# TANSCTAK/React query

Installation

### 2. TanStack React Query installations

TanStack React Query is used for data fetching, caching, and synchronization in React applications.

bash

Copy code

**npm install @tanstack/react-query**

# basic setup after tanstack/react query

Here are the installation commands for the commonly used React packages you mentioned, along with their brief descriptions:

**1. React Router DOM**

React Router DOM is used for client-side navigation in React applications.

bash

Copy code

npm install react-router-dom

**2. TanStack React Query**

TanStack React Query is used for data fetching, caching, and synchronization in React applications.

bash

Copy code

**npm install @tanstack/react-query**

**3. React Hook Form**

React Hook Form is a library for building forms in React with easy validation and improved performance.

bash

Copy code

npm install react-hook-form

**4. Axios (Optional)**

For making HTTP requests, you can use Axios.

bash

Copy code

npm install axios

**5. Zustand (Optional)**

For state management in React applications.

bash

Copy code

npm install zustand

**Dev Server**

If you are using Vite (which you are), after installing these packages, you can run your application using:

bash

Copy code

npm run dev

**Additional Commands**

* **Installing all at once**:

bash

Copy code

npm install react-router-dom @tanstack/react-query react-hook-form axios zustand

Let me know if you need setup instructions or example usage for any of these libraries!

# Steps-to work with tansatck query

Step -1 **npm install @tanstack/react-query**

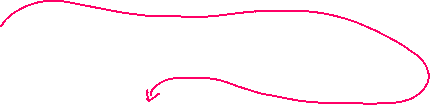
Step-2 goto app.js or app.jsx or main.jsx and wrap with queryclientprocider referenced with queryclient

First import both

For vite users its main.jsx and plain react its index.jsx

Im gonna use main.jsx

import { StrictMode } from 'react'  
import { createRoot } from 'react-dom/client'  
import {QueryClientProvider,QueryClient,} from "@tanstack/react-query";  
import './index.css'  
import App from './App.jsx'  
  
  
const queryclient=new QueryClient();  
  
createRoot(*document*.getElementById('root')).render(  
  
 <StrictMode>  
<QueryClientProvider client={queryclient}>  
 <App />  
</QueryClientProvider>  
 </StrictMode>  
)







Now to start using it on any component and start working with apis

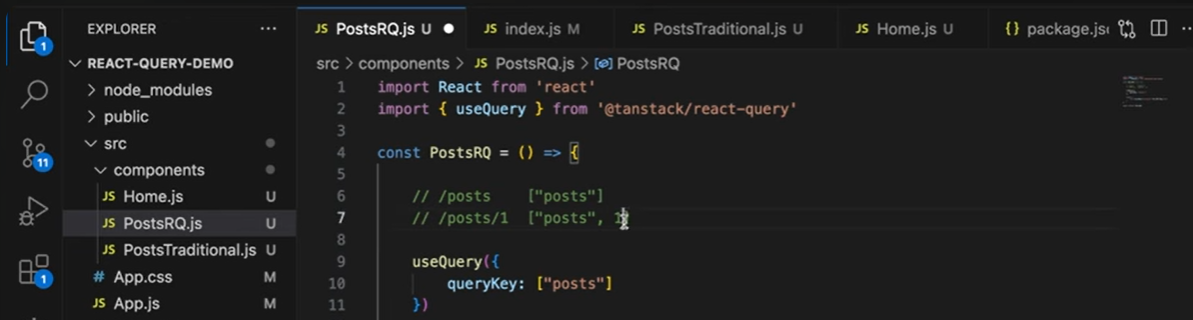




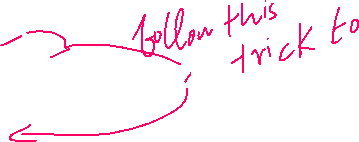
# Using React query to fetch data from apis

We need query key and queryfn

Query key i.e unique key







Now example code:

Steps :fetch the data using axios

Step 2:thn destructure usequery and use querykey and queryfn(function ref)

Querykey if no params or stats are added just the routename,if getby id or similar means use two key,one is routename and id

