**React Query – Introduction**

Let’s learn how to manage remote data, which is data stored on a external server straight from our React application.

So, far in the example, we either used local data or fetched data from an API, which is known as a server. That data was then manipulated in our front-end application.

The problem with that is that **handling data requires a lot of resources**. So, if you try to do all of your calculations in the front end, your app will probably be very slow.

Unlike front end apps, **servers as well as the databases** that are typically used on a server are **specifically designed to handle such data manipulations**, so they do it much faster. As a result, **our front-end app is still fast and performant**.

**Local Server vs Remote Server**

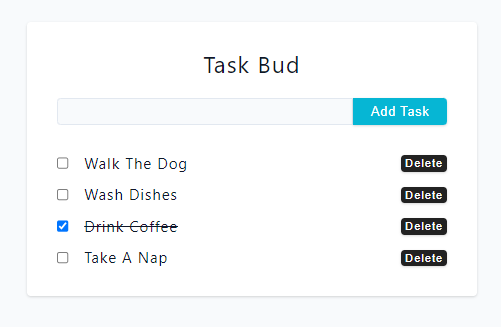
In terms of server, we can have a **local server**, or a **remote server** which is **already deployed**. It's very common to have a **local server while you're developing** because the **communication is much faster** and you can work **without the internet connection**.

**In terms of access, the** **only difference between a remote and a local server is the URL**. So, once you're done developing the application, you can deploy the server. Change the URL on the front end. And, of course, you can also after that deploy the front-end application as well.

We're not going to do any deployment in this section simply because our server doesn't have the authentication. And therefore, anyone can pretty much come to your application and write all kinds of nonsense.

**Example – Task Bud**

Our example is a “Task Bud” application, as shown below.



The goal of this example is to learn how to establish this communication between the server and our React app. At the start, we are just getting the files locally, from the object in the **data.jsx** file:

const defaultItems = [

  { id: nanoid(), title: 'walk the dog', isDone: false },

  { id: nanoid(), title: 'wash dishes', isDone: false },

  { id: nanoid(), title: 'drink coffee', isDone: true },

  { id: nanoid(), title: 'take a nap', isDone: false },

];

We will start by creating a **utils.js** file, and creating a **custom instance** there, which sets the base URL to "http://localhost:5000/api/tasks" . This is because we are going to send requests to the browser and the URL is going to start with this value.

import axios from "axios";

// Custom Instance

const customFetch = axios.create({

  baseURL: "http://localhost:5000/api/tasks",

});

export default customFetch;

**The server**

We won’t cover how to create server because that is not part of the front-end developer job, but more like backend of full-stack developers. But his is how the server code looks like. As you can see, it handles some functionality in the background. It receives the data coming from the front-end application and handles it there.

import express from 'express';

import cors from 'cors';

import { nanoid } from 'nanoid';

const app = express();

import morgan from 'morgan';

let taskList = [

  { id: nanoid(), title: 'walk the dog', isDone: false },

  { id: nanoid(), title: 'wash dishes', isDone: false },

  { id: nanoid(), title: 'drink coffee', isDone: true },

  { id: nanoid(), title: 'take a nap', isDone: false },

];

if (process.env.NODE\_ENV !== 'production') {

  app.use(morgan('dev'));

}

app.use(cors());

app.use(express.json());

app.get('/', (req, res) => {

  res.send('<h1>Hello From Server...</h1>');

});

app.get('/api/tasks', (req, res) => {

  res.json({ taskList });

});

app.post('/api/tasks', (req, res) => {

  const { title } = req.body;

  if (!title) {

    res.status(400).json({ msg: 'please provide title' });

    return;

  }

  const newTask = { id: nanoid(), title, isDone: false };

  taskList = [...taskList, newTask];

  res.json({ task: newTask });

});

app.patch('/api/tasks/:id', (req, res) => {

  const { id } = req.params;

  const { isDone } = req.body;

  taskList = taskList.map((task) => {

    if (task.id === id) {

      return { ...task, isDone };

    }

    return task;

  });

  res.json({ msg: 'task updated' });

});

app.delete('/api/tasks/:id', (req, res) => {

  const { id } = req.params;

  taskList = taskList.filter((task) => task.id !== id);

  res.json({ msg: 'task removed' });

});

app.use((req, res) => res.status(404).send('Route does not exist'));

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

const startApp = () => {

  try {

    app.listen(port, () => {

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

    });

  } catch (error) {

    console.log(error);

    process.exit(1);

  }

};

startApp();

**HTTP Methods**

HTTP (Hypertext Transfer Protocol) methods define the types of actions that can be performed on a web server to retrieve, modify or delete information. The most commonly used HTTP methods are GET, POST, PATCH and DELETE.

* GET retrieves data
* POST sends data to be processed
* PATCH update or replace existing data
* DELETE removes data.

**Get Request**

This HTTP method is used to **retrieve data from a server**. When a client sends a GET request to a server, the server will return a response that includes the requested data. This method is typically used to retrieve information from a database, to read a web page, or to download a file. The HTTP GET method is the default method used by web browsers to retrieve data from a server, as it is a safe and efficient way to request resources.

