets/roadtrip.svg";

const RoadTripNav = () => {
  return (
    <ReactNavbar.ReactNavbar
      color="rgb(25, 25, 25)"
      logo="./logo.svg"
      menu={[]}
      social={[
        {
          name: "Twitter",
          url: "https://twitter.com/pramit_armpit",
          icon: ["fab", "twitter"],
        },
      ]}
    />
  );
};

export default RoadTripNav;
```

Before we go any further, let's set up our Mapbox. First, go to the Mapbox site and log in or sign up if you don't already have an account. Next, create your own custom map style in the Mapbox Studio and publish it. Finally, go back to the dashboard and copy the default public API key provided by MapBox.

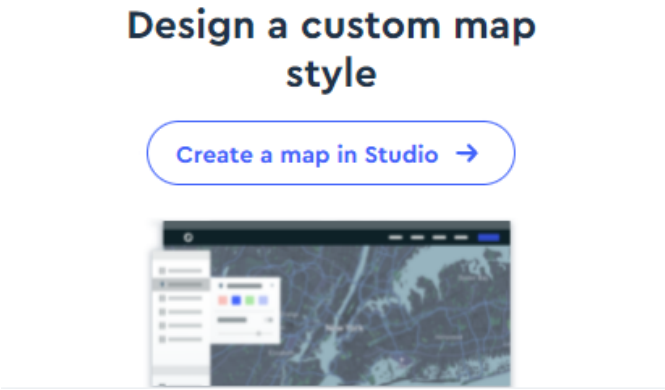


Login or create your MapBox account

mapbox

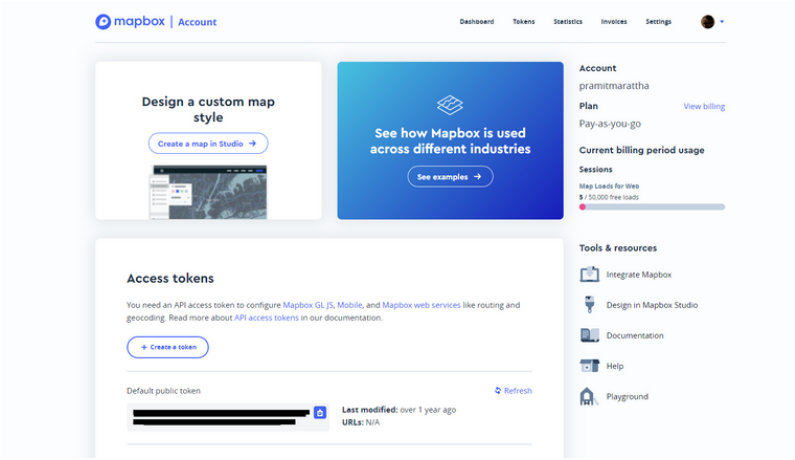


Click on design a custom map style



Customize your own style of the map inside the Mapbox studio

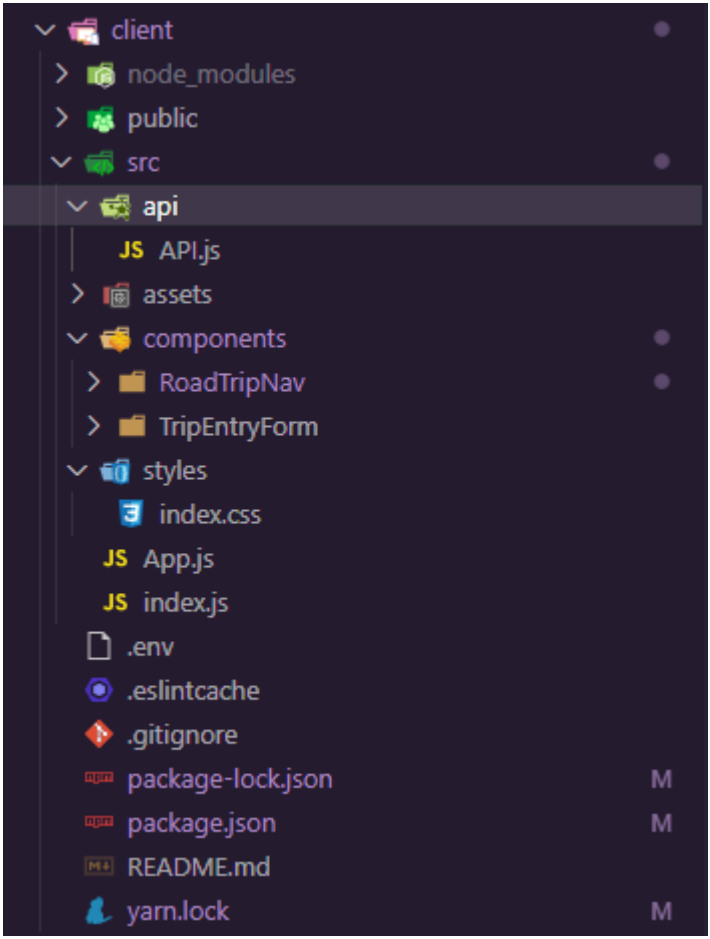
Copy the default public token



After you've successfully obtained your public token, go to the env file or create one if you don't have and after that create a variable named as REACT_APP_MAPBOX_TOKEN, then paste that token into that variable. This is what your env file should look like.

