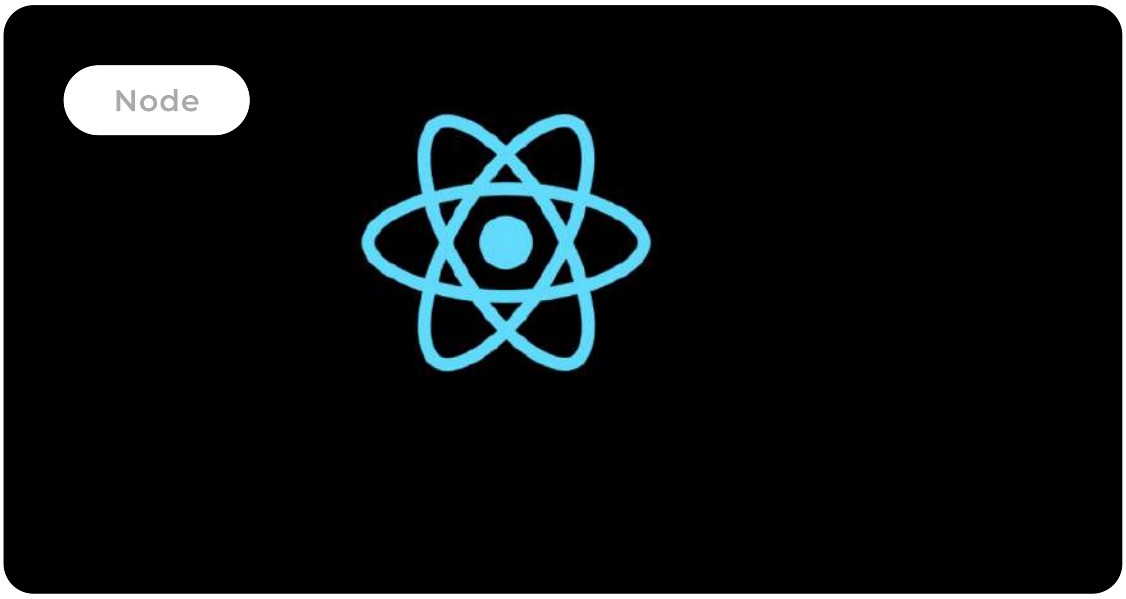


Md Niaz Rahman Khan Last Updated On: February 8, 2023 0

How to build a CRUD application using MERN stack

CRUD refers to the four functions that are Create, Read, Update and Delete and it is the most basic operation of a web application. Here, the create function allows the client to create a new record into the database and the read function allows the client to get the particular data from the database. The other function named update is used to update the data and finally, the delete function is used to delete the data from the database.

This operation is widely used in web applications. If you notice, any application around you then you will be able to see that the base of these applications is operating the CRUD operation. Let's say the most popular application is Facebook, here, you can create a post, read it, update it and also delete it from your profile. That means you are operating a CRUD operation. This CRUD operation can be performed based on some technologies. These technologies as a whole are called stacks.

Some popular stacks are MERN, LAMP, MEAN, etc. Here, MERN refers to MongoDB, ExpressJS, ReactJS, and NodeJS. On the other hand, LAMP stands for Linux, Apache, MySQL, and PHP, and MEAN refers to MongoDB, ExpressJS, AngularJS, and NodeJS.

In this tutorial, we will use the MERN stack to create a CRUD application. We are going to create a simple PhoneBook app where a user can create a phone number along with the name, update the phone number, see the phone number list and finally, delete it.

In our stack, we will use MongoDB to store our data, expressJS to handle the backend, and ReactJS to handle the frontend. After completing this tutorial, you can create your own CRUD application by using the MERN stack and you will also get a complete idea of the full-stack project.

Prerequisite:

To complete this tutorial in the most effective way you need to keep some things in your mind.

- A configured computer with an internet connection.
- Basic knowledge of JavaScript, HTML, CSS.
- A text editor – You can choose any text editor on which you may write JavaScript code such as sublime text, nodepad++, etc. In this tutorial, we will use visual studio code.
- NodeJS installed on your computer. We are using Node version **v14.17.4.** You may use this version or higher. But it is recommended not to use the node version under 10.0.0 while completing this tutorial.
- Postman and MongoDB compass installed on your computer. We will test our backend endpoints with the postman and monitor our database through MongoDB compass.

- Nice to have little knowledge about NodeJS, ReactJS, and Mongoose if you don't, not need to worry we will cover these later in this tutorial.

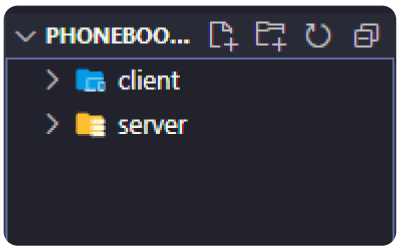
Topics To Be Covered In This Tutorial :

In this tutorial, we will divide our topics into two parts. First, we will implement our backend portion with expressJS, and Mongoose, and then we will implement our frontend portion with react.

- Environment Setup for Backend
- Create Node Server
- Connect MongoDB Database
- Create PhoneBook model
- Create Post Route
- Create Get Route
- Create Update Route
- Create Delete Route
- Environment setup for Frontend
- Add data with Axios.post
- Get data with Axios.get
- Update data with Axios.put
- Delete data with Axios.delete
- Conclusion and Next step

Environment Setup for Backend

Setting up the environment is the most important part of any application development because without a proper environment you can not implement any functionality. For the backend, we need to install some important packages. But at first, we will create a folder named `phonebook-app` and inside this folder, we will create two folders as `client` and `server`. The client folder will be used for implementing the frontend part and the server folder will be used for implementing the backend part. Let's see the below image of our project folders:



This is what our project folders will look like. Now let's implement the backend functionality inside the server folder and to do so, at first we need to change our directory to the server folder. For the first time, when you open the phonebook folder in your Vscode the directory will be set as phonebook, and to change it you need to run a command in your terminal as `cd server`

Step 1:

Now, we are ready to implement backend functionality in our server folder. At first, we have to give a command in our terminal and the command is

```
npm init
or
npm init -y
```

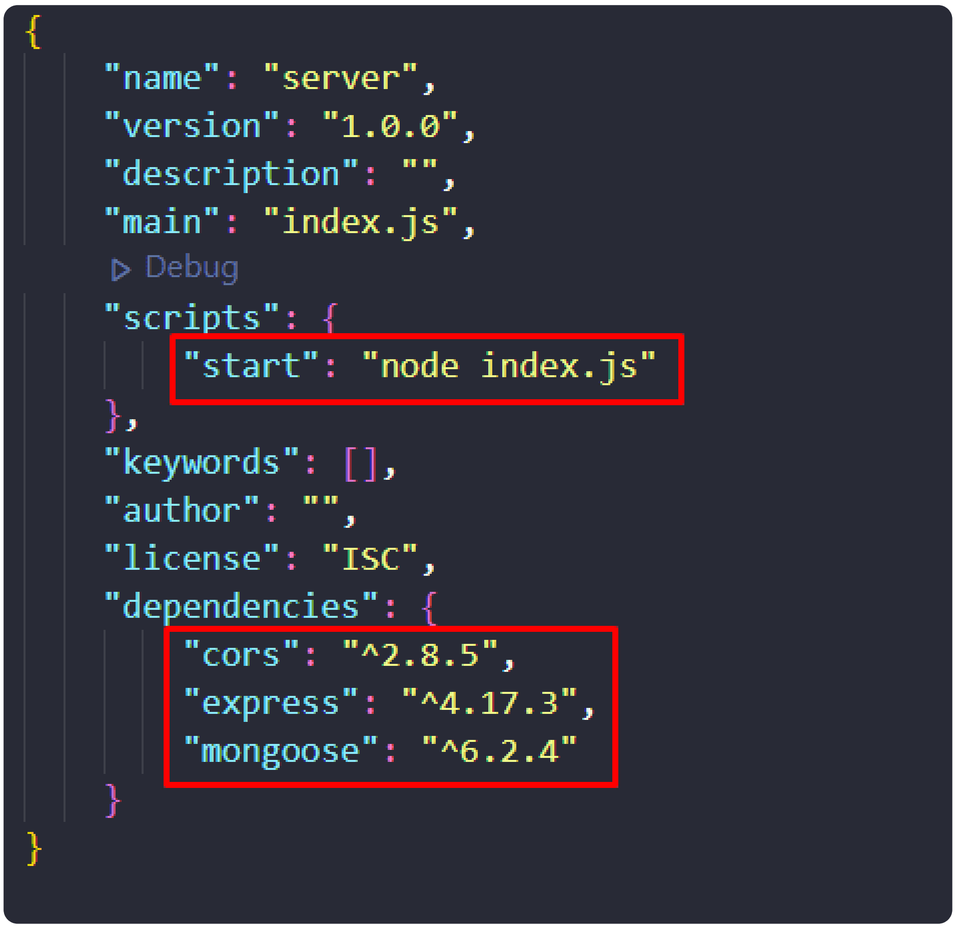
This command will create a `package.json` file for us, from where we will be able to manage our installed packages and also control the version of our application. If you want to create the package manually then you need to give the command `npm init` and if you want to create the file as a whole then you need to type `npm init -y`

Step 2:

In this step, we will install our necessary packages like express, mongoose. Express is a popular NodeJS framework and the mongoose is the Object Data Modeling library for MongoDB. We will run the `NPM` command to install these. See the below example :

```
npm install express mongoose cors
or
npm i express mongoose cors
```

Here, with the help of this command, we will be able to install both at a single time. The express will use to write NodeJS code and set up endpoints and Mongoose will be used to create a database model and save data. The CORS is referred to as Cross-Origin-Resource-Sharing which will help us to build a connection with the frontend. After installing these, our package.json file will look like this:



```json
{
    "name": "server",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    ▷ Debug
    "scripts": {
        "start": "node index.js"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "cors": "^2.8.5",
        "express": "^4.17.3",
        "mongoose": "^6.2.4"
    }
}
```

Here, we have set the script as `“start”: “node index.js”` and you can see those packages with the version. Our environment has been set up successfully and now we are ready to go to the next part.

Create Node Server

To create a Node server with express we need to require the express library first and then we need to define a `Port` number and then create a server with the help of `app.listen()` We will write this code in the `index.js` which is our app's entry point. file See the below code example:

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

app.use(express.json())
app.use(cors())
const PORT = 8080
app.listen(PORT, () => {
    console.log(`Server is running on PORT ${PORT}...`)
})
```

Here, you can see that we have set the port as `8080` now if we run the command as `npm start` we will be able to see the output as `Server is running on PORT 8080...` in the console.

Connect MongoDB Database

Like, express we need to require mongoose for counting the database and we also need the connection credentials from the Atlas MongoDB. Let me assume that, you have already know about these. See the below code example of the connecting database:

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

const DB = 'mongodb+srv://<YOUR USERNAME>:<YOUR PASSWORD>@cluster0.zozv5.mongodb.net/myFirstDatabase?retryWrites=true&w=majority'
mongoose.connect(DB, {
    useNewUrlParser: true,
     useUnifiedTopology: true,
}).then(() =>{
    console.log('Database connected..')
})
```

Here, we have saved the credentials in a variable named `DB` and we are hiding the credentials of `username` and `password` You may set your own credentials on that variable. Finally, we have connected the database with the help of mongoose by using `mongoose.connect()` We pass the variable there and write some code for avoiding the warnings.

