

1. React-query là gì ?

React Query (hay TanStack Query) là một thư viện quản lý trạng thái bất đồng bộ trong React, chuyên dùng để lấy dữ liệu từ API, lưu cache, cập nhật và đồng bộ dữ liệu một cách hiệu quả mà không cần phải tự viết nhiều code phức tạp như khi dùng `useEffect` và `useState` truyền thống

- React Query giúp bạn gọi API và quản lý dữ liệu trả về một cách dễ dàng.
- Nó tự động lưu cache dữ liệu, giúp tránh việc gọi API nhiều lần không cần thiết.
- Hỗ trợ quản lý trạng thái tải dữ liệu (`loading`), lỗi (`error`), và cập nhật dữ liệu mới.
- Không thay thế việc gọi API (bạn vẫn dùng axios, fetch,... để gọi API), React Query chỉ quản lý dữ liệu và trạng thái liên quan.
- Cung cấp các hook như `useQuery` để lấy dữ liệu, `useMutation` để tạo, sửa, xóa dữ liệu.

Sử dụng react-query

`useQuery` (Fetching data)

Để sử dụng hook `useQuery`, ta phải truyền ít nhất 2 tham số:

- Tham số đầu tiên là `queryKey`
- Tham số thứ 2 là hàm trả về 1 promise:
 - Resolve data, hoặc
 - Throw error
- Tham số thứ 3 là các options (ta sẽ tìm hiểu ở bên dưới).

`queryKey` được sử dụng để refetching, caching và chia sẻ dữ liệu giữa các component với nhau. (Ta sẽ tìm hiểu thêm với 1 số ví dụ ở bên dưới)

Sau đây là ví dụ sử dụng `useQuery` cơ bản.

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

const fetchTodos = () => fetch('/api/todos').then(res => res.json());

function Todos() {
  const { data, isLoading, isError, error } = useQuery({
    queryKey: ['todos'], // key định danh cho query này
    queryFn: fetchTodos, // hàm gọi API trả về Promise
  });

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error: {error.message}</div>;
  return (
```

```

<ul>
  {data.map(todo => (
    <li key={todo.id}>{todo.title}</li>
  ))}
</ul>
);
}

```

`useQuery` trả về những thông tin cần thiết như `data`, `isLoading` hay `isError`

```

const AllSkillCenters = useQuery<ISkillCenterInfoViewModel[]>({
  queryKey: [ `AllSkillCenters` ],
  queryFn: async () => {
    const response = await baseAxios.get("/SkillCenter/GetAll")
    return response.data.data
  },
  refetchOnWindowFocus: false
})

const AllSelectionBySkillCenterId = useQuery<ISkillCenterInfoViewModel>({
  queryKey: [ `AllProgramsBySkillCenterId` ],
  queryFn: async () => {
    let cols = "Program,Branch";
    const response = await baseAxios.get(`/SkillCenter/Get?id=${form.values.skillCenterId}&cols=${cols}`)
    return response.data.data
  },
  enabled: form.getValues().skillCenterId != null,
  refetchOnWindowFocus: false
})

```

useMutation (Create, Update, Delete)

`useMutation` khác với `useQuery`, được sử dụng để tạo/thay đổi/xóa dữ liệu ở. (ví dụ như đăng ký, đăng nhập).

Để sử dụng hook `useMutation`, ta phải truyền ít nhất 1 tham số:

- Tham số thứ nhất là hàm trả về 1 promise:
 - Resolve data, hoặc
 - Throw error
- Tham số thứ 2 là các options (ta sẽ tìm hiểu sau).

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

```
const postTodo = (newTodo) =>
```

```
fetch('/api/todos', {  
  
  method: 'POST',  
  
  body: JSON.stringify(newTodo),  
  
  headers: { 'Content-Type': 'application/json' },  
  
}).then(res => res.json());
```

```
function AddTodo() {  
  
  const queryClient = useQueryClient();  
  
  const mutation = useMutation(postTodo, {  
  
    onSuccess: () => {  
  
      // Khi thêm thành công, tự động gọi lại query 'todos' để cập nhật dữ liệu mới  
  
      queryClient.invalidateQueries(['todos']);  
  
