WELCOME TO CRUD OPERATION PROJECT

Overview

An ABC Company Pvt Ltd employees management app and perform Create,Read ,Update and Delete operations on the information of their employees.

About this file

The purpose of this file is to provide overview ,setup instructions and background information of the project. Any dependencies added/modified to this project which affect the running of the code will be mentioned.

Software requirements

1. Material UI APIs
2. Restful API for backend
3. Any code editor(I preferred Visual studio)
4. NPM or hyper or cmd (I prefer NPM)

Technical Specifications

- This is a react.js project in which material-table API(based on material-ui) is used for drawing tables and react js framework for frontend.
- Basic css and bootstrap 4 is used for styling.
- The react code is first compiled using npm create-react-app command and then api is connected with it through axios.
- The design for reference is from techciti company,banglore

Setup Instructions

As mentioned earlier,this is a react.js project,the below mentioned steps may very significantly across various operating system,please follow them accordingly.

8.1) Fetching git code to local step

- Clone the repository from GitLab
- Change current working directory to project directory
  (cd CRUD)
- Checking the latest development branch

8.2)Installing dependencies

- If you are using npm make sure it is above 8.16.3 version,you can check that by using npm –v
- Install node modules ,process will be npm i node modules
- Installing api with command npm install material-table --save

8.3)Developing whole project

- Start with the importing API:

Frontend — Reactjs, Material UI/Table,css

React Material UI provides us with many components (Alert, Tables, Navbar etc) for creating awesome user interfaces in React — without writing a single line of CSS code. Another popular UI framework is "react-bootstrap. As stated earlier, Material-table extends the default Material-UI table with additional cool features. Let’s get started with our users table.

Backend — Remote data source

This app relies on a free online REST API in order to abstract away the complexity of creating such an API ourselves as this is not the focus of the tutorial. We'll use https://reqres.in/ for this purpose. It provides API endpoints that allow us to make (users related) API requests such as POST (for creating a record, in our case: create a new user), GET (for fetching records), PUT/PATCH (for updating record), DELETE etc. I'd suggest that you visit the site and familiarize yourself with the various possible API calls. Let's move on to the frontend side of things.

Stylings:

As mentioned above react bootstrap is used but css is also used along with it,for body sans serif is also used,with background color #50a3a2;key frames and media tag is there to make it more remote and responsive.

The other things for tables are given below

Tables generally have rows and columns. We'll need to define columns and rows (data) for our table as below:

```
var columns = [{title: "id", field: "id", hidden: true},
{title: "Avatar", render: rowData => <Avatar maxInitials={1} size={40} round={true}
name={rowData === undefined ? " " : rowData.first_name} />  },
{title: "First name", field: "first_name"},
{title: "Last name", field: "last_name"},
{title: "email", field: "email"}]const [data, setData] = useState([]); //table data//for error handling
const [iserror, setIserror] = useState(false)
const [errorMessages, setErrorMessages] = useState([])
```

From the code snippet above, our table will have id, first name, last name, email, and avatar columns. Let's see an example JSON data that we'll receive from the RESTful API. This is shown below:

```
//JSON data from RESTful API
"user":
 {
   "id": 2,
   "email": "janet.weaver@reqres.in",
   "first_name": "Janet",
   "last_name": "Weaver",
   "avatar": " "
 }
```

Did you notice that the JSON data keys (id, email, …) exactly match the table columns — field attributes (id, email, …)? This is super important as Material-table relies on this to correctly display data from our API i.e. map API data to table columns. Hope you get the idea?

The "data" variable will store our table data in a state variable. The app (component in *react* language) re-renders (reacts) when the state changes (using the set method i.e. setData in our example). The other state variables are used for error handling. We'll touch on them later.

The next step is to fetch the data from the API so that the data variable can hold the users record instead of an empty array. Here, we will need to initiate a GET request to our API so we can receive the users list as a response from the API. Let's see how this is done next

```
useEffect(() => {
  api.get("/users")
    .then(res => {
      setData(res.data.data)
    })
    .catch(error=>{
```

```
      setErrorMessage(["Cannot load user data"])
      setIserror(true)
    })
}, [])
```

We fetch the list of users inside the useEffect react hook — the useEffect function runs every time a functional component is rendered e.g. when the page refreshes or when a state data change (in this case, the state data needs to be included in the useEffect array). The API responds to our initiated GET request with a list of users data. We then use the setData method to change the state of the data object. Now, the data variable contains the list of users instead of an empty array. On the other hand, if an error occurs, we alert the user.