Create PhoneBook model

After connecting the MongoDB database, it is time to create our database schema model. That will give the shape of the data and define how the data will be stored in the database. We will create it into a separate folder and will name it `Model` and inside the model folder, we will create a folder named `PhoneBook.js` Inside this folder, we will create a simple database schema model. See the below code example:

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

const PhoneBookSchema = new mongoose.Schema({
    name : {
        type : String,
        required : true
    },
    phone : {
        type : Number,
        required : true
    }
})

const PhoneBook = mongoose.model('PhoneBook',PhoneBookSchem
a)

module.exports = PhoneBook
```

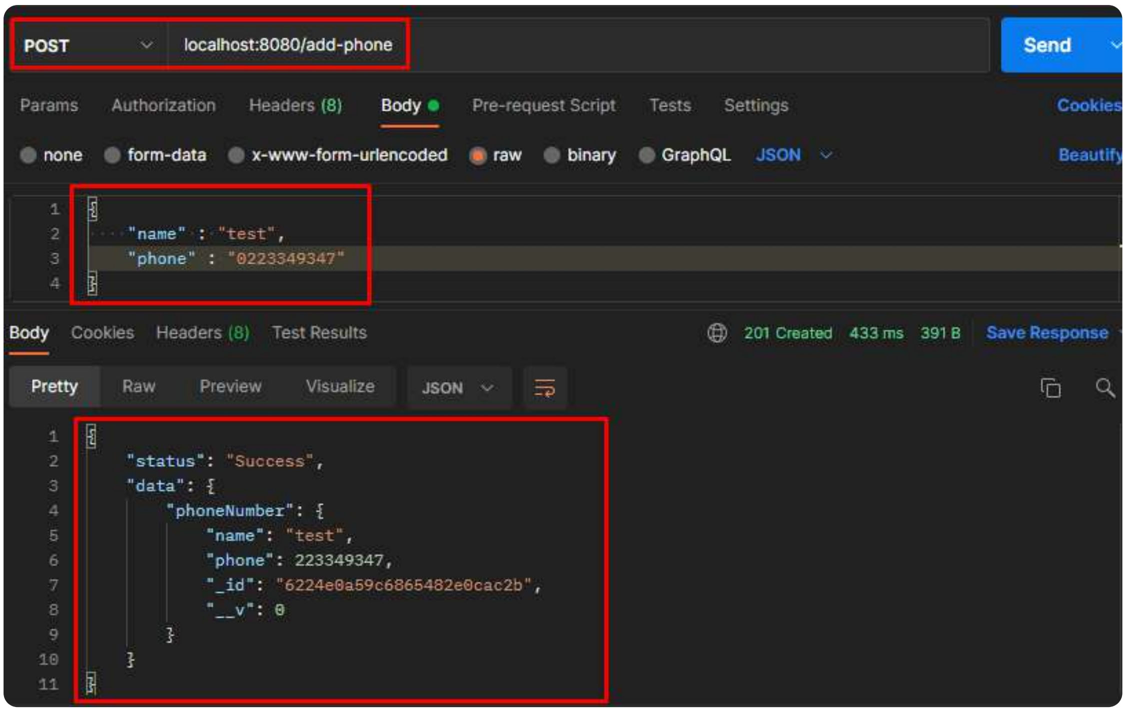
Here, you can see that we have created a schema model where we will store two types of data. One is the `name` which type is `String` and another one is the `phone` which type is `Number` We also set them as `required: true` as a result, it will throw a mongo error if any fields remain empty. Finally, we have exported this schema model so that we can use it by importing it into another file.

Create Post Route

Now, we are ready to create a post route where users will post particular data by hitting on a specific route and the data will be stored in the database. See the below code example where we have implemented this functionality:

```
const PhoneBook = require('./model/PhoneBook')

app.post('/add-phone', async(req,res) => {
    const phoneNumber = new PhoneBook(req.body)
    try{
        await phoneNumber.save()
        res.status(201).json({
            status: 'Success',
            data : {
                phoneNumber
            }
        })
    }catch(err){
        res.status(500).json({
            status: 'Failed',
            message : err
        })
    }
})
```

At first, we have imported the PhoneBook model and then create the post route with the help of the app.post() inside this, we have set the '/add-phone' that means if anyone types http://localhost:8080/add-phone then he will be able to add data and the data will be saved in the database. As we haven't implemented the frontend functionality we will check this with the help of the postman. Let's check the output:



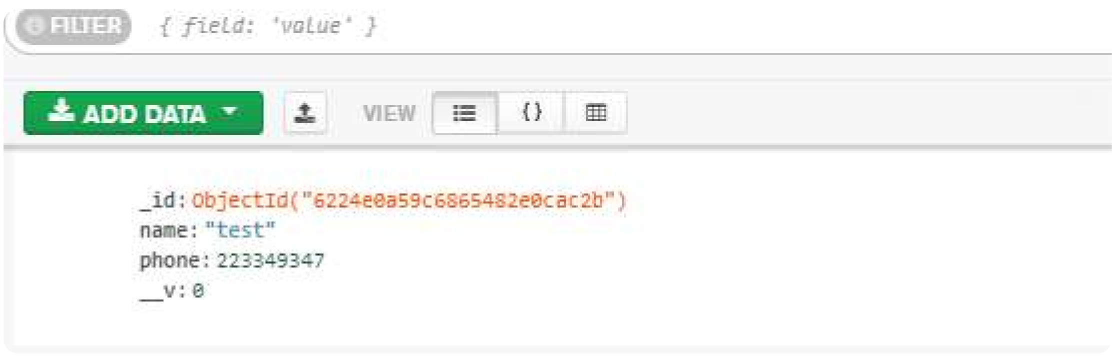
localhost:8080/add-phone

```
{
    "name" : "test",
    "phone" : "0223349347"
}
```

201 Created 433 ms 391 B

```
{
    "status": "Success",
    "data": {
        "phoneNumber": {
            "name": "test",
            "phone": 223349347,
            "_id": "6224e0a59c6865482e0cac2b",
            "__v": 0
        }
    }
}
```

Here, in the postman, you can see that we have typed the particular route and also typed a dummy name and phone number and clicked the send button. As a result, it shows us a success message with a unique id and refers that our data has been saved in the database. Now to Cross check, we will open our MongoDB compass and try to see if there's any data with the name of "test" or not.



