

Share on your Social Media



MERN Stack Tutorial for Web Development Aspirants

Published On: October 14, 2024

MERN Stack Tutorial for Web Development Aspirants

There is a growing need for competent MERN stack developers. Industry reports state that reputable companies need developers who understand every MERN (MongoDB, Express.JS, React.JS, and Node.JS) component. Gain expertise with the fundamentals in this MERN Stack tutorial designed for web development aspirants. Know the fundamentals of MERN Stack with our [JavaScript course in Chennai](#).

[Download MERN Stack Tutorial PDF](#)

Introduction to MERN Stack

MERN Stack is a set of strong and capable technologies used to create scalable master online applications with front-end, back-end, and database elements. We cover the following in this MERN Stack tutorial:

- Overview of MERN Stack
- MERN Stack Architecture
- Primary Elements of MERN Stack
- Getting Started with MERN Stack
- Creating Components for MERN
- Connecting the front end to the back end
- Advantages of MERN Stack

Overview of MERN Stack

Generally, JavaScript creates full-stack web apps more quickly and easily. A user-friendly full-stack JavaScript framework for creating dynamic websites and applications is called MERN Stack. There are four primary technologies, or components, that make MERN Stack:



Featured Articles



Want to know more about becoming an expert in IT?

Click Here to Get Started

100% Placement Assurance

AUTHORITATIVE CERTIFICATION PART



Related Courses at SLA

- ➔ MERN Stack Training in OMR
- ➔ MERN Stack Training in Chennai
- ➔ MERN Stack Online Training

Related Posts



MERN Stack Tutorial for Web Development Aspirants

Published On: October 14, 2024

MERN Stack Tutorial for Web

- **M stands for MongoDB**, a NoSQL (non-structured query language) database system for creating document databases.
- **E, or Express**, is used for developing the Node.js web framework
- **R, or React**, is mostly used to create client-side JavaScript framework
- **N is an acronym for “node.js,”** to develop the top JavaScript web server.

These four technologies all contribute significantly to giving developers an end-to-end framework. In the process of creating web apps, even these four technologies are crucial. Hone your skills with our [Angular.JS training in Chennai](#).

MERN Stack Architecture

The three primary layers of MERN's three-tier architecture scheme are:

- Web as front-end tier
- Server as the middle tier
- Database as backend tier

Web Layer of Front-Tier

This tier, the top of the MERN stack, is mainly managed by React.js. It is one of the most popular front-end JavaScript libraries for online applications that can be downloaded for free.

- It is well known for producing client-side dynamic applications.
- React allows you to design complicated interfaces with a single component.
- It connects such complex interfaces to the data on the backend server.
- It allows users to create large-scale web apps with ease that refresh the page's content without having a page reload.

Server or Middle Tier

It is the logical step below the top layer and is mostly managed by Express.js and Node.js, two elements of the MERN stack.

- One of the most popular JavaScript frameworks for backend development is Express.js.
- It makes it considerably more straightforward for developers to put up reliable web servers and APIs.
- It enhances the useful features of Node.js HTTP (HyperText Transfer Protocol) objects.

Development Aspirants There is a growing need for competent MERN...



MERN Stack Developer Salary in Chennai

Published On: October 14, 2024

Introduction A MERN Stack Developer creates web applications using MongoDB, Express.js, React.js, and Node.js, managing...

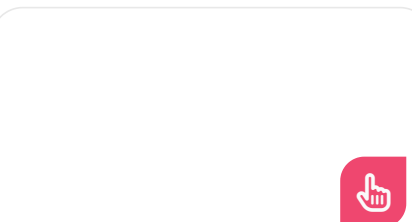
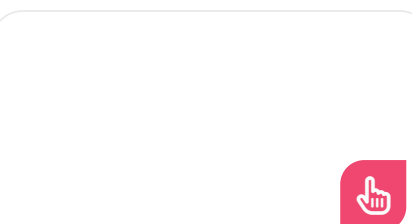


Tableau Developer Salary in Chennai

Published On: October 12, 2024

Introduction A Tableau Developer designs, develops, and maintains dashboards and visualizations using Tableau software. Key...



VMware Tutorial for Cloud Computing Aspirants

Published On: October 12, 2024

VMware Tutorial for Cloud Computing Aspirants VMware software allows you to run a virtual machine...

JavaScript is constantly used by Node.js. As a result, a computer user can swiftly construct any kind of online service or mobile application.

MERN Stack Interview Questions

Database Layer or Backend Tier

Mostly run by MongoDB, it is one of the most important tiers of the MERN Stack.

