Why Use React Query?

1. No more manual useEffect + useState soup

Without React Query:

```
useEffect(() => {
  fetch(...).then(...).catch(...)
}, [])
```

You have to set up:

- `isLoading`
- `error`
- `data`
- Re-fetch logic
- Caching logic

React Query abstracts all of that into a clean hook interface:

```
const { data, isLoading, isError } = useQuery({ queryKey, queryFn })
```

2. Automatic Caching

If you re-mount the component or go to another tab and come back:

- React Query uses the cached data immediately
- Then revalidates in the background

This makes apps feel fast and fresh without extra coding.

No Need to Re-implement Polling, Retry, Pagination

It has built-in support for:

- Automatic retries
- Background refetching

- Polling intervals
- Pagination/infinite scroll
- Mutation + optimistic updates. In React Query, a mutation is any operation that changes data on the server — usually a:
  - `POST` (create)
  - `PUT` or `PATCH` (update)
  - `DELETE` (remove)

4. Consistent Loading/Error States

Manual logic often results in bugs like:

- Data flickering on refetch
- Stale data showing during transitions React Query handles these gracefully.

5. Devtools Support

There's a built-in **React Query Devtools** for inspecting queries, cache, retries, etc. Great for debugging.

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

6. It Scales

In a large app with many API calls:

- You avoid spaghetti code
- Queries are organized and consistent
- It's easier to reason about side effects

7. Works Great with SSR

If you're using Next.js or Remix, React Query integrates beautifully with server-side rendering and hydration.

Can Use useEffect + fetch and *Don't* Need React Query If:

- Your app only makes 1–2 API calls
- You don’t need to cache
- You don’t need to poll or do retries
- You can handle everything manually

Project Setup Instructions

1. Install Dependencies

In the root of your project, run:

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

If you're using Vite:

```
npm install
npm run dev
```

This is a user directory app that:

Fetches users from a public API
Loads more users (pagination)
Adds a new user via a form
Uses optimistic updates — new users appear instantly
Uses `@tanstack/react-query` for all data management

Code Overview

`main.tsx`

- Initializes and configures the React Query `QueryClient`
- Wraps the app in `QueryClientProvider`
- Adds React Query Devtools for debugging

`App.tsx`

- Renders both the user list and the add-user form

`UserList.tsx`

- Uses `useInfiniteQuery` to:
  - Fetch users page-by-page
  - Display “Load More” button for pagination
- Shows loading and error states

`AddUserForm.tsx`

- Uses `useMutation` to POST a new user
- **Optimistically adds** the user to the UI before the server responds
- Rolls back the optimistic change if there's an error
- Avoids cache invalidation to prevent overwriting the UI too soon

More Detail - See Comments in Code:

`main.tsx` — Application Entry Point

Sets up the app’s React Query client and renders the app.

- Creates a `QueryClient` instance with default settings:
  - `retry`: Will retry failed queries up to 3 times.
  - `refetchOnWindowFocus`: Automatically refetches data when window regains focus.
  - `refetchInterval` is commented out (was used for polling every 10s).
- Wraps your app in `<QueryClientProvider>` so all components can use React Query.
- Includes `<ReactQueryDevtools />` for debugging query states in development.

`App.tsx`

Renders main application layout.

- Displays the app title: “User Directory”.
- Includes two child components:
  - `<AddUserForm />` – lets users add new entries.
  - `<UserList />` – shows a list of users with pagination.

`UserList.tsx` — Paginated Data Display

Fetches and displays a paginated list of users using `useInfiniteQuery`.

- Calls the public API `https://jsonplaceholder.typicode.com/users` with pagination (`?_limit=3&_page=X`).
- Uses `useInfiniteQuery` to:
  - Load 3 users at a time.
  - Track how many pages have been loaded.
- `fetchNextPage()` loads the next page of users when the button is clicked.
- Shows loading and error messages.
- Renders each page of users inside a separate `<ul>`.

`AddUserForm.tsx` — Form + Optimistic Mutation

Adds a new user with an optimistic UI update.

- `useState()` is used to track the form input values.
- `useMutation()` is used to **send a POST request** to add a user.
- `onMutate` performs an **optimistic update**:
  - Cancels any running user queries.
  - Gets current cached users.

    - Immediately adds a new user to the beginning of the first page in cache.
    - Stores the previous data for rollback.
- `onError` restores the previous data if something fails.
- `onSettled` does **not** invalidate the cache (you commented it out) so the new user doesn't flash and disappear.