```
_id: ObjectId("6224e0a59c6865482e0cac2b")
name: "test"
phone: 223349347
__v: 0
```

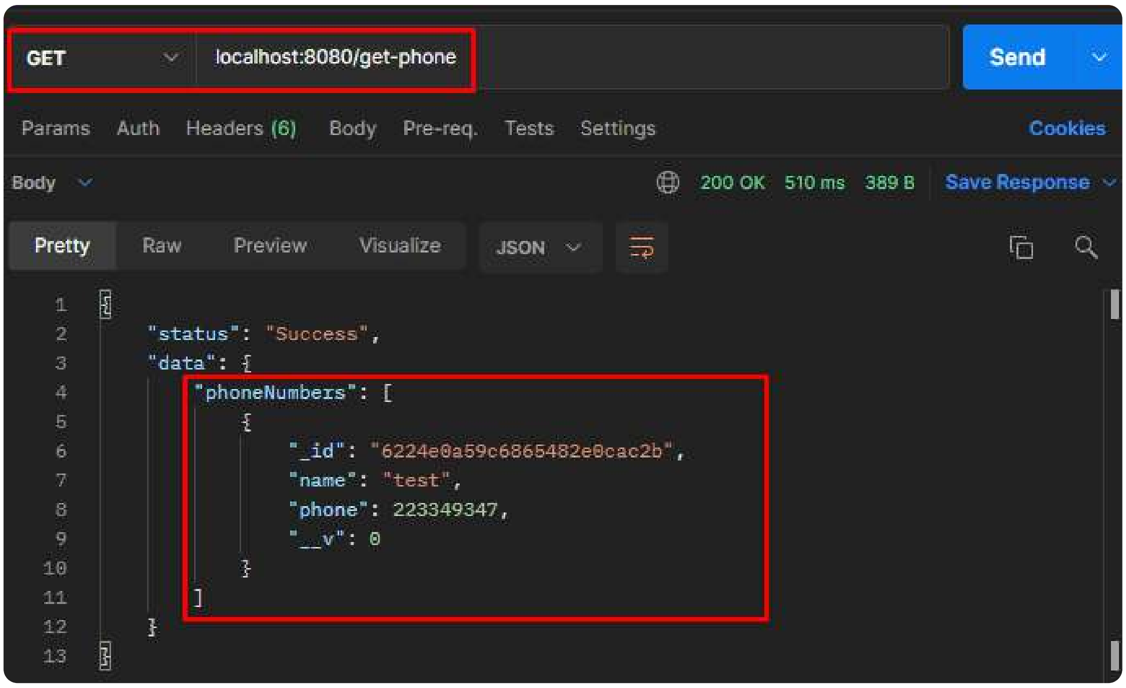
Here, in the MongoDB compass, you can see that the exact data has been saved. That means we have successfully been able to implement our `POST` functionality.

Create Get Route

We have already created the post route. That means users can save the data but how can we see those data. To see those data we need to implement the get routes. Here, we can create a get route with the help of the `app.get()` , and to query data, we will use `Model.find()` method. See the below code example of this implementation:

```
app.get('/get-phone', async (req,res) => {
    const phoneNumbers = await PhoneBook.find({})
    try{
        res.status(200).json({
            status : 'Success',
            data : {
                phoneNumbers
            }
        })
    }catch(err){
        res.status(500).json({
            status: 'Failed',
            message : err
        })
    }
})
```

Here, you can see that we have implemented the get route and we are using async-await Let’s check the functionality again with the help of the Postman below:


GET
localhost:8080/get-phone
Send
Params Auth Headers (6) Body Pre-req. Tests Settings Cookies
Body
200 OK 510 ms 389 B Save Response
Pretty Raw Preview Visualize JSON

```
{
    "status": "Success",
    "data": {
        "phoneNumbers": [
            {
                "_id": "6224e0a59c6865482e0cac2b",
                "name": "test",
                "phone": 223349347,
                "__v": 0
            }
        ]
    }
}
```

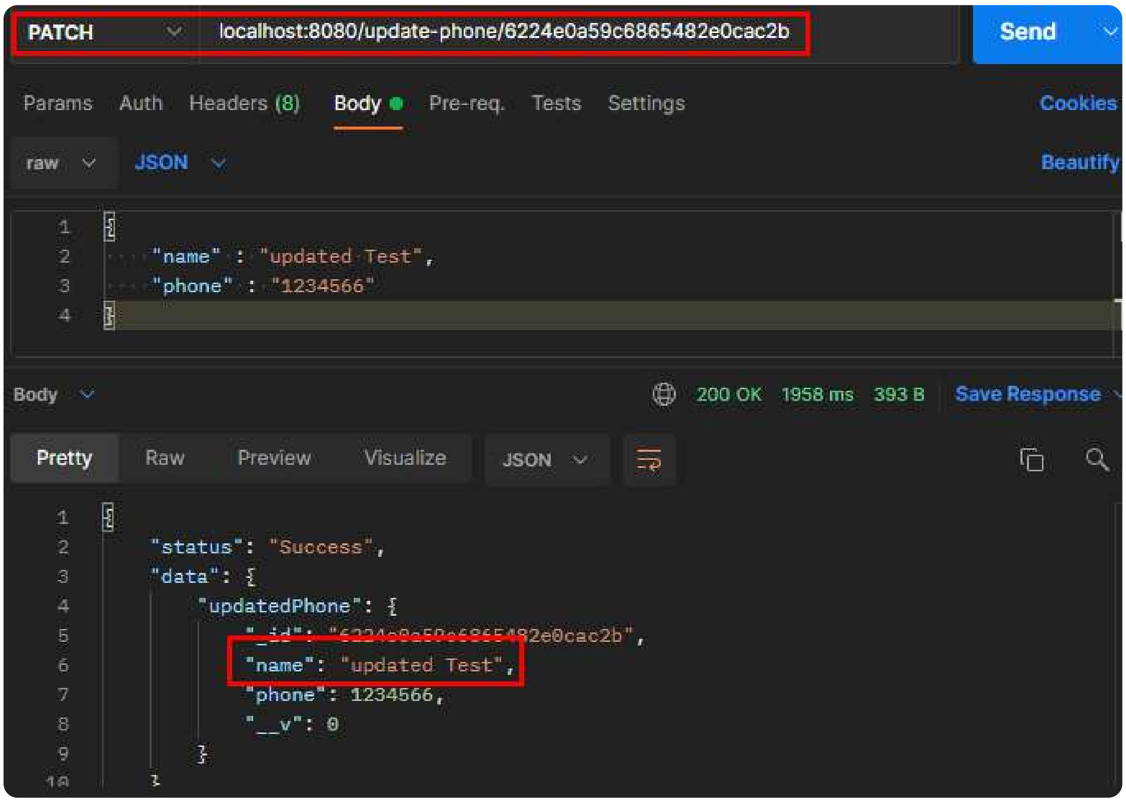
Here, you can see that we are getting the data successfully from the database with the help of hitting the specific route.

Create Update Route

Sometimes, we need to update the data and to perform this action we will implement the update route where we will be able to update the data by hitting the specific route. We will use the `findByIdAndUpdate()` method where we will be able to find a specific id and then change the value of that id's data. See the below code example :