- A database's primary function is to store all of the data associated with your application, such as user profiles, comments, content, statistics, and other data.
- All of the data is mostly stored there for security reasons.
- It keeps an accurate record and typically gives the user access to the data when needed.
- The data is primarily stored in the database.
- To ensure that the system can always recover the precise data or information that the user originally requested, it creates two or more replica files of the data.
- It suggests that the table-like relational database structure is not the foundation of MongoDB.

It offers a completely different method for storing and retrieving data. An open-source document-oriented database, MongoDB is the most widely used NoSQL (also known as Non-Structured Query Language) database.

A non-relational database that does not need a fixed schema or appropriate relational tables to store the essential data in it is commonly referred to as "NoSQL."

Unlike relational databases, which store data in a structure consisting of rows and columns, MongoDB stores data in a distinct manner.

Enroll in our [**mobile app development course in Chennai**](#) for a promising career.

Primary Elements of MERN Stack

Let's explore the four technologies or elements that make up the MERN Stack here:

MongoDB

A non-relational database that does not need a fixed schema or appropriate relational tables to store the essential data in it is commonly referred to as "NoSQL."

Unlike relational databases, which store data in a structure consisting of rows and columns, MongoDB stores data in a distinct manner.

The data is saved in a binary format called BSON, or Binary JavaScript Object Notation; this format's binary structure encapsulates the length and type of the data, making it significantly faster to parse.

- It suggests that the table-like relational database structure is not the foundation of MongoDB.
- It offers a completely different method for storing and retrieving data.

Features of MongoDB

- **Schema-less Database:** Ability to store many document types in a single collection.
- **Indexing:** It makes it simple to retrieve the relevant data from the data pool.
- **Document-Oriented:** MongoDB stores all of its data in documents.
- **Faster:** MongoDB is significantly faster than other RDBMS.
- **Scalability:** Sharding helps make MongoDB more scalable.
- **High Performance:** Extremely high performance and data persistence when compared to other databases.
- **Replication and Highly Available:** By generating several copies of the data on many servers, MongoDB improves data availability.
- **Aggregation:** The GROUPBY clause processes the grouped data through many operations to obtain the unique or calculated
- **Simple Environment Setup:** Installing MongoDB on a system doesn't require much work.

Learn more through our [MongoDB training in Chennai](#).

Express.JS

Express is a server-side framework for JavaScript that is used with JS. It is among the top JavaScript frameworks for backend development. It gives the developer a framework for building and managing reliable servers. Explore more with our [full-stack developer course](#).

Express is used to rapidly and easily construct and design mobile and web applications. Express facilitates the organization of your application's functionality using middleware and routing for strong web servers.

Important Features of Express.JS

- Quicker and easier.
- The environment setup for Express is extremely straightforward.
- Express makes connecting to databases like MongoDB quite simple.
- Express gives you the ability to specify your application's routes.
- Routing using the HTTP method and URL patterns
- Error handling middleware makes error handling simple.
- REST API (Representational State Transfer Application Programming Interface) creation is simple.
- The two template engines that Express offers, EJS and Jade, make it simple to integrate data into a website's structure.
- enhanced speed and heightened security.
- It is scalable and incredibly effective.
- It is asynchronous and single-threaded.
- It also boasts the largest Node.js community.
- It supports code reuse with its integrated router.

React.JS

One of the most widely used open-source JavaScript front-end libraries for creating websites is called React. It is employed in the creation of user interfaces, particularly for single-page web applications. Learn more with our [React.JS course in Chennai](#).

Features of React.JS

- Easy to learn as it is simple.
- Data binding and native approach.
- High-performing and testable.

Node.JS

JavaScript code can be executed outside of a browser using the cross-platform JS server environment, which is available as an open-source download. It is frequently used to create and develop a wide range of backend services, including mobile and web applications.

Features of Node.JS

- Scalable and fast
- Easy to learn and debug
- Real-time web app development
- Data streaming and caching benefits
- Object-oriented and event-driven
- Huge community.

Explore our [Node.JS training in Chennai](#) to learn back-end development separately.

Getting Started with MERN Stack

You must complete the following tasks to get started:

- **Install Node.js:** Download and install the latest version of Node.js or the LTS version.
- **Installing a Code Editor:** We'll be utilizing Visual Studio Code for this tutorial.

Setting up the project

We can develop full-stack solutions with MERN. As a result, we will be developing a MERN stack project to fully utilize its capabilities.

We will develop a front end and a back end for this project.

- React will be used for the front end.
- Node.js, Express, and MongoDB will be used for the back end.

Both the front-end client and the back-end server will be called.