    },  
  
  });  
  
  return (  
  
    <button  
  
      onClick={() => {  
  
        mutation.mutate({ title: 'New Todo' });  
  
      }}  
  
    >  
  
    Add Todo
```

</button>

);

}

```
const mutateCourse = useMutation({
  Windsurf: Refactor | Explain | Generate JSDoc | X
  mutationFn: async () => {
    let body = {
      id: courseId,
      isEnabled: false
    }
    let response = await baseAxios.post('/Course/Delete', body)
    if (response.data.isSuccess === 1) { ...
    }

    if (response.data.isSuccess !== 1) { ...
    }
  },
  Windsurf: Refactor | Explain | Generate JSDoc | X
  onSuccess: (response) => {
    //todo
    queryClient.invalidateQueries({
      queryKey: ['CourseTable']
    });
  },
});
```

```
Windsurf: Refactor | Explain | Generate JSDoc | X
  onError: () => {
    //todo
  },
});
```

- `mutation.mutate(data)` để thực thi mutation.
- `onSuccess` dùng để xử lý khi mutation thành công, ví dụ: làm mới dữ liệu.

5. Một số trạng thái quan trọng từ `useQuery` và `useMutation`

- `isLoading`: đang tải dữ liệu.
- `isError`: có lỗi khi gọi API.
- `error`: chi tiết lỗi.
- `data`: dữ liệu trả về.
- `isFetching`: đang fetch lại dữ liệu (refetch).

- **mutate**: thực thi mutation.

✅ **queryKey** là gì?

👉 Là “tên định danh duy nhất” cho mỗi query trong React Query.

Nó giống như một “**chìa khóa**” để **tra dữ liệu** trong một hệ thống cache, hoặc để biết dữ liệu nào đã được gọi, đang chờ, hay đã có sẵn.

💡 Ví dụ đơn giản

ts

```
useQuery(['users'], fetchUsers)
```

- **queryKey** là ['users'].
 - React Query sẽ gắn toàn bộ trạng thái và cache liên quan đến query này vào key 'users'.
-

🔍 Tại sao cần **queryKey**?

Vì React Query:

- **Lưu cache dựa trên key**
 - **Tự động theo dõi** trạng thái (**loading**, **error**, v.v.) dựa vào key
 - **Tái sử dụng** dữ liệu nếu key trùng
 - Cho phép **invalidation**, **refetch**, **pagination**, v.v.
-

🧱 Cấu trúc của **queryKey**

queryKey luôn nên là một **mảng** có cấu trúc.

ts

```
['resourceName', optionalParams]
```

Ví dụ:

ts

```
['users']           // danh sách tất cả user
['user', 5]         // user có id = 5
['posts', { tag: 'react' }] // danh sách post theo tag
```

✅ React Query coi `['posts', { tag: 'react' }]` khác hoàn toàn với `['posts']`.

Cách React Query sử dụng **queryKey**

1. Tạo cache

- Mỗi **queryKey** tạo ra một **kho cache riêng biệt**

ts

```
const query = useQuery(['programs'], fetchPrograms)
// cache này sẽ có tên là "programs"
```

2. Xác định trạng thái

- Mỗi **queryKey** có trạng thái riêng:
 - `isLoading`, `isFetching`, `isSuccess`, `isError`
 - Không bị ảnh hưởng bởi query khác
-

3. Tái sử dụng dữ liệu (cache hit)

- Nếu **queryKey** giống nhau và còn **fresh** (theo **staleTime**), sẽ **không gọi API lại**, dùng dữ liệu đã có.

ts

```
useQuery(['products'], fetchProducts) // gọi lần đầu  
useQuery(['products'], fetchProducts) // dùng cache!
```

4. Tùy biến theo điều kiện

Ví dụ lọc theo tham số:

ts