This indicates thn whenever id is added the refetching happens

The use of key is to keep usequery updated on any change

import React, { useState } from 'react';  
import { useQuery } from "@tanstack/react-query";  
import *axios* from "axios";  
import Card from "./Card.jsx";  
import Button from "./Button.jsx";  
import {NavLink} from "react-router-dom";  
  
const Posts = () => {  
 const [singleProduct, setSingleProduct] = useState(0);  
*console*.log(singleProduct);  
 // Fetch posts based on the selected product  
 const fetchPosts = async () => {  
 const url = singleProduct !== 0  
 ? `http://localhost:4000/posts/${singleProduct}` // Single product URL  
 : 'http://localhost:4000/posts'; // All products URL  
 const response = await axios.get(url);  
 return response.data;  
 };  
  
 // Fetch data using React Query  
 **const { data, isLoading, isError, error } = useQuery({  
 queryKey: ["posts", singleProduct],//if you are trying to fetch single product then also metion that state  
 queryFn: fetchPosts  
 });**  
  
 // Loading and Error handling  
 if (isLoading) {  
 return <h1>Loading...</h1>;  
 }  
  
 if (isError) {  
 return <h1>{error.message}</h1>;  
 }  
  
  
  
 return (  
 <div className="flex flex-wrap w-screen h-screen justify-center relative">  
 <div className="flex flex-col left-0 top-8 gap-2 fixed">  
 {[0,1, 2, 3, 4, 5].map((categ, index) => (  
 <Button key={index} cate={categ} onClick={() => {  
 *console*.log("Button clicked:", categ); // Check if button click is working  
 setSingleProduct(categ); // Set selected product ID  
 }} />  
 ))}  
 </div>  
  
 {/\* Render data based on whether it is an array or a single object \*/}  
 {*Array*.isArray(data) ? (  
 data.length > 0 ? (  
 data.map((post) => <NavLink key={post.id} to={`/productDetails/${post.id}`} ><Card key={post.id} {...post} /></NavLink>)  
 ) : (  
 <h1>No products found</h1>  
 )  
 ) : (  
 data ? <Card key={data.id} {...data} /> : <h1>No product found</h1>  
 )}  
 </div>  
 );  
};  
  
export default Posts;

2nd example

import React from 'react'  
import {useParams} from "react-router-dom";  
import {useQuery} from "@tanstack/react-query";  
import *axios* from "axios";  
  
const ProductDetailPage = () => {  
 const {id}=useParams() //extract id from url  
  
 const fetchproductdata=async()=>{  
 const response=await axios.get(`http://localhost:4000/posts/${id}`);  
 return response.data;  
 }  
  
 const {data,isError,isLoading,error}=useQuery({  
 queryKey:["productId",id],  
 queryFn: fetchproductdata  
 });  
  
 if(isLoading){  
 return <h1>Loading...</h1>  
 }else if(isError){  
 return <h1>{error.message}</h1>  
 }  
  
  
 return (  
 <div className="flex p-4 w-full h-screen items-center justify-center">  
  
 <div className="flex w-full h-[18rem] items-center justify-center box-border m-0 p-0">  
 <div  
 className="text-sm w-[14rem] text-center flex flex-col h-full justify-center items-center border-2 p-2">  
 <h1>{data.title}</h1>  
 <img className="h-[14rem] w-full" src={data.image} alt={data.title}/>  
 </div>  
 <div className="flex flex-col bg-red-100 gap-3 h-fit w-[20rem] items-center border-4 rounded-md p-2">  
 <h2 className="font-poppins first-letter:uppercase font-semibold">{data.*category*}</h2>  
 <p className="font-poppins font-light text-sm">{data.description}</p>  
 </div>  
 </div>  
  