First, let's make a new directory called mern. Our client and server folders will be located in this folder.

```
mkdir mern && cd mern
```

Next, make a folder called server for the back end. Next, we'll initialize the package.json file through npm init.

```
mkdir server && cd server
```

```
npm init -y
```

We'll add a line to the package.json file so that we may use *ECMAScript Modules*, the officially accepted standard format for packaging JavaScript code for reuse.

```
"type": "module",
```

The dependencies will also be installed.

```
npm install mongodb express cors
```

The previous command installs three distinct packages:

- The MongoDB database driver enables data manipulation and database connections for your Node.js applications.
- Express is a Node.js web framework that will simplify our development.

- Cors is a Node.js module that facilitates resource sharing between origins.

The package.json file contains installed dependencies that we can view. The packages and their versions ought to be mentioned.

Following successful installation of dependencies, we create a file named server.js and add the following code to it:

```
import express from "express";

import cors from "cors";

import records from "../routes/record.js";

const PORT = process.env.PORT || 5050;

const app = express();

app.use(cors());

app.use(express.json());

app.use("/record", records);

app.listen(PORT, () => {

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

});
```

Express and Cors are being imported here for usage. const process.env = port. The port variable will be accessed by PORT from the config.env file that we'll make next.

If you are a serious job seeker, enroll in our [MERN Stack job seeker program](#).

[MERN Stack Course Syllabus](#)

Connecting to MongoDB Atlas

It's time to link the database and our server. The database that we will utilize is MongoDB Atlas.

A cloud-based database service that offers strong data security and dependability is called MongoDB Atlas.

You can use a portion of Atlas features and functionality with MongoDB Atlas's perpetually free tier cluster.

- You should create a config.env file in the server directory after obtaining the connection string.
- There, designate a new ATLAS_URI variable with the connection string.

When you're done, your file ought to resemble the one below.

- Put your database username, password, cluster name, and project ID in place of <username>, <password>, <clusterName>, and <projectId>.

```
ATLAS_URI=mongodb+srv://<username>:<password>@<cluster>.<projectId>.mongodb.net/employees?
retryWrites=true&w=majority
```

```
PORT=5050
```

Make a connection.js file and a folder called "db" beneath the server directory. To connect to our database, we can add the following code there:

```
import { MongoClient, ServerApiVersion } from "mongodb";
```

```
const uri = process.env.ATLAS_URI || "";
```

```
const client = new MongoClient(uri, {
```

```
  serverApi: {
```

```
    version: ServerApiVersion.v1,
```

```
    strict: true,
```

```
    deprecationErrors: true,
```

```
  },
```

```
});
```

```
try {
```

```
  // Connect the client to the server
```

```
  await client.connect();
```

```
  await client.db("admin").command({ ping: 1 });
```

```
  console.log(
```

```
    "Pinged your deployment. You successfully connected to MongoDB!"
```

```
  );
```



```
} catch(err) {  
  
  console.error(err);  
  
}  
  
let db = client.db("employees");  
  
export default db;
```

Join our [web design course](#) for your promising career in web development.

Server API endpoints

Let's create a routes folder and include record.js within it. Return to the "server" directory and make the new file and directory there:

```
cd ../server  
  
mkdir routes  
  
touch routes/record.js
```

The following lines of code will also be present in the routes/record.js file:

JavaScript

```
import express from "express";  
  
import db from "../db/connection.js";  
  
import { ObjectId } from "mongodb";  
  
const router = express.Router();  
  
router.get("/", async (req, res) => {  
  
  let collection = await db.collection("records");  
  
  let results = await collection.find({}).toArray();  
  
  res.send(results).status(200);  
  
});  
  
router.get("/:id", async (req, res) => {  
  
  let collection = await db.collection("records");  
  
  let query = { _id: new ObjectId(req.params.id) };
```

```
let result = await collection.findOne(query);

if (!result) res.send("Not found").status(404);

else res.send(result).status(200);

});

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

  try {

    let newDocument = {

      name: req.body.name,

      position: req.body.position,

      level: req.body.level,

    };

    let collection = await db.collection("records");

    let result = await collection.insertOne(newDocument);

    res.send(result).status(204);

  } catch (err) {

    console.error(err);

    res.status(500).send("Error adding record");

  }

});

router.patch("/:id", async (req, res) => {

  try {

    const query = { _id: new ObjectId(req.params.id) };

    const updates = {

      $set: {

        name: req.body.name,

        position: req.body.position,

        level: req.body.level,
```

```

    },
  };

  let collection = await db.collection("records");

  let result = await collection.updateOne(query, updates);

  res.send(result).status(200);

} catch (err) {

  console.error(err);

  res.status(500).send("Error updating record");

}

});

router.delete("/:id", async (req, res) => {

  try {

    const query = { _id: new ObjectId(req.params.id) };

    const collection = db.collection("records");

    let result = await collection.deleteOne(query);

    res.send(result).status(200);

  } catch (err) {

    console.error(err);

    res.status(500).send("Error deleting record");

  }

});

export default router;

```

The following message will appear in your terminal as the connection is established if you execute the application at this stage. Observe that we are utilizing the environment variable features that are pre-installed in the most recent Node.js versions.

```
> node -env-file=config.env server
```

Pinged your deployment. You successfully connected to MongoDB!