```
REACT_APP_MAPBOX_TOKEN=
********************************** // add token
```

Before we go any further, let's make an api and styles folder in our root source directory. Inside the api folder, make a API.js file, and inside the styles folder, make a index.css file where all our styles of the application will be added. This is how your folder structure should appear.



Now go to the newly created API file and construct two functions called "listLogEntries" to collect all the log entries from the backend and "createLogEntries" to create or send the post request / post the entries to the backend, as well as

export these functions. Also, don't forget to include the URL where your server is running.


```
//api/API.js
const API_URL = "http://localhost:4000";
// const API_URL = window.location.hostname === "localhost" ?
"http://localhost:4000" :
"https://road-trip-map-mern.herokuapp.com" ;

export async function listLogEntries() {
  const response = await fetch(`${API_URL}/api/logs`);
  // const json = await response.json();
  return response.json();
}

export async function createLogEntries(entry) {
  const api_key = entry.api_key;
  delete entry.api_key;
  const response = await fetch(`${API_URL}/api/logs`, {
    method: "POST",
    headers: {
      "content-type": "application/json",
      "X-API-KEY": api_key,
```

```
    },
    body: JSON.stringify(entry),
  });
  // const json = await response.json();
  // return response.json();
  let json;
  if
(response.headers.get("content-type").includes("text/html")) {
    const message = await response.text();
    json = {
      message,
    };
  } else {
    json = await response.json();
  }
  if (response.ok) {
    return json;
  }
  const error = new Error(json.message);
  error.response = json;
  throw error;
}
```

Let's make a submission form for the pinned map location. To do so, open the TripEntryForm component from the component folder we previously made, import the useForm hook from react-hook-form, import createLogentries from api, and then import the useState hook from the React library because this hook will enable us to integrate the state into our functional component. useState(), unlike state in class components, does not work with object values. If necessary, we can use primitives directly and create multiple react hooks for multiple variables. Now, create two states: loading and

error, and then destructure register and handleSubmit from the useForm() hook from "react-hook-form" library.After you've completed that, it's time to craft our form, but first let's create a function to handle our submit request. To do so, create an asynchronous onSubmit function and inside it, simply create a try-catch block. Inside the try block set the loading to true, configure the latitude and longitude, console log the data, and invoke the onClose function, and finally inside the catch block, pass the error message to the error state, set the loading to false and simply console log the error message and then simply create a form inside the return statement exactly shown in the code below.