So we have our data and it's just about time we display them on the table.

```
<MaterialTable
  title="User list from API"
  columns={columns}
  data={data}
  icons={tableIcons}
  editable={{
    onRowUpdate: (newData, oldData) =>
      new Promise((resolve) => {
        handleRowUpdate(newData, oldData, resolve);
  }),
  onRowAdd: (newData) =>
    new Promise((resolve) => {
      handleRowAdd(newData, resolve)
    }),
  onRowDelete: (oldData) =>
    new Promise((resolve) => {
      handleRowDelete(oldData, resolve)
    }),
  }}
/>
```

The code snippet above is responsible for displaying the table. It uses the “MaterialTable” component. We specify the title, columns and data (user list). We also include a list of icons required by the component. These icons include Arrows (for pagination), search, edit, delete icons etc.

So that takes care of the **R**ead in C**R**UD operations. Let’s move on to **C**reate operation i.e. Add a new user. Notice that the code above includes “props” for performing the other CRUD operations (create, update and delete). The “onRowAdd" prop creates a new “Promise” object (a promise returns a value in the future. This allows for handling async events .) The promise object invokes the "handleRowAdd()” function. The code snippet below provides the function implementation.

```
const handleRowAdd = (newData, resolve) => {
 //validation
 let errorList = []
 if(newData.first_name === undefined){
  errorList.push("Please enter first name")
 } if(newData.last_name === undefined){
  errorList.push("Please enter last name")
 } if(newData.email === undefined || validateEmail(newData.email) === false){
  errorList.push("Please enter a valid email")
 } if(errorList.length < 1){ //no error
  api.post("/users", newData)
   .then(res => {
    let dataToAdd = [...data];
    dataToAdd.push(newData);
    setData(dataToAdd);
    resolve()
    setErrorMessages([])
    setIserror(false)
   })
   .catch(error => {
    setErrorMessages(["Cannot add data. Server error!"])
    setIserror(true)
    resolve()
```

```
    })
  }else{
    setErrorMessages(errorList)
    setIserror(true)
    resolve()
  }
}
```

The function is as simple as it looks. First we do some validations to ensure user has completed all the required fields. Next, we send a POST request with the data we want to add (i.e. "newData"). Then, we can continue doing something else (yes, this is where "Promise" comes in — think of multithreading) until we receive a response from the server (the API). If the POST request was successful, then we update the state data using the setData() function and mark the "Promise" as resolved (see the "then" function). Otherwise, we alert the user of an error.

Let's now see how to update a given row in our table. For this, we call the "handleRowUpdate(…)" function. Its implementation is given below.

```
const handleRowUpdate = (newData, oldData, resolve) => {  //validation
  .....
  if(errorList.length < 1){
    api.patch("/users/"+newData.id, newData)
      .then(res => {
        const dataUpdate = [...data];
        const index = oldData.tableData.id;
        dataUpdate[index] = newData;
        setData([...dataUpdate]);
        resolve()
        setIserror(false)
        setErrorMessages([])
      })
      .catch(error => {
        setErrorMessages(["Update failed! Server error"])
        setIserror(true)
```

```
      resolve()
    })
  }else{
    setErrorMessages(errorList)
    setIserror(true)
    resolve()
  }
}
```

I'm pretty sure you can figure out what this function does. We initiate a PATCH request which is the API call for updating data. Note that we pass in the **id** (user id in our case) in the request URL so the API knows which record to update. If the server is able to grant our request successfully, we receive a response. Then, we update our state. Otherwise, we alert the user of error(s). The form validation is skipped since it's similar to the previous case.

So, that takes care of our **C**reate **R**ead **U**pdate operations. We are now left with the **D**elete operation. Let's look at that next.

```
const handleRowDelete = (oldData, resolve) => {
  api.delete("/users/"+oldData.id)
    .then(res => {
      const dataDelete = [...data];
      const index = oldData.tableData.id;
      dataDelete.splice(index, 1);
      setData([...dataDelete]);
      resolve()
    })
    .catch(error => {
      setErrorMessages(["Delete failed! Server error"])
      setIserror(true)
      resolve()
    })
}
```

The code snippet above provides the function for handling the delete operation. This method is very similar to the previous (PATCH). The difference here is that we initiate a DELETE request.

Deploy process

- If you are running on localhost then just simply cd into that file and run npm start.
- You can deploy globally on various platforms either heroku,git or any other.(I used git here),So the steps are as follows:
    1. Create repository name.
    2. Change the source to master.
    3. Then upload files in git which are in document form.
    4. Go to the folder and open Git bash console and follow these commands:
        - Git add .
        - Git commit –m “first commit”
        - Git remote add origin ”your url”
        - Git push –u origin master
        - At last git push –u origin master –force

You will get your rep link and all the folders will be fetched in there.

Summary

In this tutorial, you have learned how to perform CRUD operations in React using the popular Material-table for React and with data from a RESTful API.

Material-table is great and I love the out-of-the-box features such as (ease of creating, updating, deleting records or rows) that it provides. I have personally used this package for a production-level code. Another interesting feature of the package is that it is very easy to customize. You can replace the columns with a Material-UI DateTime picker or change the icons (delete, edit, search). That being said, Material-table is especially great for displaying tabular data.