// HTTP GET example

try {

  const response = await axios.get('/api/data');

  console.log(response.data);

} catch (error) {

  console.error(error);

}

**Post Request**

The POST method is used to **send data to a server to create or update a resource**. When a client sends a POST request to a server, the server will process the request and create a new resource or update an existing one. This method is commonly used in web forms, where users enter information that is then sent to a server for processing.

// HTTP POST example

try {

  const response = await axios.post('/api/data', { name: 'John', age: 30 });

  console.log(response.data);

} catch (error) {

  console.error(error);

}

**Patch Request**

This method is similar to the POST method, but it is used to **update only a part of a resource**. When a client sends a PATCH request to a server, the server will update the resource with the new data provided in the request. This method is commonly used in REST APIs to update specific properties of a resource.

Notice the identifier (ID) in the URL (in this case 1) so the server knows what to modify. The second argument is the data we want to send to the server.

// HTTP PATCH example

try {

  const response = await axios.patch('/api/data/1', { age: 31 });

  console.log(response.data);

} catch (error) {

  console.error(error);

}

**Delete Request**

The DELETE method is used to **remove a resource from a server**. When a client sends a DELETE request to a server, the server will delete the resource if it exists. This method is commonly used in REST APIs to remove a resource that is no longer needed or to undo a previous action.

try {

  const response = await axios.delete('/api/data/1');

  console.log(response.data);

} catch (error) {

  console.error(error);

}

**API Documentation**

Ok, there are different methods, so how am I supposed to know what is the URL and what method I need to use? This is where the API documentation comes into play.

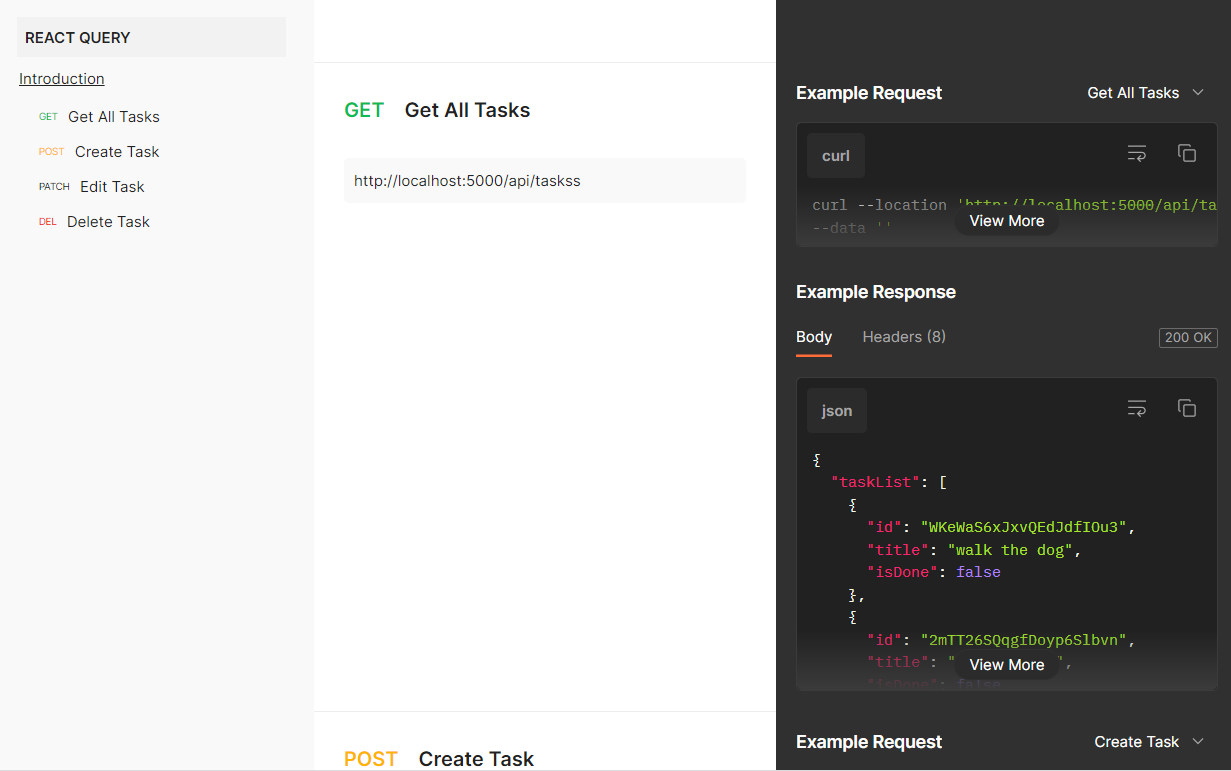
There are websites that allow to create API documentation, like the one below. On the left you can see the actions you can perform on the server, as well as what is the URL you should use.

So, every time you want to start working on a project, you will need documentation. You cannot just magically come up with your own methods. You can only perform actions that are available on the server.

Good documentation describes every detail like URL, methods and examples. Not every documentation is good though, there’s no rule for it to be, but it makes things more difficult.

You can have the internal documentation (documentation that is used within the company) or you can have a public one, so there's a public API (It's available to everyone).

It's pretty much impossible to build any kind of application without the documentation unless of course you build a server yourself. And of course, then you know exactly what are the routes as well as the methods.