 </div>  
 )  
}  
export default ProductDetailPage

# DEVTOOLS

# Cache and slatetime

1. **1. Cache in React Query:**

* **Definition**: React Query maintains a cache of query results so that data can be stored and accessed without needing to refetch it every time a component re-renders.
* **Benefits**:
  + **Reduced Network Requests**: Data is stored locally for a period of time, reducing the need for repeated API calls.
  + **Faster Performance**: Components can render cached data instantly while fetching updated data in the background.
* **Cache Lifecycle**:
  + **Initial Fetch**: When useQuery is run for the first time, it fetches data from the server and stores it in the cache.
  + **Subsequent Renders**: If the same query key is used again and the cache is still valid, React Query will serve the cached data.
  + **Automatic Garbage Collection**: React Query automatically removes unused data from the cache after a certain period, defined by cacheTime.
* **Configuration**:
  + cacheTime: The duration (in milliseconds) that data remains in the cache before it is garbage collected. Default is 5 minutes (300000 ms).

javascript

Copy code

const { data } = useQuery("posts", fetchPosts, {

cacheTime: 1000 \* 60 \* 10, // Cache data for 10 minutes

});

1. **2. Stale State in React Query:**

* **Definition**: Data fetched by useQuery can be either **fresh** or **stale**. Fresh data is considered up-to-date, while stale data might be outdated but can still be shown while fetching updated data.
* **Benefits**:
  + **Immediate Display**: Stale data allows your app to display previously fetched data immediately while checking for new data in the background.
  + **Optimized User Experience**: Users see data instantly without waiting for network requests to complete.
* **Stale Time**:
  + staleTime determines how long fetched data is considered **fresh**. After this period, the data becomes **stale**.
  + The default staleTime is 0, meaning data is immediately considered stale after fetching.

javascript

Copy code

const { data } = useQuery("posts", fetchPosts, {

staleTime: 1000 \* 60 \* 5, // Data stays fresh for 5 minutes

});

* **Refetching Stale Data**: When a component mounts and the data is stale, React Query will refetch the data in the background, ensuring the user sees up-to-date information as soon as possible.

1. **Example Use Case:**

Consider a situation where you display a list of posts, and you want the data to be cached for 10 minutes but only considered fresh for the first 5 minutes. This configuration ensures:

* Users see instant data if they revisit the page within 5 minutes.
* If they come back after 5 minutes but before 10 minutes, the cached data is shown while a background fetch updates it.
* If it’s been more than 10 minutes, the cache is cleared, and a new fetch is triggered.

javascript

Copy code

const { data, isLoading, isFetching } = useQuery("posts", fetchPosts, {

cacheTime: 1000 \* 60 \* 10, // Cache data for 10 minutes

staleTime: 1000 \* 60 \* 5, // Data stays fresh for 5 minutes

});

1. **Key Points:**

* **cacheTime** impacts how long the data remains in memory before being garbage collected.
* **staleTime** affects when data is considered stale and triggers background refetching when components re-mount or the window is refocused.

By understanding and configuring these properties, you can improve data freshness, reduce server load, and enhance the user experience.

# Query on click

We need to destructure refetch,and add enabled:false

Now just call thet refetch fuction in onclick event

Fetch the data as usual

Now in useQuery ,destructure the **refetch** function

And set enables:false

Now in the button onclick event call the refetch function or its reference

import *axios* from "axios";  
import {useQuery} from "@tanstack/react-query";  
import {NavLink} from "react-router-dom";  
import Card from "./Card.jsx";  
import {useState} from "react";  
  
const FetchOnClick = () => {  
  
  
 // Fetch posts based on the selected product  
 const fetchPosts = async () => {  
 const url = 'http://localhost:4000/posts'; // All products URL  
 const response = await axios.get(url);  
 return response.data;  
 };  
  
 // Fetch data using React Query  
 const { data, isLoading, isError, error,**refetch** } = useQuery({  
 queryKey: ["posts"],//if you are trying to fetch single product then also metion that state  
 queryFn: fetchPosts,  
 **enabled:false  
 });**  
  