```
// components/TripEntryForm.js
import React, { useState } from "react";
import { useForm } from "react-hook-form";
import { createLogEntries } from "../../api/API";
import "./TripEntryForm.css";

const TripEntryForm = ({ location, onClose }) => {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  const { register, handleSubmit } = useForm();

  const onSubmit = async (data) => {
```

```
    try {
      setLoading(true);
      data.latitude = location.latitude;
      data.longitude = location.longitude;
      const created = await createLogEntries(data);
      console.log(created);
      onClose();
    } catch (error) {
      setError(error.message);
      console.error(error);
      setLoading(false);
    }
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}
className="trip-form">
      {error ? <h3 className="error-message">{error}</h3> :
null}
      <label htmlFor="api_key">Enter Password</label>
      <input
        type="password"
        name="api_key"
        placeholder="For demo, password => {roadtripmap} "
        required
        ref={register}
      />

      <label htmlFor="title">Title</label>
      <input name="title" placeholder="Title" required
ref={register} />

      <label htmlFor="comments">Comments</label>
      <textarea
        name="comments"
        placeholder="Comments"
```

```
        rows={3}
        ref={register}
      ></textarea>

      <label htmlFor="description">Description</label>
      <textarea
        name="description"
        placeholder="Describe your journey"
        rows={4}
        ref={register}
      ></textarea>

      <label htmlFor="image">Image</label>
      <input name="image" placeholder="Image URL"
ref={register} />

      <label htmlFor="rating">Rating (1 - 10)</label>
      <input name="rating" type="number" min="0" max="10"
ref={register} />

      <label htmlFor="visitDate">Visit Date</label>
      <input name="visitDate" type="date" required
ref={register} />

      <button disabled={loading}>
        <span>{loading ? "Submitting..." : "Submit your
Trip"}</span>
      </button>
    </form>
  );
};

export default TripEntryForm;
```

Also, don't forget to add the TripEntryForm styles inside that very own component folder and name it as TripEntryForm.css and paste the exact CSS code as mentioned below


```
//TripEntryForm.css
@import
url("https://fonts.googleapis.com/css2?family=Fredoka+One&famil
y=Poppins:ital,wght@0,200;0,400;1,200;1,300&family=Roboto:ital,
wght@0,300;0,400;0,500;1,300;1,400;1,500&display=swap");

.trip-form label {
  margin: 0.5rem 0;
  display: block;
  width: 100%;
  color: rgb(255, 255, 255);
  font-family: "Fredoka One", cursive;
}
.trip-form input {
  margin: 0.5rem 0;
  background-color: #2c2e41;
  border-radius: 5px;
  border: 0;
  box-sizing: border-box;
  color: rgb(255, 255, 255);
  font-size: 12px;
```

```
  height: 100%;
  outline: 0;
  padding: 10px 5px 10px 5px;
  width: 100%;
  font-family: "Fredoka One", cursive;
}

.trip-form textarea {
  margin: 0.5rem 0;
  background-color: #2c2e41;
  border-radius: 5px;
  border: 0;
  box-sizing: border-box;
  color: rgb(255, 255, 255);
  font-size: 12px;
  height: 100%;
  outline: 0;
  padding: 10px 5px 10px 5px;
  width: 100%;
  font-family: "Fredoka One", cursive;
}

.error-message {
  color: red;
}

.trip-form button {
  background-color: #fb5666;
  border-radius: 12px;
  border: 0;
  box-sizing: border-box;
  color: #eee;
  cursor: pointer;
  font-size: 18px;
  height: 50px;
  margin-top: 38px;
```

```
  outline: 0;
  text-align: center;
  width: 100%;
}

button span {
  position: relative;
  z-index: 2;
}

button:after {
  position: absolute;
  content: "";
  top: 0;
  left: 0;
  width: 0;
  height: 100%;
  transition: all 2.35s;
}