In this example, we see that if we make a GET request for the URL, the server response will just retrieve the data it has.

|  |  |
| --- | --- |
| **Request** | **Response** |
|  |  |

A POST request to the server would look like this. So, when we are creating a new task, the only thing the server is looking for is the title. If the request is successful, then we will get back an object.

|  |  |
| --- | --- |
| **Request** | **Response** |
|  |  |

And so on. The PATCH and DELETE will also have the syntax detailed in the documentation, but you get the idea. We need to know how to ask for the data, and what exactly we get back.

**React Query**

**Introduction**

When working with server data, the biggest challenge is to keep your **React app states in sync with the server data**. And the bigger the app, the harder it's going to be.

So even though in this application, we can set up everything with just Axios and few use effects, it's going to be much easier if we use a library by the name of **React Query**.

**React Query is a state management library that simplifies the process of fetching, caching, and updating data in React applications**. Its major benefits include:

* Automatic background refetching, caching and stale data management
* Error handling
* Easy pagination and infinite scrolling
* Compared to setting up requests with useEffect, React Query provides a more declarative and centralized approach to managing data in React, which results in cleaner and more efficient code
* Reduces boilerplate code and improves performance by minimizing unnecessary re-renders and network requests.

So effectively the problem that I just mentioned keeping our app state in sync with server - React Query just makes it a breeze.

**Install and Setup**

Run the following command in the terminal: npm i @tanstack/react-query

Once you're done with install, you want to wrap your entire application in QueryClientProvider as well as set upa query client in the **main.jsx**.

As a result, all throughout our application, we'll have access to that new instance, and we can start setting up requests with React Query.

import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient();

const queryClient = new QueryClient();

ReactDOM.createRoot(document.getElementById('root')).render(

  <QueryClientProvider client={queryClient}>

    <App />

  </QueryClientProvider>

);

**React Query and Context**

Because React Query is a state management library, **we don’t always have to use context**. So, we can set up our queries in multiple components and we don’t need to always pass them down from **App.jsx** for example. Query provides some nice ways how we can communicate between those queries.

**React Query – example**

We want to import **useQuery** hook from React query (notice, no **useEffects**). As far as the argument, **useQuery** is looking for the object with these two properties:

* **Query Key**: The unique key you provide is used internally for re-fetching, caching, and sharing your queries throughout your application.
* **Query Function**: A query function can be literally any function that returns a promise. The promise that is returned should either resolve the data or throw an error.

import { useQuery } from '@tanstack/react-query';

const result = useQuery({

  queryKey: ['tasks'],

  queryFn: () => customFetch.get('/'),

});

console.log(result);

Consider the previous *Task Bud* application. This is the **Items.jsx** component, where it grabs the items array, iterates over and displays the SingleItem component with the matching id.

import SingleItem from "./SingleItem";

const Items = ({ items }) => {

  return (

    <div className="items">

      {items.map((item) => {

        return <SingleItem key={item.id} item={item} />;

      })}

    </div>

  );

};

export default Items;

Below is a fetch request done with React Query. Note that because we have provided the whole link with the **baseURL** in the **utils.js**, we just need to provide the forward slash "/".

import SingleItem from "./SingleItem";

import customFetch from "./utils";

import { useQuery } from "@tanstack/react-query";

const Items = ({ items }) => {

  const result = useQuery({

    queryKey: ["tasks"],

    queryFn: () => customFetch.get("/"),

  });

  console.log(result);

  return (

    <div className="items">

      {items.map((item) => {

        return <SingleItem key={item.id} item={item} />;

      })}

    </div>

  );

};

export default Items;

The response will look like the one below. It’s nested because that’s how *Axios* returns the data. So, React Query stores it in **data**, and *Axios* also stores it in **data** as well. And then we have a **taskList** because that is what is being returned from the server, which is what we eventually want to display.



The properties that we are going to use the most are:

* **data**: where our data is stored.
* **isError**: if we want to have a specific error, we will look for this property
* **isLoading**: which is going to be super useful while we are loading data

To access these properties, we can either use dot notation (e.g. **result.isLoading**) or we can destructure it. In our case, we are going to destructure.

**Render Data**

So, instead of rendering the items array, we are going to render the data we get from the server. Also, we will setup the *loading*, which is going to show “Loading…” while isLoading is **true**.

import SingleItem from "./SingleItem";

import customFetch from "./utils";

import { useQuery } from "@tanstack/react-query";

const Items = () => {

  const { isLoading, data } = useQuery({

    queryKey: ["tasks"],

    queryFn: () => customFetch.get("/"),

  });

  if (isLoading) {

    return <p style={{ marginTop: "1rem" }}>Loading...</p>;

  }

  return (

    <div className="items">

      {data.data.taskList.map((item) => {

        return <SingleItem key={item.id} item={item} />;

      })}

    </div>

  );

};

export default Items;

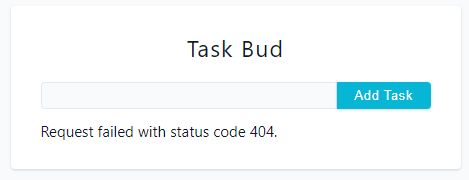
**Error Handling**

In terms or error handling, we can display an error message in the web page by doing:

  if(isError){

    return <p style={{ marginTop: "1rem" }}>there was an error...</p>;