 // Loading and Error handling  
 if (isLoading) {  
 return <h1>Loading...</h1>;  
 }  
  
 if (isError) {  
 return <h1>{error.message}</h1>;  
 }  
  
//what if i wanna toggle display on click and not display on click  
 const [display, setDisplay] = useState(true);  
  
 return (  
 <div className="flex flex-wrap w-screen h-screen justify-center">  
 <button className="bg-slate-800 p-3 w-fit h-fit font-poppins font-[900] text-slate-300" **onClick={refetch**}>fetch post</button> {/\*//directly call the reftech reference\*/}  
 { data?.map((post) => <NavLink key={post.id} to={`/productDetails/${post.id}`} ><Card key={post.id} {...post} /></NavLink>)  
 }  
 </div>  
 );  
}  
export default FetchOnClick

toggling using function to toggle and using refetch function,inside of handlediplay

import *axios* from "axios";  
import {useQuery} from "@tanstack/react-query";  
import {NavLink} from "react-router-dom";  
import Card from "./Card.jsx";  
import {useState} from "react";  
  
const FetchOnClick = () => {  
  
  
 // Fetch posts based on the selected product  
 const fetchPosts = async () => {  
 const url = 'http://localhost:4000/posts'; // All products URL  
 const response = await axios.get(url);  
 return response.data;  
 };  
  
 // Fetch data using React Query  
 const { data, isLoading, isError, error,**refetch** ,isFetching} = useQuery({  
 queryKey: ["posts"],//if you are trying to fetch single product then also metion that state  
 queryFn: fetchPosts,  
 **enabled:false**  
 });  
  
 // Loading and Error handling  
 if (isLoading) {  
 return <h1>Loading...</h1>;  
 }  
  
 if (isError) {  
 return <h1>{error.message}</h1>;  
 }  
  
//what if i wanna toggle display on click and not display on click  
 const [display, setDisplay] = useState(false);  
 const handleToggle=(e)=>{  
 setDisplay(!display);  
 **refetch();  
 }**  
 //when you use state here outside of return statement it wont work  
 //use the below logic state for set button in return statement..  
 //use state get updated in return statement  
  
  
  
 return (  
  
 <div className="flex flex-wrap w-screen h-screen justify-center">  
 <button className="bg-slate-800 p-3 w-fit h-fit font-poppins font-[900] text-slate-300" onClick={handleToggle}>{display?"hide data":"fetch data"}</button> {/\*//directly call the reftech reference\*/}  
 { display && data?.map((post) => <NavLink key={post.id} to={`/productDetails/${post.id}`} ><Card key={post.id} {...post} /></NavLink>)  
 }  
 </div>  
 );  
}  
export default FetchOnClick

# Polling or refetching

Use this only for fetching data

The properties refetchInterval and refetchIntervalInBackground in React Query are used to periodically refetch data in the background. Here’s a breakdown of each:

1. **refetchInterval**

* **Purpose**: Sets an interval for how often a query should be refetched automatically. In this case, refetchInterval: 1000 means the query will be refetched every 1 second (1000 ms).
* **Usage**: Useful for real-time data updates, like live stock prices, dashboards, or notifications.
* **Syntax**:

javascript

Copy code

refetchInterval: 1000 // Refetches data every second

1. **refetchIntervalInBackground**

* **Purpose**: Determines whether the query should continue to refetch even when the app is in the background (e.g., when the browser tab is not active).
* **Usage**: Set to true if you want the app to keep fetching in the background, which is helpful if the data needs to stay up-to-date continuously. However, it can consume more bandwidth and battery, so it’s typically used selectively.
* **Syntax**:

javascript

Copy code

refetchIntervalInBackground: true // Keeps refetching even in the background

1. **Example Usage in a Query**

javascript

Copy code

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

queryKey: ['posts'],

queryFn: fetchPosts,

refetchInterval: 1000, // Fetches data every second

refetchIntervalInBackground: true, // Fetches even in the background

});

This configuration ensures that the data is refreshed every second, regardless of whether the user actively views the tab or app.