```
useQuery(['products', { category: 'books' }], fetchBooks)  
useQuery(['products', { category: 'clothes' }], fetchClothes)
```

Hai query này **hoàn toàn độc lập** trong mắt React Query.

5. Xóa, làm mới (invalidate) dữ liệu

ts

```
queryClient.invalidateQueries(['products'])
```

→ React Query biết cache nào cần bị refetch.

Sai lầm phổ biến khi dùng **queryKey**

Sai lầm

Dùng object không ổn định làm key

Tác hại

Query bị gọi lại liên tục do key thay đổi mỗi render

Dùng key trùng cho hai mục khác nhau

Dữ liệu bị đè lên nhau, status sai, cache lỗi

Không đặt key theo cấu trúc

Không tận dụng được invalidation theo nhóm

Mẹo đặt **queryKey** đúng cách

Tình huống	queryKey đề xuất
Danh sách users	<code>['users']</code>
Thông tin user id = 7	<code>['user', 7]</code>
Danh sách bài viết theo tag	<code>['posts', { tag: 'react' }]</code>
Danh sách có pagination	<code>['posts', { page: 1, size: 10 }]</code>

Luôn đặt **queryKey** theo **dữ liệu bạn muốn nhận được**, không phải theo logic xử lý.

1. **staleTime** là gì?

Định nghĩa:

staleTime là khoảng thời gian (ms) mà **dữ liệu được coi là “tươi mới” (fresh)** sau khi được fetch.

- Trong thời gian **staleTime**, React Query sẽ **không tự động refetch lại** dữ liệu (ngay cả khi bạn focus lại tab, hay remount component).
- Sau khi hết **staleTime**, dữ liệu trở thành **stale** → **React Query sẽ tự động refetch khi:**
 - Tab focus lại
 - Component mount lại

- Gọi `refetch()` hoặc `invalidate`

Mặc định:

ts
`staleTime: 0`

Nghĩa là:

- Ngay sau khi fetch xong, dữ liệu đã stale.
- Do đó, **mọi lần focus tab/mount lại sẽ luôn refetch.**

Ví dụ:

ts
`useQuery({
 queryKey: ['products'],
 queryFn: fetchProducts,
 staleTime: 1000 * 60 * 5, // 5 phút
})`

- Dữ liệu được coi là **tươi** trong 5 phút → **không refetch tự động** trong thời gian này.

2. Các trạng thái (`query.status` hoặc flags như `isLoading`, `isFetching`, ...)

React Query cung cấp **nhiều trạng thái tiện lợi** để bạn xử lý logic hiển thị loading, error, success,...

Các status chính:

Trạng thái (`status`)

Khi nào xảy ra

`'loading'`

Khi query đang fetch lần đầu tiên (cache chưa có)

'success'	Khi query fetch thành công
'error'	Khi query bị lỗi

✓ Các flags chi tiết hơn:

Flag	Ý nghĩa chi tiết
isLoading	true nếu query chưa từng fetch thành công , và đang được gọi lần đầu (lúc đó cũng isFetching = true)
isFetching	true nếu query đang fetch lại ở bất kỳ thời điểm nào (bao gồm lần đầu và refetch)
isSuccess	true nếu đã fetch xong và thành công
isError	true nếu query có lỗi
isIdle	true nếu query chưa từng được kích hoạt (dùng enabled: false)
isStale	true nếu dữ liệu đã cũ (sau staleTime) → React Query sẽ có thể refetch trong những tình huống như focus lại tab hoặc remount component
isFetched	true nếu query đã từng được fetch ít nhất một lần (thành công hoặc lỗi)
isRefetching	true nếu query đang được refetch sau lần fetch đầu tiên

↺ Mối liên hệ giữa **staleTime** và các trạng thái

Hành động	isLoading	isFetching	isStale
Lần đầu gọi query (chưa có cache)	✓	✓	? (false cho tới khi fetch xong)
Fetch thành công xong	✗	✗	✗ (dữ liệu còn fresh)
Hết staleTime	✗	✗	✓ (có thể refetch)
Focus lại tab sau staleTime	✗ (nếu có cache)	✓	✓ (cho đến khi fetch xong)
Gọi refetch()	✗ hoặc ✓	✓	✓

Ví dụ minh họa:

ts