Server is running on port: 5050

That concludes the back end. We will now begin to work on the front end.

Configuring the React application

Let's configure this in a new terminal in the mern directory:

```
npm create vite@latest client --template react
```

```
cd client
```

```
npm install
```

We will be utilizing a few extra dependencies in our app.

```
npm install -D tailwindcss postcss autoprefixer
```

```
npx tailwindcss init -p
```

JavaScript

```
/** @type {import('tailwindcss').Config} */
```

```
export default {
```

```
  content: [
```

```
    "./index.html",
```

```
    "./src/**/*.{js,ts,jsx,tsx}",
```

```
  ],
```

```
  theme: {
```

```
    extend: {},
```

```
  },
```

```
  plugins: [],
```

```
}
```

The Tailwind directives must be added to the src/index.css file, and everything else must be removed.

```
@tailwind base;
```

```
@tailwind components;
```

```
@tailwind utilities;
```

Lastly, react-router-dom will be installed.

```
npm install -D react-router-dom
```

Tailwind is a utility-first CSS framework that uses preset class names to add CSS styles. Furthermore, React Router gives React client-side page routing! Gain expertise with our [web development course syllabus](#).

Configuring the React Router

```
import * as React from "react";

import * as ReactDOM from "react-dom/client";

import {
  createBrowserRouter,
  RouterProvider,
} from "react-router-dom";

import App from "./App";

import Record from "./components/Record";

import RecordList from "./components/RecordList";

import "./index.css";

const router = createBrowserRouter([

  {
    path: "/",
    element: <App />,
    children: [
      {
        path: "/",
        element: <RecordList />,
      },
    ],
  },
],

{
```

```

    path: "/edit/:id",
    element: <App />,
    children: [
      {
        path: "/edit/:id",
        element: <Record />,
      },
    ],
  },
  {
    path: "/create",
    element: <App />,
    children: [
      {
        path: "/create",
        element: <Record />,
      },
    ],
  },
]);

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);

```

Our UI is kept in sync with the URL by using RouterProvider, and we have configured our application page routes in the router variable.

- *RoutingProvider* facilitates smooth transitions between components.
- In essence, rather than refreshing or reloading the entire page, it will only reload or refresh the component that needs to be modified.
- React Router is not required, but it is essential if you want your application to be responsive.

Learn the fundamentals of web development with our [web development training in Chennai](#).

MERN Stack Developer Salary in Chennai

Creating Components

Let's now create the elements that we specified in our routes. Inside the src folder, make a components folder.

We'll add a new.js file to the components folder for every component we make.

In this instance, *Navbar.jsx*, *RecordList.jsx*, and *ModifyRecord.jsx* will be included.

```
mkdir src/components
```

```
cd src/components
```

```
touch Navbar.jsx RecordList.js ModifyRecord.jsx
```

Navbar.jsx: The following code will be used in the *navbar.js* component to create a navigation bar that will connect us to the necessary components.

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

```
export default function Navbar() {
```

```
  return (
```

```
    <div>
```

```
      <nav className="flex justify-between items-center mb-6">
```

```
        <NavLink to="/"> </img>
```

```
</NavLink>
```

```
<NavLink className="inline-flex items-center justify-center  
whitespace-nowrap text-md font-medium ring-offset-  
background transition-colors focus-visible:outline-none focus-  
visible:ring-2 focus-visible:ring-ring focus-visible:ring-offset-2  
disabled:pointer-events-none disabled:opacity-50 border  
border-input bg-background hover:bg-slate-100 h-9 rounded-  
md px-3" to="/create">
```

```
  Create Employee
```

```
</NavLink>
```

```
</nav>
```

```
</div>
```

```
);
```

```
}
```