# UseMutation

Mutate fuction of usequery is always connected to asyc fuction which has api

# Post Mapping or adding the data api

Donot use use state and thn use it to mutate the posting data,

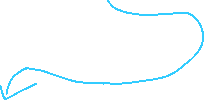
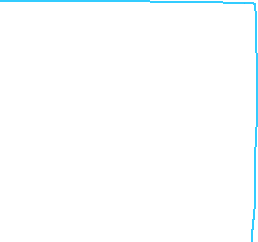
Since usestate works asynchronously it doesn’t pic objects immediately..as it takes time to get updated

So directly insert form Data to mutate(formdata) to trigger the post mapping

Step1:import refrence of queryClient use its hook useQueryClient

Step2:u need this to invlaidateQuery(querykey) which take query key to trigger the get mapping onve data is posted use correct key

import React, {useState} from 'react'  
import {useForm} from "react-hook-form";  
import {useMutation,useQueryClient} from "@tanstack/react-query";  
import *axios* from "axios";  
  
const AddDetails = () => {  
 //no need if on change event if you are using react hoook form  
 //register is also and object which keep appending details once automatically,so use {...register()} with inputs spread operator  
 const {handleSubmit,register,reset}=useForm()  
 const [posts, setPosts] = useState({});  
 **const handleSubmitData=(productdata)=>{  
 *console*.log(productdata)  
 setPosts(prev=>prev=productdata);//this usstate works Asynchronously so might be empty  
 *console*.log(posts) //donot use state in such scinarios  
 //here call the mutate funtion mutate() or its alias name if you ahev given any ,i have given  
 postdata(productdata);//this will pic parameters,pass form data directly not with state  
 reset();  
 }** //for post update del you need query client obj  
 **const queryClient = useQueryClient();**  
 **//now call the api with seperate fuction to make it modular  
 const addPosts=async(productdata)=>{  
 const add=await axios.post("http://localhost:4000/posts",productdata)  
 }** //use mutations  
 //give alias name for mutate if you want  
 //next call the mutate function  
 **const {mutate:postdata,isSuccess,isError,error}=useMutation({  
 //remeber to call mutate function in hadnle submit to trigger post api  
 mutationFn:addPosts,  
 onSuccess: () => {  
 queryClient.invalidateQueries(["posts"]); // Matches default state in `Posts`  
 },  
 });**  
 return (  
 <div className="bg-slate-800">  
 <form className="flex flex-col"  
 onSubmit={handleSubmit(handleSubmitData)}> {/\*//the data will be automatically passed by react hook for the fuction defined in handle submit\*/}  
 <input placeholder="Enter title" {...register('title')} type="text"/>  
 <input placeholder="Enter price" {...register('price')} type="text"/>  
 <input placeholder="Enter decription" {...register('description')} type="text"/>  
 <input placeholder="Enter category" {...register('category')} type="text"/>  
 <input placeholder="Enter imageUrl" {...register('image')} type="text"/>  
 <button type="submit" className="bg-blue-500 text-white py-2 px-4 rounded">  
 Submit  
 </button>  
 </form>  
 </div>  
  
 )  
}  
export default AddDetails



# Similary for del post

Combination of fetching on click,delete invalidate query

import *axios* from "axios";  
import {useMutation, useQuery, useQueryClient} from "@tanstack/react-query";  
import Card from "./Card.jsx";  
import {useState} from "react";

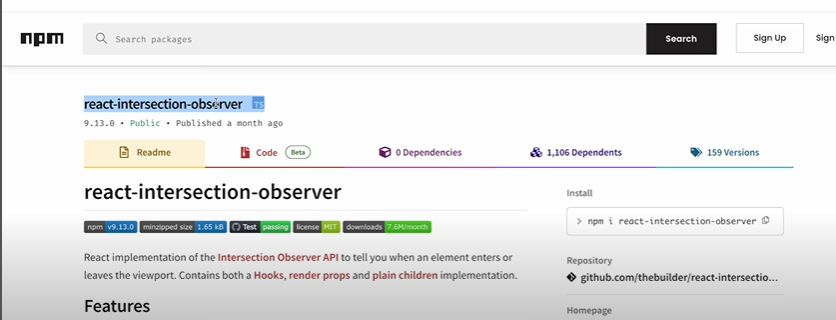
const FetchOnClick = () => {  
  
  
 // Fetch posts based on the selected product  
 const fetchPosts = async () => {  
 const url = 'http://localhost:4000/posts'; // All products URL  
 const response = await axios.get(url);  
 return response.data;  
 };  
  