  }

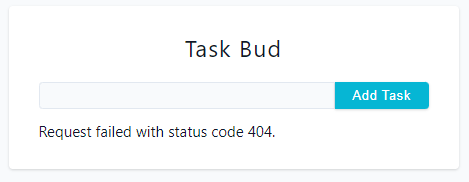


Or we can display the Axios error message

  if (error) {

    return <p style={{ marginTop: "1rem" }}>{error.message}.</p>;

  }

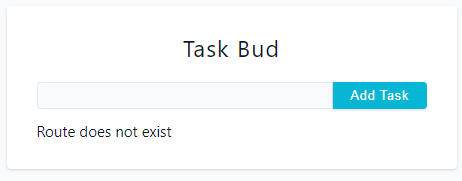


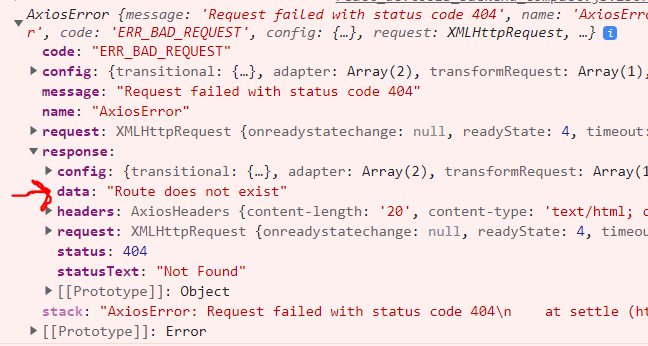
Or we can even get the error message sent from the server response.

  if (isError) {

    return <p style={{ marginTop: "1rem" }}>{error.response.data}</p>;

  }





import SingleItem from "./SingleItem";

import customFetch from "./utils";

import { useQuery } from "@tanstack/react-query";

const Items = () => {

  const { isLoading, data, isError, error } = useQuery({

    queryKey: ["tasks"],

    queryFn: () => customFetch.get("/adsd"),

  });

  if (isLoading) {

    return <p style={{ marginTop: "1rem" }}>Loading...</p>;

  }

  //display custom error message

  /\*if (isError) {

    return <p style={{ marginTop: "1rem" }}>there was an error...</p>;

  }\*/

  //display axios error message

  /\* if (error) {

    return <p style={{ marginTop: "1rem" }}>{error.message}.</p>;

  } \*/

  //display error returned from the server

  if (isError) {

    return <p style={{ marginTop: "1rem" }}>{error.response.data}</p>;

  }

  return (

    <div className="items">

      {data.data.taskList.map((item) => {

        return <SingleItem key={item.id} item={item} />;

      })}

    </div>

  );

};

export default Items;

**Post Request**

So, now we need to setup the logic where once the user provides the input and clicks on “*add task*” button, the **form is submitted and a post request is sent to the server**, and a **new task is created on the server**.

We want to set up the “create task” functionality in the Form component, because this is where we are getting the new item newItemName. So, as we’re updating the state value as we’re typing and also have the handleSubmit. So, when we submit the form, we will communicate with the server.

import { useState } from 'react';

const Form = () => {

  const [newItemName, setNewItemName] = useState('');

  const handleSubmit = (e) => {

    e.preventDefault();

  };

  return (

    <form onSubmit={handleSubmit}>

      <h4>task bud</h4>

      <div className='form-control'>

        <input

          type='text '

          className='form-input'

          value={newItemName}

          onChange={(event) => setNewItemName(event.target.value)}

        />

        <button type='submit' className='btn'>

          add task

        </button>

      </div>

    </form>

  );

};

export default Form;

When we want to **create**, **edit** and **delete** items from the server we use useMutation Hook. When we want to **fetch**, we use useQuery.

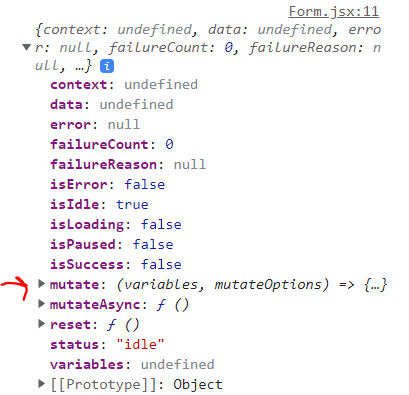
If we do a POST request as shown below, we get the following response. Bare in mind we are just hard coding the title here.

  const result = useMutation({

    mutationFn: () => customFetch.post("/", { title: "some title" }),

  });

  console.log(result);



Notice the **mutate** property in the response. That **mutate** is referencing the mutationFn funtion. In order to communicate with the server, we need to invoke this function, when submitting the form.

Again, we can use dot notation or destructure. Or, we can also change its name, which is going to become important when you have multiple functionalities in the same file, because you will want to delete and edit items, and JS rules apply, we cannot have multiple mutate variables in our application. So, we change the name from **mutate** to createTask (every time we refer to createTask, we are refering to **mutate** property).