```
const query = useQuery({
  queryKey: ['posts'],
  queryFn: fetchPosts,
  staleTime: 10000, // 10 giây
})
```

Sau khi gọi xong **query.isSuccess = true** **query.isStale = false**

Sau 10 giây, **query.isStale = true**, và nếu bạn:

- Focus lại tab: React Query sẽ refetch tự động.
- Remount component: cũng sẽ refetch

1. **isLoading**

- **true** khi: query **chưa có dữ liệu trong cache** và đang thực hiện lần fetch đầu tiên.

ts

```
const query = useQuery(['posts'], fetchPosts)
```

Khi bắt đầu fetch **isLoading = true**

Khi fetch xong **isLoading = false**

•

Sau đó, nếu component mount lại (và có cache), thì **isLoading = false** dù có thể **isFetching = true**.

Cảnh báo:

- Không nhầm **isLoading** với **isFetching**. **isLoading** chỉ **lần đầu gọi**, còn **isFetching** có thể xảy ra nhiều lần.

2. **isFetching**

- **true khi:** query đang gọi API (fetching), dù là lần đầu hay refetch.

Tình huống	isFetching
Lần đầu fetch	✓
Refetch do <code>refetch()</code>	✓
Refetch do focus tab	✓
Khi fetch xong	✗

📌 Gợi ý sử dụng:

- Dùng để hiện spinner nhỏ (loading overlay), không nên dùng để ẩn toàn bộ UI.

3. isSuccess

- **true khi:** fetch thành công (status = 'success').

Trạng thái	isSuccess
Chưa gọi	✗
Đang gọi	✗
Gọi xong	✓ nếu không lỗi

- Vẫn giữ là **true** ngay cả sau khi stale, trừ khi bị `refetch()` mà gặp lỗi → lúc đó chuyển sang **isError**.

✗ 4. isError

- **true khi:** `queryFn` ném lỗi hoặc trả về rejected Promise.

```
useQuery(['user', id], () => fetchUser(id))
```

| Nếu `fetchUser(id)` bị lỗi | `isError = true`, `error` chứa thông tin lỗi |

5. `isStale`

- **true** khi: dữ liệu đã hết hạn **`staleTime`**.
- Dù `isSuccess = true`, nhưng nếu `isStale = true` → có thể bị refetch khi:
 - Tab focus
 - Component remount
 - Manual refetch
 - Query bị invalidate

| Sau khi fetch | `isStale = false` |

| Sau `staleTime` | `isStale = true` |

6. `isRefetching`

- **true** khi: query đã từng fetch thành công và đang refetch lại.
- Đây là `isFetching` kết hợp với `isSuccess = true`.

ts

```
useQuery(['products'], fetchProducts)
```

| Lần đầu fetch | `isRefetching = false` |

| Gọi `refetch()` sau đó | `isRefetching = true` |

7. `isIdle`

- **true khi:** query chưa được gọi, thường xảy ra khi **enabled: false**.