 // Fetch data using React Query  
 const { data, isLoading, isError, error,refetch} = useQuery({  
 queryKey: ["posts"],//if you are trying to fetch product then also mention that state  
 queryFn: fetchPosts,  
 // enabled:false  
 });  
  
  
//what if i wanna toggle display on click and not display on click  
 const [display, setDisplay] = useState(false);  
 const handleToggle=()=>{  
 setDisplay(!display);  
 // refetch();  
 }  
  
 //when you use state here outside of return statement it wont work  
 //use the below logic state for set button in return statement..  
 //use state get updated in return statement  
  
 **//for deletion-------------------------------------------  
 const deletePost=async(prductId)=>{  
 const url='http://localhost:4000/posts/'+prductId;  
 return await axios.delete(url);  
 }  
 const queryClient=useQueryClient();//this is to invalidate and trigure the change on success after del to auto fetch-----  
  
 const {mutate:deldata,isSuccess:success,data:deleteddata,isError:dataerror,error:message}=useMutation({  
 mutationFn:deletePost,  
 onSuccess:()=>{  
 queryClient.invalidateQueries("posts")  
 }  
 })  
  
 const handleDelete=(id)=>{  
 deldata(id);  
 *console*.log(deleteddata)  
 if(success){  
 alert("deleted: "+id+" sucesfully deleted")  
 }else if(dataerror){  
 alert("error: "+dataerror)  
 }  
 }**  
 return (  
  
 <div className="flex flex-wrap w-screen h-screen justify-center pt-[4em] bg-slate-800 overflow-y-auto">  
 <button className="bg-slate-800 p-3 w-fit h-fit font-poppins font-[900] text-slate-300 fixed top-3 z-30 shadow-slate-600 shadow-md border-2 rounded-md" onClick={handleToggle}>{display?"hide data":"fetch data"}</button> {/\*//directly call the reftech reference\*/}  
 {isLoading ? (  
 <h1>Loading...</h1>  
 ) : isError ? (  
 <h1>{error.message}</h1>  
 ) : (  
 display &&  
 data?.map((post) => (  
 // <NavLink key={post.id} to={`/productDetails/${post.id}`}>  
 //  
 // </NavLink>  
 //here use button to get product id  
 <div className="flex w-fit flex-col bg-slate-800 m-1 rounded-md" key={post.id}>  
 <Card {...post} handleDelete={handleDelete} postId={post.id}/> {/\*//now get the id from the card\*/}  
 <button className="bg-slate-700 rounded-sm w-fit self-center px-4 shadow-red-500 shadow-sm inset-4 drop-shadow-md" onClick={()=>handleDelete(post.id)}>🗑️</button>  
 </div>  
  
 ))  
 )}  
 </div>  
 );  
}  
export default FetchOnClick

# infinite query on Scroll

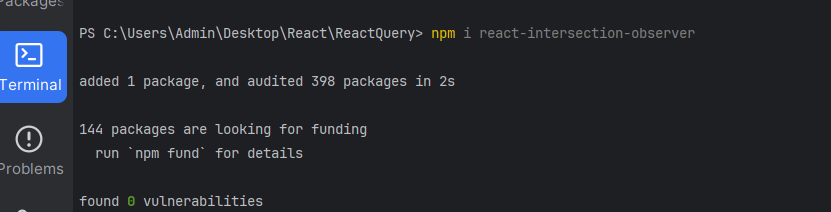
we need the intersection observe our scrolling