button:hover {
  color: #fff;
}

button:hover:after {
  width: 100%;
}

.small_description {
  font-size: 60px;
}
```

Now go to this repo and download all of the SVG files that are available there.

https://github.com/pramit-marattha/road-trip-mapper-mern-app/tree/main/client/src/assets

After you've downloaded all of the svg files, go to the main app component, and begin importing all of the key requirements from the libraries we previously installed, such as ReactMapGl, marker, and popup from the "react-map-gl" library, import all of the components as well as svgs from the assets folder, and finally create four state logEntries whose initial value is empty array, showPopup whose initial value is an empty object, addEntryLocation has a default value of null, and for viewport specify the initial value exactly like the code mentioned below or you can add whatever you want. Create an asynchronous function called getEntries that asynchronously calls the listLogEntries function that was previously established within the api file and whose main task is to retrieve all of the entries made by the users and feed them to the logEntries state and then call that function inside the useEffect() hook by using this Hook, you tell React that your component needs to do something after render.

React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. To this effect, we set the document title, but we could also perform data fetching or call some other imperative API. Placing useEffect() inside the component lets

us access the count state variable (or any props) right from the effect. We don’t need a special API to read it — it’s already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution.useEffect() the hook is somewhat similar to the life-cycle methods that we are aware of for class components. It runs after every render of the component including the initial render. Hence it can be thought of as a combination of componentDidMount, componentDidUpdate, and componentWillUnmount.If we want to control the behavior of when the effect should run (only on initial render, or only when a particular state variable changes), we can pass in dependencies to the effect to do so. This hook also provides a clean-up option to allow cleaning up of resources before the component is destroyed. basic syntax of the effect: useEffect(didUpdate) .

Create a function named showMarkerPopup and provide the event parameters to it. Inside that function, destruct the latitude and longitude from "event.lngltd" and pass it to addEntryLocation state. Finally, employ all of the imported components within our return statement by simply following the code shown below.


```
//src/app.js
import * as React from "react";
import { useState, useEffect } from "react";
import ReactMapGL, { Marker, Popup } from "react-map-gl";
import { listLogEntries } from "./api/API";
import MapPinLogo from "./assets/mapperPin.svg";
import MarkerPopup from "./assets/MarkerPopup.svg";
import TripEntryForm from "./components/TripEntryForm";
import ReactStars from "react-rating-stars-component";
import RoadTripNav from
"./components/RoadTripNav/RoadTripNav";

const App = () => {
  const [logEntries, setLogEntries] = useState([]);
  const [showPopup, setShowPopup] = useState({});
  const [addEntryLocation, setAddEntryLocation] =
useState(null);
  const [viewport, setViewport] = useState({
    width: "100vw",
    height: "100vh",
    latitude: 27.7577,
    longitude: 85.3231324,
    zoom: 7,
  });

  const getEntries = async () => {
    const logEntries = await listLogEntries();
```

```
    setLogEntries(logEntries);
    console.log(logEntries);
  };

  useEffect(() => {
    getEntries();
  }, []);

  const showMarkerPopup = (event) => {
    console.log(event.lngLat);
    const [longitude, latitude] = event.lngLat;
    setAddEntryLocation({
      longitude,
      latitude,
    });
  };

  return (
    <>
      <RoadTripNav />
      <ReactMapGL
        {...viewport}

mapStyle="mapbox://styles/pramitmarattha/ckiovge5k3e7x17tcmydc
42s3"

mapboxApiAccessToken={process.env.REACT_APP_MAPBOX_TOKEN}
        onViewportChange={(nextViewport) =>
setViewport(nextViewport)}
        onDblClick={showMarkerPopup}
      >
        {logEntries.map((entry) => (
          <React.Fragment key={entry._id}>
            <Marker latitude={entry.latitude}
longitude={entry.longitude}>
              <div
```

```
            onClick={() =>
              setShowPopup({
                // ...showPopup,
                [entry._id]: true,
              })
            }
          >
            <img
              className="map-pin"
              style={{
                width: `${5 * viewport.zoom}px`,
                height: `${5 * viewport.zoom}px`,
              }}
              src={MapPinLogo}
              alt="Map Pin Logo"
            />
          </div>
        </Marker>
        {showPopup[entry._id] ? (
          <Popup
            latitude={entry.latitude}
            longitude={entry.longitude}
            closeButton={true}
            closeOnClick={false}
            dynamicPosition={true}
            onClose={() => setShowPopup({})}
            anchor="top"
          >
            <div className="popup">
              <ReactStars
                count={10}
                value={entry.rating}
                size={29}
                activeColor="#ffd700"
              />
              <div className="popup_image">
```