In order for the task to be dynamic and not hard coded, we now need to pass the newItemName as parameter to our createTask function. Also, the argument in our callback function in the mutation Fucntion needs to be the same as the title value:

  const [newItemName, setNewItemName] = useState("");

  const { mutate: createTask, isLoading } = useMutation({

    mutationFn: (taskTitle) => customFetch.post("/", { title: taskTitle }),

  });

  console.log(result);

  const handleSubmit = (e) => {

    e.preventDefault();

    createTask(newItemName);

  };

**While the request is taking place, we want to disable the submit** **button**. We want to wait for the response and only when we get the response the user can create another task. We can do this by adding a disabled property to our submit button while isLoading is **true**.

        <button type="submit" className="btn" disabled={isLoading}>

At this point, we can type the value and submit it in order to show up in the task list. However, we still need to refresh the page in order for it to show. In order to solve that, useMutation comes with awesome helper options – onSuccess and onError.

const { mutate: createTask, isLoading } = useMutation({

  mutationFn: (taskTitle) => customFetch.post('/', { title: taskTitle }),

  onSuccess: () => {

    // do something

  },

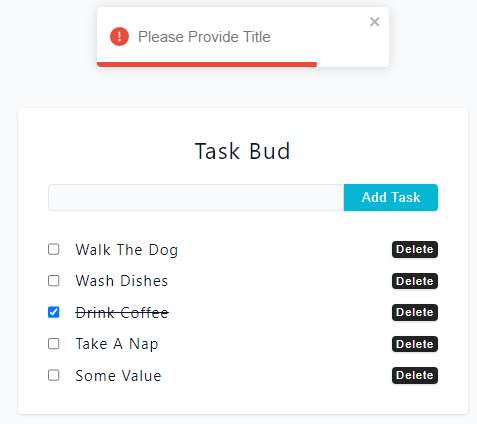
  onError: () => {

    // do something

  },

});

In terms of error, one good option to handle it is by **picking up the error message sent by our server and display is as a *toast***. For example, if we don’t provide a value on submit, the server will send a message saying “*please provide Title*”, we can then access that message and display it as a toast.



In terms of onSucess, we want to add the task without refreshing the page. That’s very hard to do if you’re just using useEffect, because you manually need to refetch those latest tasks.

With React Query, it has a method that invalidates the query. In the **main.jsx** file, we had passed the queryClient prop, so now we have access to it in the form. So, we need to import it and create a queryClient instance. And now, in the onSucess, we want to do 3 things:

1. Invalidate the tasks query set up in the **Items** component (queryKey).
2. Use the toast to display a message that the item was added.
3. Set the newItemName input to an empty string.

import { useState } from "react";

import { useMutation, useQueryClient } from "@tanstack/react-query";

import customFetch from "./utils";

import { toast } from "react-toastify";

const Form = () => {

  const [newItemName, setNewItemName] = useState("");

  const queryClient = useQueryClient();

  const { mutate: createTask, isLoading } = useMutation({

    mutationFn: (taskTitle) => customFetch.post("/", { title: taskTitle }),

    onSuccess: () => {

      queryClient.invalidateQueries({ queryKey: ["tasks"] });

      toast.success("task added");

      setNewItemName("");

    },

    onError: (error) => {

      toast.error(error.response.data.msg);

    },

  });

  const handleSubmit = (e) => {

    e.preventDefault();

    createTask(newItemName);

  };

  return (

    <form onSubmit={handleSubmit}>

      <h4>task bud</h4>

      <div className="form-control">

        <input

          type="text "

          className="form-input"

          value={newItemName}

          onChange={(event) => setNewItemName(event.target.value)}

        />

        <button type="submit" className="btn" disabled={isLoading}>

          add task

        </button>

      </div>

    </form>

  );

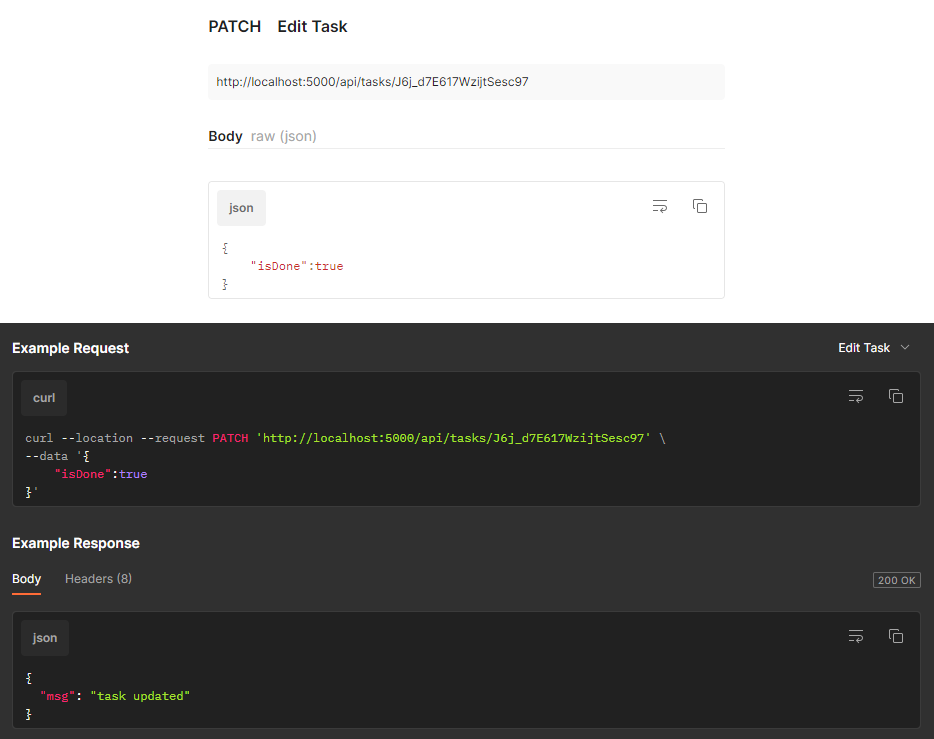
};