install it

npm i react-intersection-observer







2.import useInView HOOK from react-intersection-observer

//import useInView hook from react-intersection-observer

Now use this hook



3.Destructure the objects in useInView hook

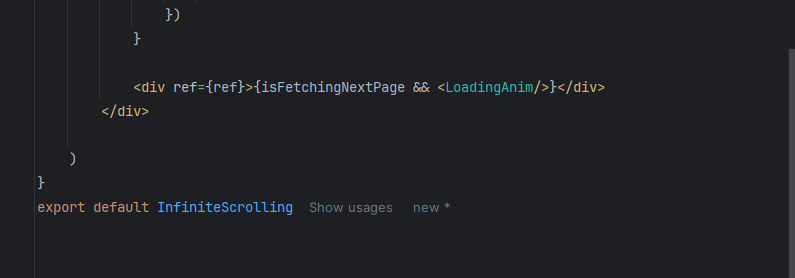
Provides three properties to use

i.e ref,inview,entry

**const {ref,inView,entry}=useInView()**

next create a div at the end of data and Add the above ref=ref and destrcture isfetchingNextPage from useInfiniteQuery

and use it as flag to have loading part

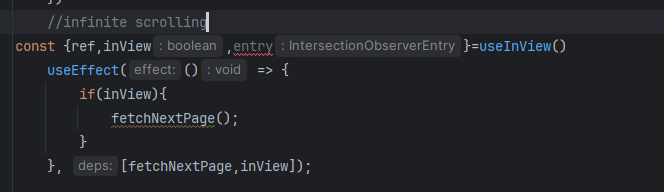




Step 4:

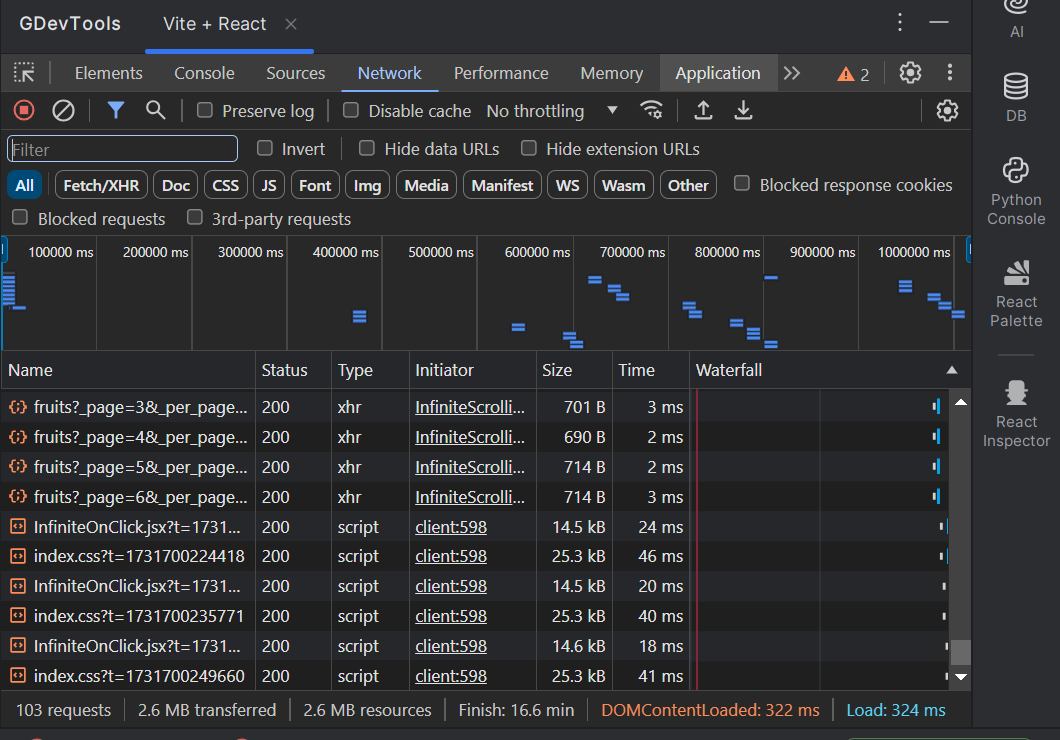
Invoke inView and remember inview returns Boolean,if inview thn new data is fetched

If inview thn fetch next page





Devtools as you can see it fetches step by step





### \*\*Documentation for Implementing Infinite Scrolling\*\*

Infinite scrolling allows you to dynamically fetch and display data as the user scrolls down a page. Below is a step-by-step guide to implement infinite scrolling using `react-intersection-observer` and `react-query`.