```
                    {entry.image && <img src={entry.image}
alt={entry.title} />}
                  </div>
                  <h3>{entry.title}</h3>
                  <p>{entry.comments}</p>
                  <small>
                    Visited :{" "}
                    {new
Date(entry.visitDate).toLocaleDateString("en-US", {
                      weekday: "long",
                      year: "numeric",
                      month: "long",
                      day: "numeric",
                    })}
                  </small>
                  <p>Ratings: {entry.rating}</p>
                  <div
className="small_description">{entry.description}</div>
                </div>
              </Popup>
            ) : null}
          </React.Fragment>
        ))}
        {addEntryLocation ? (
          <>
            <Marker
              latitude={addEntryLocation.latitude}
              longitude={addEntryLocation.longitude}
            >
              <div>
                <img
                  className="map-pin"
                  style={{
                    width: `${8 * viewport.zoom}px`,
                    height: `${8 * viewport.zoom}px`,
                  }}
```

```
                  src={MarkerPopup}
                  alt="Map Pin Logo"
                />
              </div>
              {/* <div
style={{color:"white"}}>{entry.title}</div> */}
            </Marker>

            <Popup
              latitude={addEntryLocation.latitude}
              longitude={addEntryLocation.longitude}
              closeButton={true}
              closeOnClick={false}
              dynamicPosition={true}
              onClose={() => setAddEntryLocation(null)}
              anchor="top"
            >
              <div className="popup">
                <TripEntryForm
                  onClose={() => {
                    setAddEntryLocation(null);
                    getEntries();
                  }}
                  location={addEntryLocation}
                />
              </div>
            </Popup>
          </>
        ) : null}
      </ReactMapGL>
    </>
  );
};
export default App;
```

The very final step is to add all of the styles to our project, which can be done by going to our previously established styles folder and copying and pasting the following mentioned code into the index.css file.


```
/* styles/index.css */
@import
url("https://fonts.googleapis.com/css2?family=Fredoka+One&family
=Poppins:ital,wght@0,200;0,400;1,200;1,300&family=Roboto:ital,wg
ht@0,300;0,400;0,500;1,300;1,400;1,500&display=swap");

body {
  margin: 0;
  font-family: "Fredoka One", cursive;
  height: 100vh;
  width: 100vw;
  overflow: hidden;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas,
"Courier New",
    monospace;
}
```

```
.map-pin {
  position: absolute;
  transform: translate(-50%, -100%);
  z-index: -1;
}

.popup {
  width: 20vw;
  height: auto;
  padding: 1rem;
  background-color: #8661d1;
  border-radius: 5px;
  z-index: 999;
}

.popup img {
  width: 40%;
  height: auto;
  border-radius: 5%;
  justify-content: center;
  align-items: center;
  margin: 0 auto;
  padding-top: 1rem;
}

.popup_image {
  display: flex;
  justify-content: center;
  align-items: center;
}

.small_description {
  font-size: 1.5rem;
  color: #fff;
  border-radius: 5px;
  z-index: 999;
```

```
}

button {
  border: none;
  color: #fa5252;
  padding-right: 1rem;
  border-radius: 50%;
  font-size: 4rem;
  margin-top: 0.2rem;
  height: auto;
  cursor: pointer;
}
```

Finally, start both the client and the server.

```
Compiled successfully!

You can now view client in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.221.2:3000

Note that the development build is not optimized.
To create a production build, use yarn build.
```

Application up and running

This application's entire source code is available here.

https://github.com/pramit-marattha/road-trip-mapper-mern-app

Conclusion

In this eBook, we successfully built a CRUD app , Blog app , project case study calorie journal Saas app , road trip mapper application . From here, we can be highly inventive and come up with a wide range of methods to improve these app while practising or mastering our MERN skills. If you found this to be rather simple, play around with the code and see what you can come up with.

Cheers,

Happy Coding !!