Yes, using React Query to call an API is often a preferred approach for managing server state in React applications. React Query provides powerful features like caching, automatic refetching, background updates, and better error handling. Here’s how you can set up API calls with React Query in a React + Vite + TypeScript project.

1. Install React Query

First, add @tanstack/react-query to your project:

npm install @tanstack/react-query

2. Set Up the Query Client Provider

To use React Query in your app, you need to set up the QueryClientProvider at the root of your application.

In src/main.tsx, wrap your app with QueryClientProvider:

import React from 'react';

import ReactDOM from 'react-dom/client';

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

import App from './App';

const queryClient = new QueryClient();

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

<React.StrictMode>

<QueryClientProvider client={queryClient}>

<App />

</QueryClientProvider>

</React.StrictMode>

);

3. Use React Query in a Component

Now, let’s create a component that fetches data using useQuery from React Query. This example will automatically handle caching, background updates, and error handling for us.

Example: Fetching Data with React Query

1. Create the API function: Define a separate function for your API call.

In src/api.ts, set up the fetchData function:

import axios from 'axios';

export type DataType = {

id: number;

title: string;

body: string;

};

export const fetchData = async (): Promise<DataType[]> => {

const response = await axios.get<DataType[]>('http://localhost:5000/api/data');

return response.data;

};

2. Use the API function in a component with useQuery:

Now, create a component that uses React Query’s useQuery hook to fetch the data and manage loading and error states automatically.

import React from 'react';

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

import { fetchData, DataType } from '../api';

const ApiComponent: React.FC = () => {

const { data, error, isLoading, isError } = useQuery<DataType[], Error>({

queryKey: ['data'], // Unique key to identify this query

queryFn: fetchData, // Function to fetch the data

});

if (isLoading) return <p>Loading...</p>;

if (isError) return <p>Error: {error?.message}</p>;

return (

<div>

<h1>Data from Server</h1>

<ul>

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

<li key={item.id}>

<h3>{item.title}</h3>

<p>{item.body}</p>

</li>

))}

</ul>

</div>

);

};

export default ApiComponent;

Key Benefits of Using React Query

• Caching: React Query caches the data by default, so it doesn’t need to refetch the data on every component mount.

• Automatic Refetching: React Query refetches the data in the background when the component mounts again or after certain events (e.g., window focus).

• Optimistic Updates: React Query supports optimistic updates, which can improve the user experience by updating the UI before the server responds.

• Error Handling and Retry: Built-in error handling with retry options for failed requests.

• Polling and Stale-While-Revalidate: Supports polling and re-fetching stale data when a component re-mounts or window focus is regained.

Optional: Add Configuration Options

React Query allows you to configure options such as staleTime, cacheTime, and retry strategies.

For example, to set a staleTime (so data won’t refetch within a certain timeframe), add it to useQuery:

const { data, error, isLoading, isError } = useQuery<DataType[], Error>({

queryKey: ['data'],

queryFn: fetchData,

staleTime: 1000 \* 60 \* 5, // 5 minutes

});

Summary

Using React Query is generally preferred for complex state management needs and larger applications because it simplifies data fetching and improves performance through caching, background updates, and error handling. This makes it a powerful tool for managing asynchronous data in React apps.