---

### \*\*1. Install Required Dependencies\*\*

1. Install `react-intersection-observer` for detecting the viewport entry of elements:

```bash

npm install react-intersection-observer

```

2. Install `react-query` (if not already installed) for data fetching:

```bash

npm install @tanstack/react-query axios

```

---

### \*\*2. Setup the Component\*\*

#### \*\*Import Required Libraries\*\*

At the beginning of your component file, import necessary modules:

```jsx

import React, { useEffect } from 'react';

import axios from "axios";

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

import { useInView } from "react-intersection-observer";

import LoadingAnim from "./LoadingAnim.jsx"; // Loading animation component

```

---

### \*\*3. Create the Data Fetching Function\*\*

Define a function to fetch paginated data from the server:

```jsx

const fetchFruits = async ({ pageParam }) => {

const url = `http://localhost:4000/fruits?\_page=${pageParam}&\_per\_page=5`;

const response = await axios.get(url);

return response.data;

};

```

---

### \*\*4. Use `useInfiniteQuery`\*\*

Set up the infinite query using `useInfiniteQuery` from `react-query`:

```jsx

const { data, isLoading, isError, error, fetchNextPage, hasNextPage, isFetchingNextPage } = useInfiniteQuery({

queryKey: ["fruits"], // Unique query key

queryFn: fetchFruits, // Fetch function

initialPageParam: 1, // Start fetching from page 1

getNextPageParam: (\_lastPage, allPages) => {

// Define when to fetch the next page

return allPages.length < 6 ? allPages.length + 1 : undefined;

},

});

```

---

### \*\*5. Use `useInView`\*\*

Use `react-intersection-observer` to detect when the user scrolls near the bottom of the page:

```jsx

const { ref, inView } = useInView(); // Returns a ref and inView boolean

```

---

### \*\*6. Trigger Fetching When in View\*\*

Use the `useEffect` hook to call `fetchNextPage` whenever the observer detects the bottom element:

```jsx

useEffect(() => {

if (inView) {

fetchNextPage();

}

}, [fetchNextPage, inView]);

```

---

### \*\*7. Render the UI\*\*

1. Handle loading, error, and success states:

2. Map over the fetched data to display it.

3. Use the `ref` returned by `useInView` on the last element to trigger infinite scrolling.

Here’s the complete UI code:

```jsx

return (

<div className="w-screen h-screen flex flex-col justify-center items-center text-xl font-bold bg-gradient-to-r from-cyan-500 to-blue-500 overflow-x-hidden px-4 pb-5">

<h1 className="bg-clip-text text-transparent bg-gradient-to-r from-blue-700 via-sky-900 to-violet-800 text-4xl mt-2 font-[900]">

FRUITS

</h1>

{data?.pages?.map((page) =>

page?.data.map((fruit) => (

<div key={fruit.id} className="mx-2 my-1 p-2 rounded-md bg-white/30 w-[10em] h-[15em] text-center">

{fruit.name}

</div>

))

)}

<div ref={ref}>{isFetchingNextPage && <LoadingAnim />}</div>

</div>

);

```

---

### \*\*8. Key Notes\*\*

1. \*\*Pagination API:\*\* Ensure your backend API supports pagination with query parameters like `\_page` and `\_per\_page`.

2. \*\*Error Handling:\*\* Always include error handling in your component to show meaningful messages if the API fails.

3. \*\*Optimize Performance:\*\* Avoid redundant re-renders by ensuring dependencies in hooks are properly defined.

---

### \*\*Summary\*\*

- \*\*Dependencies:\*\* Install `react-query` and `react-intersection-observer`.

- \*\*Fetch Data:\*\* Use `axios` with a query function for paginated API calls.

- \*\*Infinite Query:\*\* Use `useInfiniteQuery` to handle paginated data fetching.

- \*\*In-View Trigger:\*\* Use `useInView` to detect when the user scrolls near the bottom and fetch the next page.

- \*\*Render Data:\*\* Dynamically render the data and append new content as it loads.

This implementation provides a smooth infinite scrolling experience for any dataset.