export default Form;

So, keeping the application state as well as the server in sync is the toughest part. It’s very easy to start coding away and realize the data on the server doesn’t match the front end. Reacy Query makes it very easy. Effectively, 2 lines of code and we’re up to date. With **useEffect**, this would be much harder.

**Edit Task**

This is what we get from the documentation regarding editing a task (basically adding a check to the checkbox):



The edit task is going to be done in the SingleItem component:

const SingleItem = ({ item }) => {

  return (

    <div className='single-item'>

      <input

        type='checkbox'

        checked={item.isDone}

        onChange={() => console.log('edit task')}

      />

      <p

        style={{

          textTransform: 'capitalize',

          textDecoration: item.isDone && 'line-through',

        }}

      >

        {item.title}

      </p>

      <button

        className='btn remove-btn'

        type='button'

        onClick={() => console.log('delete task')}

      >

        delete

      </button>

    </div>

  );

};

export default SingleItem;

By editing, it means setting isDone to **true** (check the checkbox, meaning the task is done).

We'll need to pass in two arguments in the mutationFn function:

* the ID of the task we want to edit (taskId)
* the data we want to send (isDone)

Again, we are going to invalidate the task query because we want to get the latest task.

We can combine the taskId and isDone together in an object, so now we know we’re going to get an object and we can tight away destructure it.

So, in our patch method, we need to provide the **id** and **data**, and since the property name matches our variables in the parameter, we can use shorthand.

  const { mutate: editTask } = useMutation({

    mutationFn: ({ taskId, isDone }) => {

      return customFetch.patch(`/${taskId}`, { isDone });

    },

  });

After that, we want to invoke the editTask function when the checkbox button is clicked, by setting isDone equal to the opposite of the current value. We need to set taskId equal to the id property of the item in our FE application, and the isDone property in our server equal to the isDone property in our FE application.

onChange={() => editTask({ taskId: item.id, isDone: !item.isDone })}

So, we nicely edit the item on the server, but again we want to re-fetch all the all the tasks so we can right away see the latest value that’s on the server. Again, we need to set up the query client and onSuccess.

import { useMutation, useQueryClient } from "@tanstack/react-query";

const SingleItem = ({ item }) => {

  const queryClient = useQueryClient();

  const { mutate: editTask } = useMutation({

    mutationFn: ({ taskId, isDone }) => {

      return customFetch.patch(`/${taskId}`, { isDone });

    },

    onSuccess: () => {

      queryClient.invalidateQueries({ queryKey: ["tasks"] });

    },

  });

  return (

    <div className="single-item">

      <input

        type="checkbox"

        checked={item.isDone}

        onChange={() => editTask({ taskId: item.id, isDone: !item.isDone })}

      />

      <p

        style={{

          textTransform: "capitalize",

          textDecoration: item.isDone && "line-through",

        }}

      >

        {item.title}

      </p>

      <button

        className="btn remove-btn"

        type="button"

        onClick={() => console.log("delete task")}

      >

        delete

      </button>

    </div>

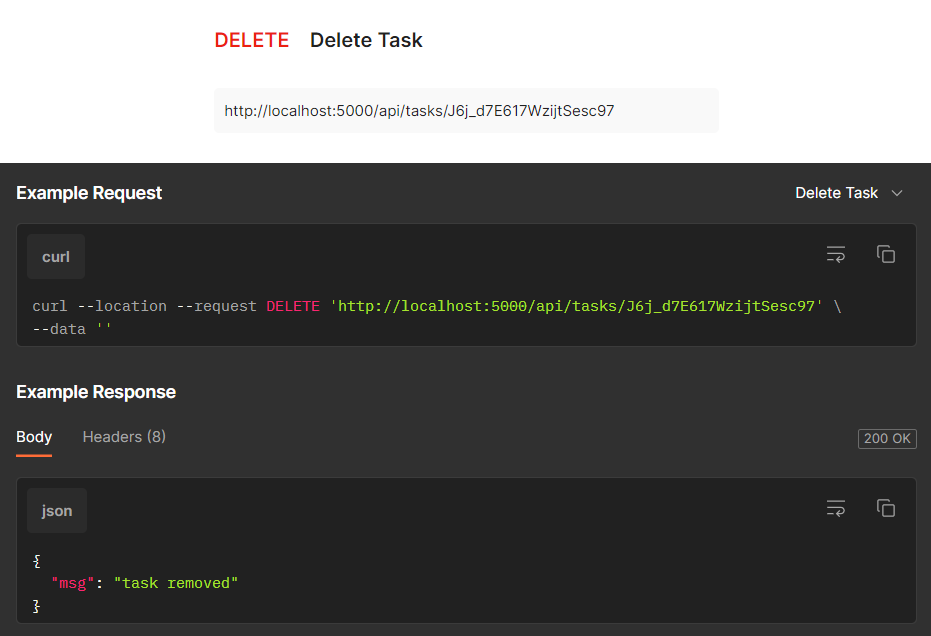
  );