RecordList.jsx: The viewing component for our records will be the code that follows. It will use the GET technique to retrieve every record in our database.

```
import { useEffect, useState } from "react";
```

```
import { Link } from "react-router-dom";
```

```
const Record = (props) => (<tr className="border-b transition-  
colors hover:bg-muted/50 data-[state=selected]:bg-muted">
```

```
  <td className="p-4 align-middle  
[&:has([role=checkbox]):pr-0">
```

```
    {props.record.name}
```

```
</td>
```

```
  <td className="p-4 align-middle  
[&:has([role=checkbox]):pr-0">
```

```
    {props.record.position}
```

```
</td>
```

```
  <td className="p-4 align-middle  
[&:has([role=checkbox]):pr-0">
```

```
    {props.record.level}
```

```
</td>
```



```
<td className="p-4 align-middle  
[&:has([role=checkbox]):pr-0">
```

```
<div className="flex gap-2">
```

```
<Link
```

```
  className="inline-flex items-center justify-center  
  whitespace-nowrap text-sm font-medium ring-offset-  
  background transition-colors focus-visible:outline-none focus-  
  visible:ring-2 focus-visible:ring-ring focus-visible:ring-offset-2  
  disabled:pointer-events-none disabled:opacity-50 border  
  border-input bg-background hover:bg-slate-100 h-9 rounded-  
  md px-3"
```

```
    to={`~/edit/${props.record._id}`}>
```

```
>
```

```
  Edit
```

```
</Link>
```

```
<button
```

```
  className="inline-flex items-center justify-center  
  whitespace-nowrap text-sm font-medium ring-offset-  
  background transition-colors focus-visible:outline-none focus-  
  visible:ring-2 focus-visible:ring-ring focus-visible:ring-offset-2  
  disabled:pointer-events-none disabled:opacity-50 border  
  border-input bg-background hover:bg-slate-100 hover:text-  
  accent-foreground h-9 rounded-md px-3"
```

```
  color="red"
```

```
  type="button"
```

```
  onClick={() => {
```

```
    props.deleteRecord(props.record._id);
```

```
  }}>
```

```
>
```

```
  Delete
```

```
</button>
```

```
</div>
```

```

    </td>
  </tr>
);

export default function RecordList() {

  const [records, setRecords] = useState([]);

  // This method fetches the records from the database.

  useEffect(() => {

    async function getRecords() {

      const response = await
fetch(`http://localhost:5050/record/`);

      if (!response.ok) {

        const message = `An error occurred:
${response.statusText}`;

        console.error(message);

        return;

      }

      const records = await response.json();

      setRecords(records);

    }

    getRecords();

    return;

  }, [records.length]);

  // This method will delete a record

  async function deleteRecord(id) {

    await fetch(`http://localhost:5050/record/${id}`, {

      method: "DELETE",

    });

    const newRecords = records.filter((el) => el._id !== id);

```

```

    setRecords(newRecords);
  }

  function recordList() {
    return records.map((record) => {
      return (
        <Record
          record={record}
          deleteRecord={() => deleteRecord(record._id)}
          key={record._id}
        />
      );
    });
  }

  return (
    <>
      <h3 className="text-lg font-semibold p-4">Employee
Records</h3>
      <div className="border rounded-lg overflow-hidden">
        <div className="relative w-full overflow-auto">
          <table className="w-full caption-bottom text-sm">
            <thead className=" [&_tr]:border-b">
              <tr className="border-b transition-colors hover:bg-
muted/50 data-[state=selected]:bg-muted">
                <th className="h-12 px-4 text-left align-middle font-
medium text-muted-foreground [&:has([role=checkbox])]:pr-
0">
                  Name
                </th>
                <th className="h-12 px-4 text-left align-middle font-
medium text-muted-foreground [&:has([role=checkbox])]:pr-

```

```
0">
```

```
    Position
```

```
  </th>
```

```
    <th className="h-12 px-4 text-left align-middle font-medium text-muted-foreground [&:has([role=checkbox])]:pr-0">
```

```
    Level
```

```
  </th>
```

```
    <th className="h-12 px-4 text-left align-middle font-medium text-muted-foreground [&:has([role=checkbox])]:pr-0">
```

```
    Action
```

```
  </th>
```

```
</tr>
```

```
</thead>
```

```
<tbody className="[&_tr.last-child]:border-0">
```

```
  {recordList()}
```

```
</tbody>
```

```
</table>
```

```
</div>
```

```
</div>
```

```
</>
```

```
);
```

```
}
```