```
app.patch('/update-phone/:id', async (req,res) => {
    const updatedPhone = await PhoneBook.findByIdAndUpdate(req.params.id,req.body,{
        new : true,
        runValidators : true
      })
    try{
        res.status(200).json({
            status : 'Success',
            data : {
              updatedPhone
            }
          })
    }catch(err){
        console.log(err)
    }
})
```

Here, we have to select the id first and then update the value of this. Let's check the postman in the below:

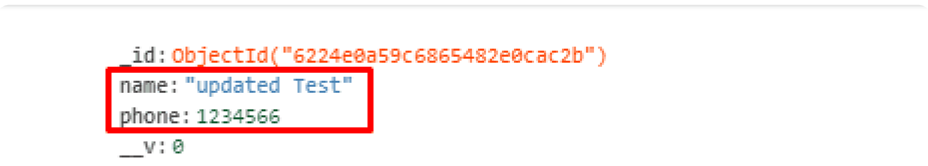


```
localhost:8080/update-phone/6224e0a59c6865482e0cac2b
```

```
{
    "name" : "updated Test",
    "phone" : "1234566"
}
```

```
{
    "status": "Success",
    "data": {
        "updatedPhone": {
            "_id": "6224e0a59c6865482e0cac2b",
            "name": "updated Test",
            "phone": 1234566,
            "__v": 0
        }
    }
```

Here, you can see that we have updated the previous data with the name updated Test, and it shows a success message in the Postman. Let's check the database also if the data is actually updated or not.



```
_id: ObjectId("6224e0a59c6865482e0cac2b")
name: "updated Test"
phone: 1234566
__v: 0
```

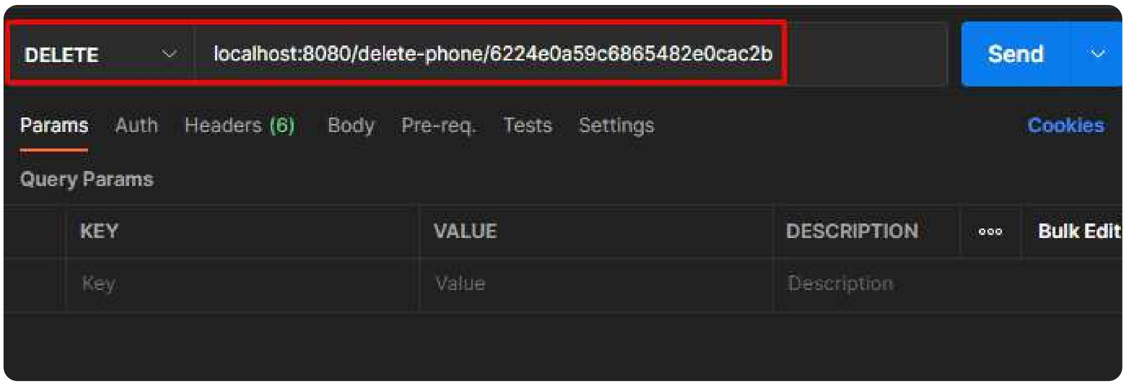
You can see that the data has been successfully updated in the database also. In the next step, we will implement the delete route.

Create Delete Route

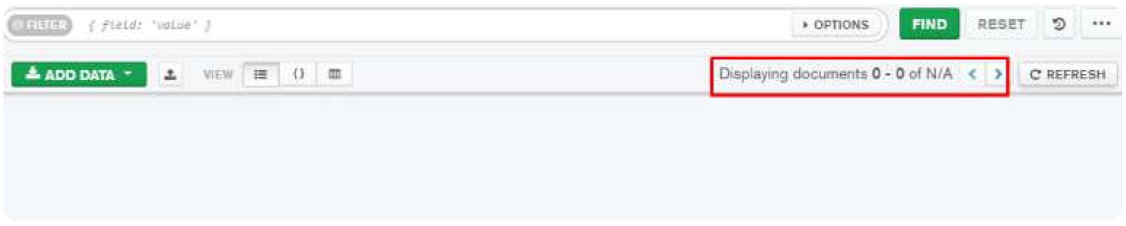
In our application, sometimes we need to delete a particular data from the database and we can do that by creating a delete route. We will create the functionality of deleting a single item from the database with the help of an id. Because we do not want to delete the whole data from the database. Let's see the below code of this:

```
  app.delete('/delete-phone/:id', async(req,res) => {
     await PhoneBook.findByIdAndDelete(req.params.id)

     try{
       res.status(204).json({
           status : 'Success',
           data : {}
       })
     }catch(err){
         res.status(500).json({
             status: 'Failed',
             message : err
         })
     }
 })
```

Here, at first, we search for specific data with the help of the findByIdAndDelete() method and simply set the value of that data empty. Let's check it in the below:


DELETE localhost:8080/delete-phone/6224e0a59c6865482e0cac2b

Here, you can see that we simply hit the delete route and send a delete request and it should delete the item from the database. Let's check our database and see if the data has been deleted or not.


Displaying documents 0 - 0 of N/A

You can see that, there are no elements that exist in our database. That means our program is working successfully.

We have implemented the CRUD operation in the backend because now we can create data, read data, update data, and delete data. But, we are half done with this tutorial and to perform this action we need to use Postman. This is not our goal. We want to visible our data and perform these actions from the frontend. Our API is ready and now we are ready to implement our next part of this tutorial. we will use React as frontend technology and try to build a complete CRUD application that is based on the MERN stack.

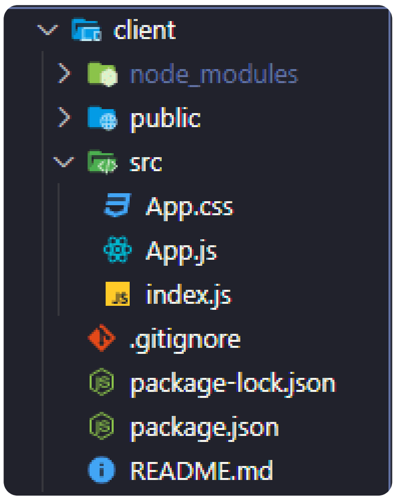
Environment setup for Frontend

To work with the frontend, at first we need to set up the environment for it. Till now, we were in the server directory and wrote our backend code. Now, we have to shift our directory first and change the directory, as before, we need to open our phonebook app and then run the command `cd client` into the terminal. It will take us to the client folder. After that, we need to write another command that will create a demo react app for us. See the below command:

```
npx create-react-app .
```

With the help of this command, we will be able to install react and its necessary packages in our client folder. Here, `npx create-react-app` is the main command, and the `“dot(.)”` represents that the app will be created in the present folder that is the `client`. It will take some time to install all the packages for you and it depends on the internet connection. Make sure that you have connected your computer to the internet.

After the installation process has been finished, it will create a few files for you but all those files will not be necessary to complete the tutorial. So, you may simply remove them. We have already done that and the file structure will look something like the below image:



```
client
  node_modules
  public
  src
    App.css
    App.js
    index.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

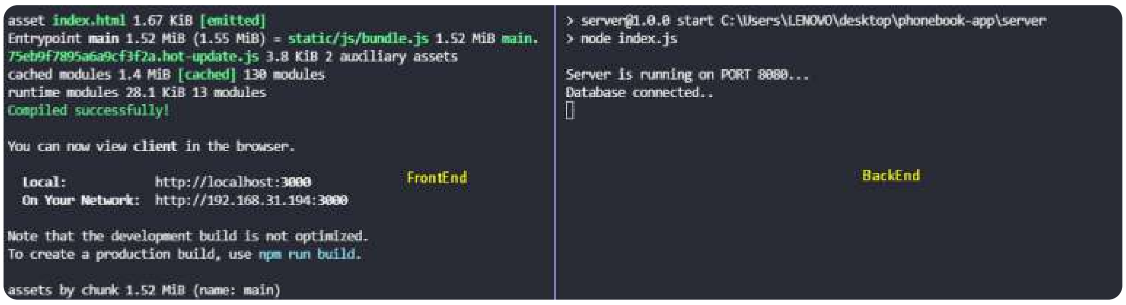
You will find some extra files in your `src` folder but to complete this project these files will be enough. Now we are ready to work with react and implement the frontend functionality.

Add data with Axios.post

At first, we will implement the adding data from the frontend. To do so we will use a medium that will help us to send a post request from the frontend to the backend. We will use `Axios` and to work with it we need to install this package. The installation process is the same, all we need to do is to run `npm i axios` command in the terminal and it will install this package for us. Now, see the below code example of implementing the adding data:

```
import React, { useState } from 'react';
import Axios from 'axios'
import './App.css';
function App() {

  const [name, setName] = useState('')
  const [phone, setPhone] = useState(0)

  const addNewNumber = () => {
    Axios.post('http://localhost:8080/add-phone',{name,phon
e})
  }

  }
  return (
    <div className="container">

        <label htmlFor="">Name: </label>
        <input type="text" onChange={(e) => {setName(e.targe
t.value)}}/><br/><br/>
        <label htmlFor="">Phone: </label>
        <input type="number" onChange={(e) => {setPhone(e.ta
rget.value)}}/><br/><br/>

        <button onClick={addNewNumber}>Add New Number</button
>

    </div>
  );
}

export default App;
```

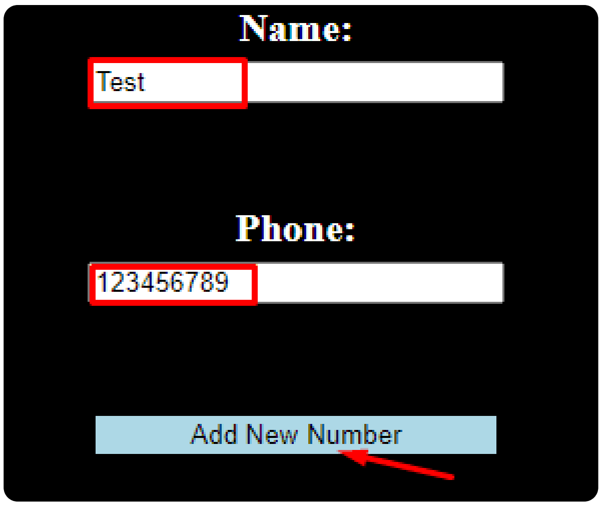
Here, at first, we have imported react, Axios, and CSS files. After that, inside the app function, we use react `useState` which will let us hold the data. Next, we define a function named `addNewNumber` and define the route that we have already implemented in the backend and simply send the data from the frontend. Later on, inside the return, we take a div and also take two input fields so that the user can input their name and phone number and finally, takes a button. We implemented those functions inside these fields so that when the user hits the Add New Number button with all the necessary data it will save the data in the database. Let's try to test it in the below section. To test it, you need to run the both app frontend and backend and you can simply run it by giving the npm start command in your terminal.