};

export default SingleItem;

**Delete Task**

Check the documentation below regarding deleting a task. We don’t need to provide any values, we simply want to go with the delete method and then the ID taskId of the item we want to remove. And we’ll get back a property msg with “task removed” text.



  const { mutate: deleteTask, isLoading } = useMutation({

    mutationFn: (taskId) => {

      return customFetch.delete(`/${taskId}`);

    },

    onSuccess: () => {

      queryClient.invalidateQueries({ queryKey: ["tasks"] });

    },

  });

<button

  className="btn remove-btn"

  type="button"

  disabled={isLoading}

  onClick={() => deleteTask(item.id)}

>

  delete

</button>

**Refactoring the code**

There are way too many lines in our components, and it’s getting kinda messy. So, it might be a good idea to pass some code to a separate file.

Instead of having all these lines of code in each component to fetch data, we can set up a separate file with custom hooks. Bare in mind that we will have to create a query instance inside each hook.

Also, because we will have 2 isLoading variables sin the same file (one coming from **editTask**, and other from **deleteTask**) we are going to have to change their name.

In createTask hook, we won’t have access to setNewItemName() function. We have multiple possible approaches. We could pass the function as a hook parameter, but we will do it in another way. We will set it on onSuccess inside the handle submit function. We don’t need to import anything because the state is already created in the Form component.

* **main.jsx**: here we wrap our application in the QueryClientProvider and set up a query client, so we have access to our new QueryClient instance and we can start setting up requests with React Query.
* **App.jsx**: Where we have our components and toaster.
* **utils.js:** Where we have our custom hook customFetch with the base URL, since all the requests will have the same base URL.
* **ReactQueryCustomHooks.jsx:** Where we set up 4 custom hooks:
* useFetchTasks**:** we use this hook to fetch the tasks by providing a Query Key (unique key used internally in our app) and a Query Function (a promise that will either resolve the data or throw an error). It will get the data from the API and grab/destructure {isLoading, data, isError} and return them as {isLoading, data, isError}
* useCreateTask: here we access the createTask inside the **mutate** property from the response and destructure it right away. We also use useMutation to fetch the title property from the server. Then we set onSuccess (which invalidates de query and calls the toast) and onError (which calls the toast with the error message). In the end, returns the function createTask and the value createTaskLoading which comes from the response from the server.
* useEditTask: here we have a mutationFn which will pass the object {taskId,isDone}. It will grab the URL using the method patch, where the first argument is the URL + the item id taskId, and the second is the data sent isDone. Then we re-fetch the latest data from the server with queryClient.invalidateQueries.
* useDeleteTask: here we have a mutationFn which will pass taskId. It will grab the URL using the method delete, where the only argument is the URL + idof the item we want to delete taskId. Then we re-fetch the latest data from the server with queryClient.invalidateQueries.
* **Form.jsx:** here we have a state value newItemName which is linked to our input value (controlled input approach). Here is the functionality to create a task (POST data to the server) by using createTask which comes from our custom hook, using the newItemName as a parameter, so the new task is linked to whatever is in the input. Once the form is submitted, it clears the input field,This component also renders the form.
* **Items.jsx:** here we grab/destructure some data we have fetched from the API {isLoading, data, isError}. Display a loading message if isLoading is true; display an error message if isError is true; or if both of them are false then we access the data coming from the API and we get the item.
* **SingleItem.jsx:** here we import the mutation functions editTask, deleteTask and the state value deleteTaskLoading because this is where we are going to put our edit and delete functionalities. To edit, we trigger the editTask function when the checkbox is clicked, and set the arguments taskId to the id of the item we are iterating and the property isDone to its opposite value. If it’s successful, we want to edit the item on the server and re-fetch it, so we see the latest value on the server. So we set the query client and call the function invalidateQueries and pass in the queryKey. And he way to re-fetch Also we want to put a line though the task when it’s done. To delete the task, we simply invoke deleteTask and pass in the id of the item we want to delete.