Record.jsx: The code that follows will function as a form element for adding or editing records. This part will send a command to our server to either create or update.

```
import { useState, useEffect } from "react";
```

```
import { useParams, useNavigate } from "react-router-dom";
```

```
export default function Record() {
```

```

const [form, setForm] = useState({
  name: "",
  position: "",
  level: "",
});

const [isNew, setIsNew] = useState(true);

const params = useParams();

const navigate = useNavigate();

useEffect(() => {
  async function fetchData() {
    const id = params.id?.toString() || undefined;
    if(!id) return;
    setIsNew(false);

    const response = await fetch(
      `http://localhost:5050/record/${params.id.toString()}`
    );

    if (!response.ok) {
      const message = `An error has occurred:
      ${response.statusText}`;
      console.error(message);
      return;
    }
  }

  const record = await response.json();

  if (!record) {
    console.warn(`Record with id ${id} not found`);
    navigate("/");
    return;
  }
}

```

```

    setForm(record);
  }

  fetchData();

  return;
}, [params.id, navigate]);

function updateForm(value) {
  return setForm((prev) => {
    return { ...prev, ...value };
  });
}

async function onSubmit(e) {
  e.preventDefault();

  const person = { ...form };

  try {
    let response;

    if (isNew) {
      response = await fetch("http://localhost:5050/record", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(person),
      });
    } else {
      response = await
fetch(`http://localhost:5050/record/${params.id}`); {
        method: "PATCH",

```

```

headers: {
  "Content-Type": "application/json",
},
body: JSON.stringify(person),
});
}
if (!response.ok) {
  throw new Error(`HTTP error! status: ${response.status}`);
}
} catch (error) {
  console.error("A problem occurred with your fetch operation:
; error);
} finally {
  setForm({ name: "", position: "", level: "" });
  navigate("/");
}
}
return (
  <>
  <h3 className="text-lg font-semibold p-4">Create/Update
Employee Record</h3>
  <form
  onSubmit={onSubmit}
  className="border rounded-lg overflow-hidden p-4"
  >
  <div className="grid grid-cols-1 gap-x-8 gap-y-10
border-b border-slate-900/10 pb-12 md:grid-cols-2">
  <div>
  <h2 className="text-base font-semibold leading-7 text-

```

```
slate-900">
```

```
    Employee Info
```

```
</h2>
```

```
<p className="mt-1 text-sm leading-6 text-slate-600">
```

```
    This information will be displayed publicly so be careful  
what you
```

```
    share.
```

```
</p>
```

```
</div>
```

```
<div className="grid max-w-2xl grid-cols-1 gap-x-6  
gap-y-8 ">
```

```
<div className="sm:col-span-4">
```

```
<label
```

```
    htmlFor="name"
```

```
    className="block text-sm font-medium leading-6  
text-slate-900"
```

```
>
```

```
    Name
```

```
</label>
```

```
<div className="mt-2">
```

```
<div className="flex rounded-md shadow-sm ring-1  
ring-inset ring-slate-300 focus-within:ring-2 focus-within:ring-  
inset focus-within:ring-indigo-600 sm:max-w-md">
```

```
<input
```

```
    type="text"
```

```
    name="name"
```

```
    id="name"
```

```
    className="block flex-1 border-0 bg-transparent  
py-1.5 pl-1 text-slate-900 placeholder:text-slate-400 focus:ring-0  
sm:text-sm sm:leading-6"
```



```

        placeholder="First Last"

        value={form.name}

        onChange={(e) => updateForm({ name:
e.target.value })}

    />

</div>

</div>

</div>

<div className="sm:col-span-4">

    <label

        htmlFor="position"

        className="block text-sm font-medium leading-6
text-slate-900"

    >

        Position

    </label>

    <div className="mt-2">

        <div className="flex rounded-md shadow-sm ring-1
ring-inset ring-slate-300 focus-within:ring-2 focus-within:ring-
inset focus-within:ring-indigo-600 sm:max-w-md">

            <input

                type="text"

                name="position"

                id="position"

                className="block flex-1 border-0 bg-transparent
py-1.5 pl-1 text-slate-900 placeholder:text-slate-400 focus:ring-0
sm:text-sm sm:leading-6"

                placeholder="Developer Advocate"

                value={form.position}

                onChange={(e) => updateForm({ position:

```

```

e.target.value }}}

    />

  </div>

</div>

</div>

<div>

  <fieldset className="mt-4">

    <legend className="sr-only">Position
Options</legend>

    <div className="space-y-4 sm:flex sm:items-center
sm:space-x-10 sm:space-y-0">

      <div className="flex items-center">

        <input

          id="positionIntern"

          name="positionOptions"

          type="radio"

          value="Intern"

          className="h-4 w-4 border-slate-300 text-slate-
600 focus:ring-slate-600 cursor-pointer"

          checked={form.level === "Intern"}

          onChange={(e) => updateForm({ level:
e.target.value })}}

        />

        <label

          htmlFor="positionIntern"

          className="ml-3 block text-sm font-medium
leading-6 text-slate-900 mr-4"

          >

            Intern

          </label>