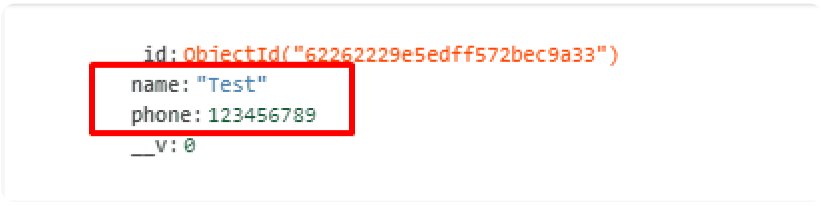
```
asset index.html 1.67 KiB [emitted]
Entrypoint main 1.52 MiB (1.55 MiB) = static/js/bundle.js 1.52 MiB main.
75eb9f7895a6a9cf3f2a.hot-update.js 3.8 KiB 2 auxiliary assets
cached modules 1.4 MiB [cached] 130 modules
runtime modules 28.1 KiB 13 modules
Compiled successfully!

You can now view client in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.31.194:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

assets by chunk 1.52 MiB (name: main)
```

```
> server@1.0.0 start C:\Users\LENOVO\desktop\phonebook-app\server
> node index.js

Server is running on PORT 8080...
Database connected..
```

You can split the terminal like this in your VSCode and run the above command in the terminal and also make sure that you are in the exact directory. Now let's try to see the output:



Here, we have set some data and sent a post request from the front end. As we haven't yet added the functionality to see the data in the frontend so we will check it from the database.



```
_id: ObjectId("62262229e5edff572bec9a33")
name: "Test"
phone: 123456789
__v: 0
```

Here, you can see that new data has been created in our database with the exact same name and phone number that we have typed in the input field. That means our first functionality is working perfectly. In the next step, we will implement the functionality of getting these data from the database and making them visible in the frontend.

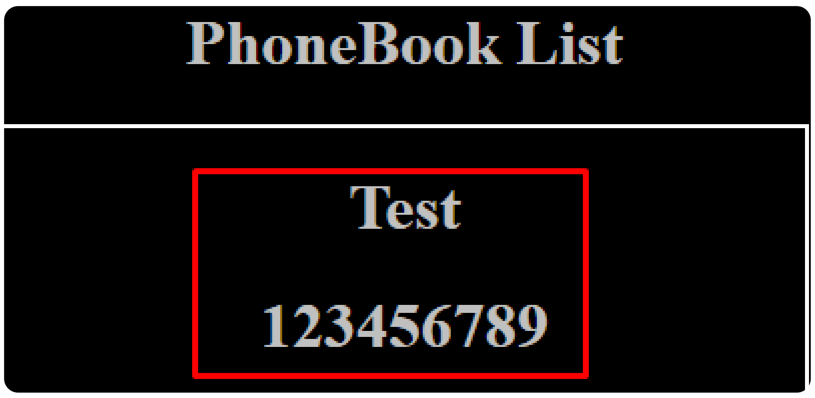
Get data with Axios.get

To see the data we need to use the `Axios.get()` request and define the route that we have set in the backend. See the below code of this implementation:

```
import React, { useEffect} from 'react';
import Axios from 'axios'
import './App.css';

function App() {

  const [phonebook, setPhonebook] = useState([])

  useEffect(() => {
    Axios.get('http://localhost:8080/get-phone').then(res =>
{
      setPhonebook(res.data.data.phoneNumbers)
    })
  },[])
  return (
    <div className="container">

      <h1>PhoneBook List</h1>
      {
        phonebook.map((val,key) => {
          return <div key={key} className="phone" >
            <h1>{val.name}</h1>
            <h1>{val.phone}</h1>
          </div>
        })
      }

    </div>

  );
}

export default App;
```

Here, we use the react useEffect and this function will return an array of elements. Remember, we have sent the

JSON request inside an extra data object and that is why we have to use es.data.data. phoneNumbers to get the access. Finally, we run a map method to make visible the data in the frontend. Now, let's check it by hitting the http://localhost:3000/get-phone route in the below section:



PhoneBook List

Test

123456789

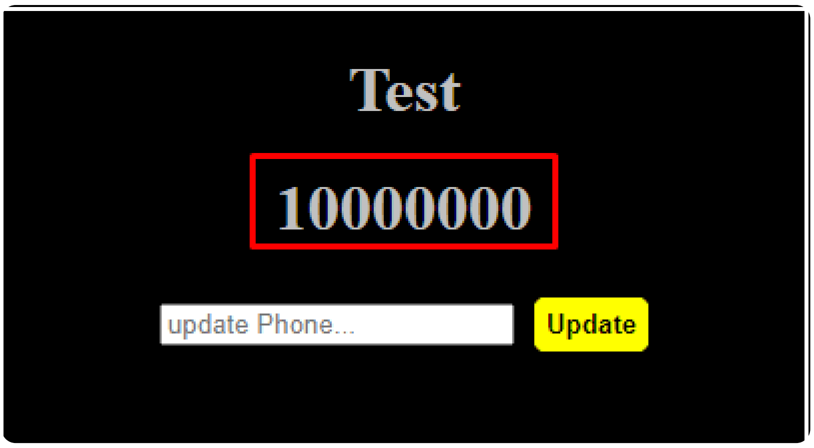
Here, you can see that are able to see the exact data from the frontend and this data has been coming from the backend. In the next step, we will implement the update functionality.