```
ts
const result = useQuery({
  queryKey: ['posts'],
  queryFn: fetchPosts,
  enabled: false
})
```

result.isIdle // true

| Khi enabled = false | **isIdle = true** |
 | Khi chạy lần đầu | **isIdle = false** |

✅ 8. **isFetched**

- **true khi:** query **đã từng được gọi ít nhất một lần**, bất kể thành công hay thất bại.

Trạng thái	isFetched
Trước khi fetch	❌
Sau khi fetch xong	✅
Sau lỗi cũng	✅

↔ **So sánh isFetched vs isSuccess:**

Tình huống	isFetched	isSuccess
Fetch thành công	✅	✅
Fetch bị lỗi	✅	❌
Chưa từng fetch	❌	❌

📊 Tổng hợp các flag theo timeline

Giai đoạn	isIdle	isLoading	isFetching	isSuccess	isError	isFetched	isStale	isRefetching
-----------	---------------	------------------	-------------------	------------------	----------------	------------------	----------------	---------------------

Trước khi fetch	✓	✗	✗	✗	✗	✗	✗	✗
Đang fetch lần đầu	✗	✓	✓	✗	✗	✗	✗	✗
Fetch thành công	✗	✗	✗	✓	✗	✓	✗	✗
Sau staleTime	✗	✗	✗	✓	✗	✓	✓	✗
Refetch sau stale	✗	✗	✓	✓	✗	✓	✓	✓
Refetch bị lỗi	✗	✗	✗	✗	✓	✓	✓	✗

🧱 NHỮNG THÀNH PHẦN CHÍNH CỦA REACT QUERY

1. 🧩 QueryClient

♦ Là "trung tâm điều phối" dữ liệu

- Lưu trữ **cache**, **trạng thái** của **query**, **mutation**, v.v.
- Bạn chỉ cần tạo một **QueryClient** duy nhất trong ứng dụng và dùng **QueryClientProvider** để chia sẻ nó.

✓ Cách dùng:

```
js
import { QueryClient } from '@tanstack/react-query';

const queryClient = new QueryClient();
```

2. 🧩 QueryClientProvider

♦ Là thành phần bọc quanh toàn bộ ứng dụng

- Giúp mọi component con có thể dùng `useQuery`, `useMutation`, ...

✅ Cách dùng:

```
jsx
import { QueryClientProvider } from '@tanstack/react-query';

<QueryClientProvider client={queryClient}>
  <App />
</QueryClientProvider>
```

3. 🔍 `useQuery` – GET data

♦ Dùng để lấy dữ liệu từ API hoặc nguồn bất đồng bộ.

- React Query sẽ quản lý `loading`, `error`, `data`, `refetch`, `cache`, v.v.

✅ Ví dụ:

```
jsx
const { data, isLoading, error } = useQuery({
  queryKey: ['users'],
  queryFn: fetchUsers,
});
```

Field	Ý nghĩa
<code>queryKey</code>	Định danh cache
<code>queryFn</code>	Hàm bất đồng bộ (thường là gọi API)
<code>data</code>	Kết quả trả về
<code>isLoading</code>	Đang fetch
<code>error</code>	Có lỗi

4. 🔄 `useMutation` – POST / PUT / DELETE

♦ Dùng cho các tác vụ thay đổi dữ liệu

- Không tự cache kết quả, nhưng có thể trigger `invalidateQueries` để cập nhật query liên quan.

✓ Ví dụ:

jsx

```
const mutation = useMutation({
  mutationFn: addUser,
  onSuccess: () => {
    queryClient.invalidateQueries(['users']); // Refetch lại dữ liệu
  },
});
```

Field	Ý nghĩa
<code>mutationFn</code>	Hàm bất đồng bộ (POST, PUT, DELETE)
<code>mutate(data)</code>	Thực thi mutation
<code>isPending, isSuccess, isError</code>	Trạng thái của mutation

5. 🧠 `queryKey` – định danh duy nhất cho mỗi query

- Là một mảng (thường là `[resource, id]`)
- Giúp React Query biết **dữ liệu nào cần refetch/cache riêng**

js

```
queryKey: ['user', 5] // Dữ liệu người dùng ID = 5
```

6. 🧰 `QueryClient` methods

Sử dụng `useQueryClient()` để truy cập `queryClient` trong component:

js

```
const queryClient = useQueryClient();
```

Một số method hay dùng:

Method	Công dụng
--------	-----------

<code>invalidateQueries(['users'])</code>	Xóa cache và tự refetch
<code>setQueryData(['users'], data)</code>	Cập nhật cache thủ công
<code>getQueryData(['users'])</code>	Lấy data từ cache
<code>refetchQueries(['users'])</code>	Gọi lại API

1. Hooks phụ trợ (Advanced hooks)

`useInfiniteQuery`

- Dùng để lấy dữ liệu theo từng “trang” một cách **cuộn vô hạn** (infinite scroll).