```

```
<input
  id="positionJunior"
  name="positionOptions"
  type="radio"
  value="Junior"
  className="h-4 w-4 border-slate-300 text-slate-
600 focus:ring-slate-600 cursor-pointer"
  checked={form.level === "Junior"}
  onChange={(e) => updateForm({ level:
e.target.value })}
/>

<label
  htmlFor="positionJunior"
  className="ml-3 block text-sm font-medium
leading-6 text-slate-900 mr-4"
>
  Junior
</label>

<input
  id="positionSenior"
  name="positionOptions"
  type="radio"
  value="Senior"
  className="h-4 w-4 border-slate-300 text-slate-
600 focus:ring-slate-600 cursor-pointer"
  checked={form.level === "Senior"}
  onChange={(e) => updateForm({ level:
e.target.value })}
/>
```

```

    <label
      htmlFor="positionSenior"
      className="ml-3 block text-sm font-medium
leading-6 text-slate-900 mr-4"
    >
      Senior
    </label>
  </div>
</div>
</fieldset>
</div>
</div>
</div>
<input
  type="submit"
  value="Save Employee Record"
  className="inline-flex items-center justify-center
whitespace-nowrap text-md font-medium ring-offset-
background transition-colors focus-visible:outline-none focus-
visible:ring-2 focus-visible:ring-ring focus-visible:ring-offset-2
disabled:pointer-events-none disabled:opacity-50 border
border-input bg-background hover:bg-slate-100 hover:text-
accent-foreground h-9 rounded-md px-3 cursor-pointer mt-4"
  />
</form>
</>
);
}

```

The following is now added to the src/App.jsx file.

```
import { Outlet } from "react-router-dom";
```

```

import Navbar from "../components/Navbar";

const App = () => {

  return (

    <div className="w-full p-6">

      <Navbar />

      <Outlet />

    </div>

  );

};

export default App

```

This is the primary element of our layout. The Outlet will accept the children components we set in our routes in the main.jsx file before, and our Navbar will always be at the top of every page.

Explore our [MERN Stack course syllabus](#) to get started on your learning journey.

MERN Stack Online Training

Connecting the front end to the back end

The creation of the components is now complete. We also used fetch to link our React app to the back end, which offers simpler and more streamlined methods for handling http requests.

We added the following code to the onSubmit(e) block of Record.jsx.

Fetch will either add a new record to the database or update an existing record when a POST or PATCH request is made to the URL.

```

async function onSubmit(e) {

  e.preventDefault();

  const person = { ...form };

  try {

    const response = await
    fetch(`http://localhost:5050/record${params.id ? "/" + params.id :`

```

```

    ""}, {
      method: `${params.id ? "PATCH" : "POST"}`,
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(person),
    });
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
  } catch (error) {
    console.error('A problem occurred with your fetch operation: ',
error);
  } finally {
    setForm({ name: "", position: "", level: "" });
    navigate("/");
  }
}
}

```

To load an existing record, if it exists, we also added the following block of code to Record.jsx underneath the constructor block.

```

useEffect(() => {
  async function fetchData() {
    const id = params.id?.toString() || undefined;
    if(!id) return;
    const response = await fetch(
      `http://localhost:5050/record/${params.id.toString()}`
    );
    if (!response.ok) {

```

```

    const message = `An error has occurred:
    ${response.statusText}`;

    console.error(message);

    return;
  }

  const record = await response.json();

  if (!record) {

    console.warn(`Record with id ${id} not found`);

    navigate("/");

    return;

  }

  setForm(record);

}

fetchData();

return;

}, [params.id, navigate]);

```

The last file is RecordList.jsx. We will use fetch's GET method to retrieve records from the database because RecordList.jsx fetches the records from the database. The following lines of code were added to RecordList.jsx above the RecordList() function to accomplish this.

```

useEffect(() => {

  async function getRecords() {

    const response = await fetch(`http://localhost:5050/record/`);

    if (!response.ok) {

      const message = `An error occurred: ${response.statusText}`;

      console.error(message);

      return;

    }

    const records = await response.json();

```

```
    setRecords(records);  
  }  
  
  getRecords();  
  
  return;  
}, [records.length]);
```

Once everything has been shut off, take the following actions to launch the app:

Ensure that the server application is still active. If not, use the following command in the server directory to start it:

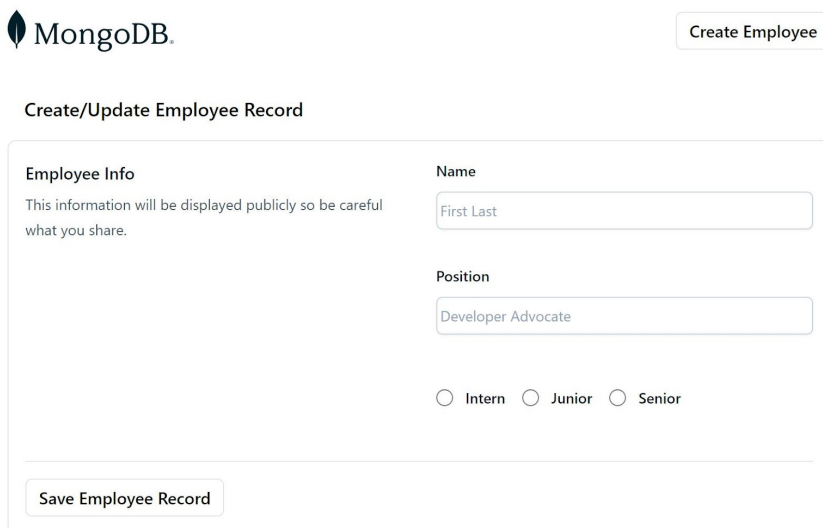
```
node --env-file=config.env server
```

Navigate to the client directory in a new terminal and type the following command:

```
npm run dev
```

Now it is ready to add the records.

Output



The screenshot shows a web application interface for MongoDB. At the top left is the MongoDB logo. To its right is a button labeled "Create Employee". Below this is a heading "Create/Update Employee Record". The main content area is a form with two columns. The left column is titled "Employee Info" and contains the text "This information will be displayed publicly so be careful what you share." The right column is titled "Name" and has a text input field with "First Last" inside. Below the "Name" section is a "Position" section with a text input field containing "Developer Advocate". Underneath the position field are three radio buttons labeled "Intern", "Junior", and "Senior". At the bottom left of the form is a button labeled "Save Employee Record".

Learn MongoDB separately in our [MongoDB training in Chennai](#).

Advantages of MERN Stack

There are numerous benefits to the MERN stack, such as:

- **Performance:** The MERN stack is optimized for Node.js to provide good performance.
- **Scalability:** The MERN stack makes it simple to scale existing functionalities or add new ones.

- **Open source:** There is no proprietary licensing associated with the full MERN stack.
- **Community support:** There is a vibrant community for the MERN stack that can assist with prompt problem-solving.
- **Flexibility:** The MERN stack can write code for servers and browsers alike.
- **Cost-effectiveness:** Compared to alternative frameworks, the MERN stack uses fewer resources.
- **Architecture:** The architecture of the MERN stack is simple to maintain.
- **MVC architecture:** This design keeps business logic and display details apart.
- **Full stack development:** Frontend and backend app development are supported by the MERN stack.

Conclusion

We hope this MERN Stack tutorial will be helpful for beginners who started learning web development. Hone your skills with our

[MERN Stack training in Chennai.](#)

Share on your Social Media



Softlogic Academy

Softlogic Systems

KK Nagar [Corporate Office]

No.10, PT Rajan Salai, K.K. Nagar, Chennai – 600 078.

Landmark: Karnataka Bank Building

Phone: [+91 86818 84318](tel:+918681884318)

Email: enquiry@softlogicsys.in

Map: [Google Maps Link](#)

OMR

No. E1-A10, RTS Food Street
92, Rajiv Gandhi Salai (OMR),
Navalur, Chennai – 600 130.

Navigation

[About Us](#)

[Blog Posts](#)

[Careers](#)

[Contact](#)

[Placement Training](#)

[Corporate Training](#)

[Hire With Us](#)

[Job Seekers](#)

[SLA's Recently Placed Students](#)

[Reviews](#)

[Sitemap](#)

Important Links

[Disclaimer](#)

[Privacy Policy](#)

[Terms and Conditions](#)

Landmark: Adj. to AGS Cinemas

Phone: [+91 89256 88858](tel:+918925688858)

Email: info@softlogicsys.in

Map: [Google Maps Link](#)

Courses

Python

Software Testing

Full Stack Developer

Java

Power BI

Clinical SAS

Data Science

Embedded

Cloud Computing

Hardware and Networking

VBA Macros

Mobile App Development

DevOps

Social Media Links



Review Sources

Google

Trustpilot

Glassdoor

Mouthshut

Sulekha

Justdial

Ambitionbox

Indeed

Software Suggest

Sitejabber