|  |
| --- |
| **main.jsx** |
| import React from “react”;  import ReactDOM from “react-dom/client”;  import App from “./App”;  import “./index.css”;  import “react-toastify/dist/ReactToastify.css”;  import { QueryClientProvider, QueryClient } from “@tanstack/react-query”;  const queryClient = new QueryClient();  ReactDOM.createRoot(document.getElementById(“root”)).render(    <QueryClientProvider client={queryClient}>      <App />    </QueryClientProvider>  ); |
| **App.jsx** |
| import { ToastContainer } from “react-toastify”;  import Form from “./Form”;  import Items from “./Items”;  const App = () => {    return (      <section className=”section-center”>        <ToastContainer position=”top-center” />        <Form />        <Items />      </section>    );  };  export default App; |
| **Form.jsx** |
| import { useState } from “react”;  import { useCreateTask } from “./ReactQueryCustomHooks”;  const Form = () => {    const [newItemName, setNewItemName] = useState(“”);    const { createTask, createTaskLoading } = useCreateTask();    const handleSubmit = (e) => {      e.preventDefault();      createTask(newItemName, {        onSuccess: () => {          setNewItemName(“”);        },      });    };    return (      <form onSubmit={handleSubmit}>        <h4>task bud</h4>        <div className=”form-control”>          <input            type=”text “            className=”form-input”            value={newItemName}            onChange={(event) => setNewItemName(event.target.value)}          />          <button type=”submit” className=”btn” disabled={createTaskLoading}>            add task          </button>        </div>      </form>    );  };  export default Form; |
| **Items.jsx** |
| import SingleItem from “./SingleItem”;  import { useFetchTasks } from “./ReactQueryCustomHooks”;  const Items = () => {    const { isLoading, isError, data } = useFetchTasks();    if (isLoading) {      return <p style={{ marginTop: “1rem” }}>Loading…</p>;    }    //display custom error message    if (isError) {      return <p style={{ marginTop: “1rem” }}>there was an error…</p>;    }    //display axios error message    /\* if (error) {      return <p style={{ marginTop: “1rem” }}>{error.message}.</p>;    } \*/    //display error returned from the server    /\*if (isError) {      return <p style={{ marginTop: “1rem" }}>{error.response.data}</p>;    }\*/    return (      <div className="items">        {data.data.taskList.map((item) => {          return <SingleItem key={item.id} item={item} />;        })}      </div>    );  };  export default Items; |
| **SingleItem.jsx** |
| import { useDeleteTask, useEditTask } from "./ReactQueryCustomHooks";  const SingleItem = ({ item }) => {    const { editTask } = useEditTask();    const { deleteTask, deleteTaskLoading } = useDeleteTask();    return (      <div className="single-item">        <input          type="checkbox"          checked={item.isDone}          onChange={() => editTask({ taskId: item.id, isDone: !item.isDone })}        />        <p          style={{            textTransform: "capitalize",            textDecoration: item.isDone && "line-through",          }}        >          {item.title}        </p>        <button          className="btn remove-btn"          type="button"          disabled={deleteTaskLoading}          onClick={() => deleteTask(item.id)}        >          delete        </button>      </div>    );  };  export default SingleItem; |
| **ReactQueryCustomHooks.jsx** |
| import { useQuery, useMutation, useQueryClient } from "@tanstack/react-query";  import customFetch from "./utils";  import { toast } from "react-toastify";  export const useFetchTasks = () => {    const { isLoading, data, isError } = useQuery({      queryKey: ["tasks"],      queryFn: () => customFetch.get("/"),    });    return { isLoading, data, isError };  };  export const useCreateTask = () => {    const queryClient = useQueryClient();    const { mutate: createTask, isLoading: createTaskLoading } = useMutation({      mutationFn: (taskTitle) => customFetch.post("/", { title: taskTitle }),      onSuccess: () => {        queryClient.invalidateQueries({ queryKey: ["tasks"] });        toast.success("task added");      },      onError: (error) => {        toast.error(error.response.data.msg);      },    });    return { createTask, createTaskLoading };  };  export const useEditTask = () => {    const queryClient = useQueryClient();    const { mutate: editTask } = useMutation({      mutationFn: ({ taskId, isDone }) => {        return customFetch.patch(`/${taskId}`, { isDone });      },      onSuccess: () => {        queryClient.invalidateQueries({ queryKey: ["tasks"] });      },    });    return { editTask };  };  export const useDeleteTask = () => {    const queryClient = useQueryClient();    const { mutate: deleteTask, isLoading: deleteTaskLoading } = useMutation({      mutationFn: (taskId) => {        return customFetch.delete(`/${taskId}`);      },      onSuccess: () => {        queryClient.invalidateQueries({ queryKey: ["tasks"] });      },    });    return { deleteTask, deleteTaskLoading };  }; |
| **utils.js** |
| import axios from "axios";  // Custom Instance  const customFetch = axios.create({    baseURL: "http://localhost:5000/api/tasks",  });  export default customFetch; |

**Server – Storing the values**

Until now we have just been using an array of hard coded values every time we start the server. This is going to overwrite our data each time the server starts and start from scratch.

let taskList = [

  { id: nanoid(), title: 'walk the dog', isDone: false },

  { id: nanoid(), title: 'wash dishes', isDone: false },

  { id: nanoid(), title: 'drink coffee', isDone: true },

  { id: nanoid(), title: 'take a nap', isDone: false },

];

But there's another approach where essentially we can save it to a local file system, so every time we restart the server, all our items will be saved.

To do that:

1. go to package.json and find the local-server command.
2. Open the intergrated terminal and run npm run local-server

Now of course, normally you'll save this to a database and again, the result is going to be exactly the same, where every time you'll start up your server, the data is going to be already saved. But in this case, since I did not want to connect to a database, I just use the local file system.