Update data with Axios.put

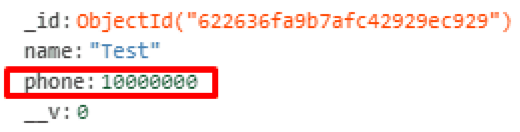
We have already implemented the update functionality in the backend. Now we will implement it from the frontend. See the below code example of doing it :

```
import React, { useState } from 'react';
import Axios from 'axios'
import './App.css';

function App() {
  const [newPhone, setNewPhone] = useState(0)

  const updatePhone = (id) =>{
    Axios.put('http://localhost:8080/update-phone',{id, newP
hone})
  }

  return (
    <div className="container">
      {
        phonebook.map((val,key) => {
          return <div key={key} className="phone" >
            <h1>{val.name}</h1>
            <h1>{val.phone}</h1>
            <input type="number" placeholder='update Phon
e...' onChange={(e) => {
              setNewPhone(e.target.value)
            }}/>
            <button className="update-btn"  onClick={() =>
{updatePhone(val._id)}}>Update</button>
          </div>
        })
      }

    </div>
  );
}

export default App;
```

Here, inside the map function, we have taken an input field and a button named it to update so that the user can update the phone number by clicking the update button. Let's check the output:



Here, you can see that our data has been updated with 10000000 instead of 123456789 Let's take a look into our database and see if there's any changes that occurred or not.



```
_id: ObjectId("622636fa9b7afc42929ec929")
name: "Test"
phone: 10000000
__v: 0
```

You can see that our data has been updated from the database also. In the next step, we will implement our last CRUD operation that is implementing the delete functionality.

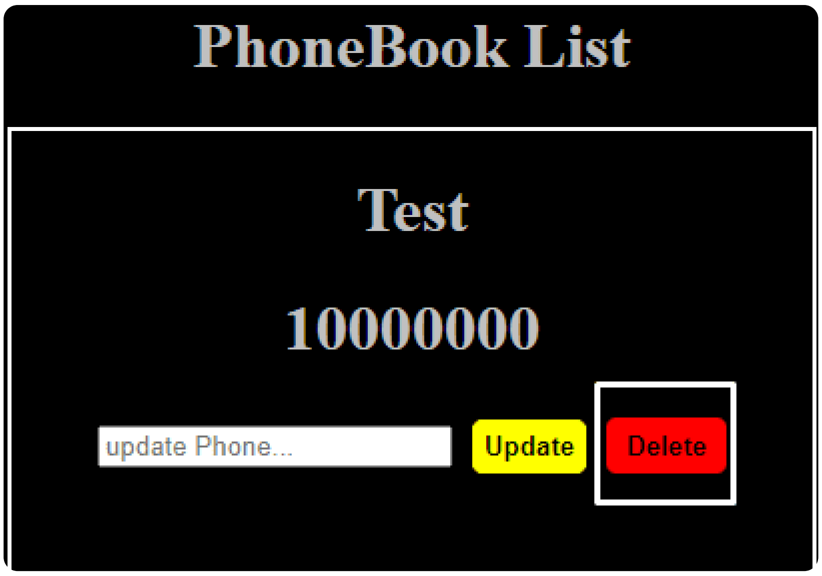
Delete data with Axios.delete

To delete the data we will make the `Axios.delete()` request and simply pass the id of the element that we want to delete. Let's see the below code:

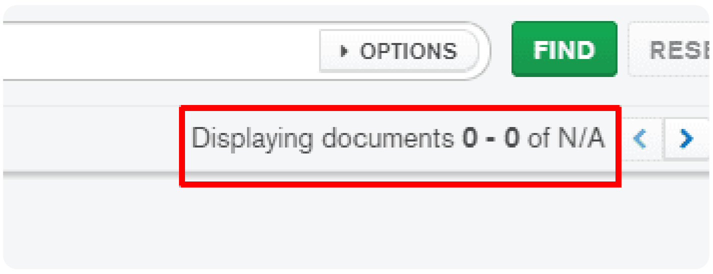
```
import React, { useState } from 'react';
import Axios from 'axios'
import './App.css';
```

```
function App() {

  const [newPhone, setNewPhone] = useState(0)

  const deletePhone = (id) => {
    Axios.delete(`http://localhost:8080/delete-phone/${id}`)
  }
  return (
    <div className="container">

      <h1>PhoneBook List</h1>
      {
        phonebook.map((val,key) => {
          return <div key={key} className="phone" >
            <h1>{val.name}</h1>
            <h1>{val.phone}</h1>
            <input type="number" placeholder='update Phon
e...' onChange={(e) => {
              setNewPhone(e.target.value)
            }}/>
            <button className='update-btn'  onClick={() =>
{updatePhone(val._id)}}>Update</button>
            <button  className='delete-btn'onClick={() =>{del
etePhone(val._id)}}>Delete</button>
          </div>
        })
      }

    </div>
  );
}

export default App;
```

Here, we have simply added the delete button and passed the id when a user clicks on the delete button it will trigger the Axios.delete() and delete the data. Let's check the output:



Here, the moment we will click on the delete button the test data will be vanished not only from the frontend but also from the Database. Let's check the database to confirm it.



Displaying documents 0 - 0 of N/A

You can see that there is no data exists in our database. With this, we have successfully completed the CRUD operation in our application.

Conclusion and Next step

We have successfully completed our CRUD operation using the MERN stack. In this whole tutorial, we have covered, how you can perform a CRUD operation not only from the backend but also from the frontend. If you have completed this tutorial, now you are ready to build your own CRUD application that will be based on the MERN stack. But there are a lot more things to do and this tutorial will give a quick kickstart about how to work with APIs, how to handle backend, frontend, and database, and also how to make a connection in between them.

Now, your task is to practice each section of this tutorial carefully. For the first time, you can practice it by seeing it and after that try to do it alone without any help. If you have any confusion, feel free to make a comment, I will try to solve your problem. It's just the beginning of your journey, there are a lot more things you can take from this tutorial. Let me tell you what will be the next step after completing this tutorial

For simplicity, I show all the data in a single file. If you want to take your knowledge to the next level, the first thing you can do is to make separate files and folders both in the backend and frontend. Later on, you can give the UI a better look. You can also implement test functionality and if you also have any suggestions you can tell me but for now, try to do these things.

Related Posts

- 20 Best Django courses for beginners