React Query (TanStack Query)

Server state made easy: useQuery for fetching with cache, loading and error built in, useMutation for sending, and why server state is not store state.

Introduction

Every fetch in this course hand-built the same machine: data/loading/error state, an effect, a cleanup worry. **React Query** (TanStack Query) is that machine, perfected and shared: it fetches, **caches**, refetches when data goes stale, and hands your component `data`, `isLoading` and `error` as finished products. Its key insight starts the lesson: **server state is not store state** — it belongs to the server, you merely borrow a copy. *(Roman Urdu: Server ka data tumhara store nahi — tum sirf aik copy udhaar lete ho, aur React Query us copy ki dekh bhal karta hai.)*

Key points

- **Two kinds of state, two kinds of tools:** *client state* (theme, cart, form text) is yours — Zustand/Redux/context own it. *Server state* (user lists, posts, weather) is the server's — it can be stale, shared, and changed by others. React Query manages the borrowed copy.
- **`useQuery` is the fetch hook:** `useQuery({ queryKey: ["users"], queryFn: fetchUsers })` returns `{ data, isLoading, error }` — the state trio from 5.19, with no effect and no manual states.
- **The query key is the cache address:** `["users"]`, `["user", 5]`, `["posts", { page: 2 }]` — React Query files each result under its key; any component asking with the same key reads the same cached copy instantly.

  (Roman Urdu: Query key cache ka pata hai — wohi pata, wohi data, dobara fetch nahi.)
- **Caching + staleness, tuned for you:** revisiting a screen shows cached data *immediately* and quietly refetches in the background if it is stale — the app feels instant and stays honest.
- **`useMutation` is the send hook:** for POST/PUT/DELETE it wraps your send function with `mutate`, `isPending` and error handling — and on success you **invalidate** the related query keys so lists refresh themselves: `queryClient.invalidateQueries({ queryKey: ["posts"] })`.
- **Setup is one wrapper:** a `QueryClient` created once, and `<QueryClientProvider client={queryClient}>` around the app — the only ceremony in the library.
- **What you stop writing:** loading flags, error state plumbing, "did someone already fetch this?" guards, and manual refetch-after-save code — the most-repeated code in data apps, deleted.

Code examples

Example 1 — useQuery replaces the whole fetch machine

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

async function fetchUsers() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  if (!res.ok) throw new Error("Failed to load users");
  return res.json();
}

function UserList() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["users"],
    queryFn: fetchUsers,
  });

  if (isLoading) return <p>Loading users…</p>;
  if (error) return <p className="error">{error.message}</p>;

  return (
    <ul>
      {data.map((u) => (
        <li key={u.id}>{u.name} — {u.email}</li>
      ))}
    </ul>
  );
}
```

What it does: Compare this with lesson 5.19's Example 1 — the same screen with the entire effect-and-states apparatus gone. The query function keeps the honest `res.ok` gate; React Query owns when it runs, what it caches, and which of the three states you are in.

What you will see: the loading line, then the user list. Now unmount and remount the component (switch tabs in the app): the list appears **instantly** from cache while a background refetch quietly verifies it.

Example 2 — Setup: the QueryClient and its Provider

```
// main.jsx
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";

const queryClient = new QueryClient();

root.render(
  <QueryClientProvider client={queryClient}>
    <App />
  </QueryClientProvider>
);
```

What it does: One client object holds the whole cache and its settings (how long data stays fresh, retry counts). The Provider makes it available to every `useQuery` below — five lines once per app, and the fetching layer exists.

What you will see: nothing visual — but with the TanStack DevTools added, a panel showing every query key, its cached data and its fresh/stale status: the borrowed copies, all visible.

Example 3 — useMutation: send, then invalidate

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

async function createPost(post) {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(post),
  });
  if (!res.ok) throw new Error("Could not save");
  return res.json();
}

function NewPostForm() {
  const queryClient = useQueryClient();
  const mutation = useMutation({
    mutationFn: createPost,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ["posts"] });
    },
  });

  function handleSubmit(e) {
    e.preventDefault();
    mutation.mutate({ title: "New post", body: "From React Query", userId: 1 });
  }

  return (
    <form onSubmit={handleSubmit}>
      <button disabled={mutation.isPending}>
        {mutation.isPending ? "Saving…" : "Add post"}
      </button>
      {mutation.isError && <p className="error">Saving failed — try again.</p>}
    </form>
  );
}
```

What it does: The mutation wraps the POST with its own pending/error states. On success it **invalidates** the `["posts"]` key — React Query marks that borrowed copy stale and refetches it, so every list of posts on screen refreshes itself. This replaces 5.19's manual "update state after save" bookkeeping.

What you will see: click Add — "Saving…" — then the posts list (elsewhere on the page, using `useQuery(["posts"])`) gains the new entry on its own, with no code connecting the two components.

Quick recap

- **Server state is borrowed, not owned** — React Query manages the copy: fetching, **caching by query key**, background refetching when stale.
- `useQuery` returns `{ data, isLoading, error }` and deletes the hand-built fetch machine; `useMutation` sends changes and **invalidates** keys so lists heal themselves.
- Client state still belongs to stores (Zustand/Redux); React Query is not a store replacement — it is the server-state specialist.

Common mistakes

1. **Copying query data into useState or a store** — the moment you duplicate the borrowed copy, you own two sources of truth that drift apart. The fix: read from the query where you need it; the cache *is* the state for server data.
2. **Unstable or missing query keys** — two different data needs sharing key `["data"]` collide in one cache slot and show each other's results. The fix: keys describe the data precisely — resource + identity + parameters.
3. **Forgetting invalidation after a mutation** — the save succeeds but the list shows yesterday's copy until a manual reload. The fix: `onSuccess` invalidates every key the change touches — the pair is part of the mutation, not an afterthought.

Practice

1. Rebuild lesson 5.19's UserList on React Query (it is Example 1 — now make it yours): add a second component, `UserCount`, that also asks for `["users"]`, and prove with a network log that only ONE fetch happens for both components.
2. Take the PostBox from 5.19 and rebuild it fully: `useQuery(["posts"])` for the list (limit the practice API to a page of posts), `useMutation` for adding, invalidation on success. Compare line counts with the hand-built version and write the difference down — that gap is the library's argument.

Practice quiz

1. What is server state, and who manages its borrowed copy?

- Theme and cart text, managed by Zustand
- Data owned by the server that can be stale, shared and changed by others; React Query manages the borrowed copy, while client state belongs to stores
- Any state, managed by useState
- Only weather data

Answer: Data owned by the server that can be stale, shared and changed by others; React Query manages the borrowed copy, while client state belongs to stores

2. What does useQuery({ queryKey: ["users"], queryFn: fetchUsers }) return, and what is the query key?

- Only the data, and the key is a password
- { data, isLoading, error } with no manual effect or states, and the key is the cache address: any component asking with the same key reads the same cached copy instantly
- A promise only
- Only an error object

Answer: { data, isLoading, error } with no manual effect or states, and the key is the cache address: any component asking with the same key reads the same cached copy instantly

3. After a successful mutation, how do lists refresh themselves in React Query?

- By reloading the whole page
- By invalidating the related query keys on success, for example queryClient.invalidateQueries({ queryKey: ["posts"] })
- They do not, lists stay stale until a manual reload
- By copying the data into useState

Answer: By invalidating the related query keys on success, for example queryClient.invalidateQueries({ queryKey: ["posts"] })

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-react-query · More free tools and notes at https://scopetreasure.com