```
jsx
const {
  data,
  fetchNextPage,
  hasNextPage,
  isFetchingNextPage,
} = useInfiniteQuery({
  queryKey: ['posts'],
  queryFn: fetchPage,
  getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
});
```

`useIsFetching`, `useIsMutating`

- Trả về số lượng query hoặc mutation đang ở **trạng thái loading**.

```
js
const isFetching = useIsFetching(); // Số lượng query đang load
const isMutating = useIsMutating(); // Số mutation đang thực hiện
```

👉 Hữu ích để hiển thị spinner loading toàn cục.

`useQueryClient`

- Truy cập đối tượng `queryClient` để:
 - invalidate query
 - cập nhật cache
 - lấy dữ liệu cache
 - refetch thủ công

js

```
const queryClient = useQueryClient();  
queryClient.invalidateQueries(['users']);
```

⚙️ 2. Options mở rộng trong `useQuery` / `useMutation`

🟡 **enabled: false**

- Dùng để **tạm thời không gọi API** cho đến khi điều kiện sẵn sàng.

js

```
useQuery({  
  queryKey: ['user', id],  
  queryFn: () => fetchUser(id),  
  enabled: !!id, // chỉ gọi khi có id  
});
```

🟡 **select**

- Dùng để **biến đổi dữ liệu** sau khi fetch, trước khi render.

js

```
useQuery({  
  queryKey: ['user'],  
  queryFn: fetchUser,  
  select: (data) => data.name.toUpperCase(), // chỉ lấy tên  
});
```

🟡 **initialData**

- Dùng để cung cấp dữ liệu mặc định trước khi fetch hoàn tất (prefetch từ SSR hoặc LocalStorage chẳng hạn).

```
js
useQuery({
  queryKey: ['user'],
  queryFn: fetchUser,
  initialData: { name: 'Ngọc' },
});
```

🟡 **staleTime và cacheTime**

- **staleTime**: thời gian **không cần refetch lại** (coi là “mới”)
- **cacheTime**: thời gian **giữ cache sau khi unmount**

```
js
useQuery({
  queryKey: ['data'],
  queryFn: fetchData,
  staleTime: 5 * 60 * 1000, // 5 phút
  cacheTime: 10 * 60 * 1000 // 10 phút
});
```

🧰 **3. Prefetch & Hydration**

🟡 **queryClient.prefetchQuery()**

- Dùng để lấy sẵn dữ liệu trước khi cần, ví dụ trong SSR hoặc trước khi chuyển route.

```
js
queryClient.prefetchQuery({
  queryKey: ['user', 1],
  queryFn: () => fetchUser(1),
});
```

});

🟡 dehydrate & Hydrate

- Dùng trong SSR (Next.js...) để **chuyển cache từ server về client**.

🔧 4. Devtools nâng cao

🟡 ReactQueryDevtools

- Giao diện kiểm tra trạng thái query, mutation, cache trực tiếp trên trình duyệt.

🔧 5. Error Boundary hỗ trợ

- React Query **không tự throw lỗi** – bạn có thể dùng React's Error Boundary kết hợp với `useErrorBoundary`.

✅ Tổng hợp bảng nhanh

Thành phần phụ	Mô tả	Dùng khi
<code>useInfiniteQuery</code>	Cuộn vô hạn	Pagination động
<code>useIsFetching</code>	Số query đang loading	Hiển thị loading global
<code>enabled</code>	Dừng fetch tạm thời	Chưa có ID, chưa đăng nhập
<code>select</code>	Biến đổi data trước khi render	Lấy riêng 1 trường
<code>initialData</code>	Dữ liệu ban đầu	SSR, localStorage
<code>staleTime</code>	Thời gian "tươi"	Giảm số lần refetch
<code>prefetchQuery</code>	Lấy trước dữ liệu	Trước khi chuyển trang
<code>queryClient</code> methods	Quản lý cache thủ công	Force refetch, update thủ công

⚡ 1. Query Observer (custom logic)

- Cho phép bạn **quan sát nhiều query** cùng lúc, ngoài hook `useQuery`.

Ví dụ:

```
js
import { QueryObserver } from '@tanstack/react-query';

const observer = new QueryObserver(queryClient, {
  queryKey: ['posts'],
  queryFn: fetchPosts,
});

const unsubscribe = observer.subscribe(result => {
  console.log(result.data);
});
```

✅ Dùng trong **lib logic**, không phụ thuộc React.

📦 2. `useQueries()` – chạy nhiều query song song

Nếu bạn cần gọi **nhiều API độc lập** cùng lúc:

```
jsx
const results = useQueries({
  queries: [
    { queryKey: ['user', 1], queryFn: () => fetchUser(1) },
    { queryKey: ['posts'], queryFn: fetchPosts },
  ],
});
```

Mỗi `result` giống như một `useQuery`, dùng `results[0].data`, v.v.

📖 3. Paginated Queries (classic)

Ngoài `useInfiniteQuery`, bạn có thể dùng `useQuery` với pagination thủ công:

```
jsx
const [page, setPage] = useState(1);

const { data } = useQuery({
  queryKey: ['posts', page],
  queryFn: () => fetchPosts(page),
});
```

Có thể kết hợp với `keepPreviousData: true` để giữ cache page cũ.

4. Polling (refetchInterval)

Tự động refetch theo thời gian:

```
js
useQuery({
  queryKey: ['notifications'],
  queryFn: fetchNoti,
  refetchInterval: 5000, // mỗi 5 giây
});
```

5. Retry logic

React Query có sẵn retry khi API lỗi:

```
js
useQuery({
  queryKey: ['data'],
  queryFn: fetchData,
  retry: 3, // Thử lại 3 lần
  retryDelay: attempt => attempt * 1000, // delay tăng dần
});
```

6. Garbage Collection (cacheTime)

Dữ liệu được cache nhưng sẽ bị **xóa tự động sau 5 phút (mặc định)** nếu không còn ai dùng
→ Bạn có thể điều chỉnh bằng `cacheTime`.

7. Placeholder Data (hiện UI nhanh)

Hiển thị dữ liệu giả lập ngay lập tức, thay vì **loading**:

```
js
useQuery({
  queryKey: ['user', id],
  queryFn: fetchUser,
  placeholderData: { name: 'Đang tải...' },
});
```

8. Global Query Config (tùy biến mặc định)

Khi khởi tạo **QueryClient**, bạn có thể config mặc định cho toàn bộ **useQuery**, **useMutation**,...

```
js
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retry: 2,
      staleTime: 1000 * 60, // 1 phút
    },
  },
});
```

9. Query Cancellation

React Query tự hủy promise khi component bị unmount để tránh memory leak – bạn có thể tùy chỉnh nếu cần (ít gặp).

10. DevTools nâng cao

```
bash
npm install @tanstack/react-query-devtools
```

```
jsx
import { ReactQueryDevtools } from '@tanstack/react-query-devtools';
```

✓ **Tổng Kết Tính Năng "Ẩn" Hữu Ích:**

Tính năng	Giải thích	Dùng khi nào
<code>useQueries()</code>	Gọi nhiều API song song	Hiển thị dashboard tổng hợp
<code>refetchInterval</code>	Polling định kỳ	Notification, realtime
<code>retry, retryDelay</code>	Tự động thử lại khi lỗi	API không ổn định
<code>keepPreviousData</code>	Giữ dữ liệu trang cũ khi đổi page	Pagination mượt
<code>placeholderData</code>	Giảm flicker UI	Trải nghiệm tốt hơn
<code>defaultOptions</code>	Cấu hình toàn cục	Dự án lớn
<code>queryObserver</code>	Lập trình nâng cao	Custom store, lib riêng
<code>Hydrate, dehydrate</code>	SSR	Với Next.js
<code>queryClient.setQueryData()</code>	Update cache thủ công	Khi mutation thành công