TanStack Query

Version 5.105 · verified 2026-11-02

TanStack Query v5 for React — keys, freshness, mutations, optimistic updates, infinite lists, retries and Next.js hydration.

Contents

Mental model 2

Setup and query keys 2

useQuery states 4

staleTime vs gcTime 6

Mutations and invalidation 8

Optimistic updates 9

Pagination and infinite queries 11

Errors and retries 13

SSR and Server Components 15

Further reading 17

Mental model

Server state is a cache of data you do not own, not client state you set. TanStack Query keeps one cache entry per query key and decides when that entry is fresh, when to refetch it in the background, and when to garbage-collect it; you decide the keys and what to invalidate. Almost every bug is a key that does not match, or a `staleTime` that does not say what you meant.

Setup and query keys

API	Role
`new QueryClient()`	The cache; create once per app
`QueryClientProvider`	Puts the client in context
`queryKey: ["todos", id]`	Cache identity, array
`queryOptions({ … })`	Typed, reusable key + fn
`ReactQueryDevtools`	Cache inspector, dev builds only

A key is an array hashed deterministically: object property order does not matter, array item order does. Everything the `queryFn` reads must be in the key, or two different requests share one cache entry. Build keys in one factory so components cannot drift.

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

const get = (u: string) =>
  fetch(u).then((r) => r.json());

export const todoKeys = {
  all: ["todos"] as const,
  list: (done: boolean) =>
    queryOptions({
      queryKey: [...todoKeys.all, { done }],
      queryFn: () => get(`/api/todos?done=${done}
`),
```

```
    }),
};
```

> **Gotcha:** A `queryFn` that reads a variable missing from the key serves stale data for the wrong input. Put every input in the key.

useQuery states

Flag	True when
`isPending`	No data yet (`status: "pending"`)
`isError`	Last fetch failed after retries
`isSuccess`	`data` is defined
`isFetching`	Any request in flight, incl. background
`isLoading`	`isPending && isFetching`

`status` describes the data; `fetchStatus` (`"fetching"`, `"paused"`, `"idle"`) describes the request. A query can be `success` and `fetching` at once: that is a background refresh, and you keep showing the old data.

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

type Todo = { id: number; title: string };
function Todos() {
  const q = useQuery({
    queryKey: ["todos"],
    queryFn: (): Promise<Todo[]> =>
      fetch("/api/todos").then((r) => r.json()),
  });
  if (q.isPending) return <p>Loading…</p>;
  if (q.isError) return <p>{q.error.message}</p>;
  return <p>{q.data.length} todos</p>;
}
```

> **Gotcha:** `isLoading` is false for a disabled query (`enabled: false`) that has no data, because nothing is fetching. Gate the spinner on `isPending` unless you mean “first fetch now”.

staleTime vs gcTime

Option	Default	Controls
`staleTime`	`0`	How long data counts as fresh
`gcTime`	5 min	How long an unused entry stays cached
`refetchOnWindowFocus`	`true`	Refetch stale data on focus
`refetchOnMount`	`true`	Refetch stale data on mount
`refetchInterval`	off	Polling, ignores `staleTime`

With the default `staleTime: 0`, every mount, window focus and network reconnect refetches. `staleTime` is the knob to turn: raise it to how long the data is genuinely good for. `gcTime` only matters once no component uses the query.

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

export const client = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 60_000,       // fresh for 1 min
      gcTime: 10 * 60_000,     // keep unused 10
min
    },
  },
});
```

> **Tip:** `staleTime: Infinity` still allows `invalidateQueries`; `staleTime: "static"` blocks even that. Use `"static"` only for data that cannot change while the app runs.

Mutations and invalidation

API	Does
`useMutation({ mutationFn })`	Wraps a write
`mutate(vars)` / `mutateAsync`	Fire it / await it
`invalidateQueries({ queryKey })`	Mark stale, refetch active
`setQueryData(key, data)`	Write the cache directly

`invalidateQueries` matches by key prefix, so `["todos"]` invalidates every list and detail under it. Active queries refetch at once; inactive ones refetch next time they mount. Return the promise from `onSuccess` to keep the mutation pending until the refetch finishes.

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

function useAddTodo() {
  const qc = useQueryClient();
```

```
  return useMutation({
    mutationFn: (t: string) => fetch("/api/todos",
      { method: "POST", body: t }),
    onSuccess: () =>
      qc.invalidateQueries({ queryKey:
["todos"] }),
  });
}
```

> **Gotcha:** Mutations do not retry by default (queries retry 3 times), and nothing refetches after a mutation unless you invalidate or write the cache yourself.

Optimistic updates

Callback	Arguments (v5)
`onMutate`	`(variables, context)`
`onError`	`(error, variables, onMutateResult, context)`

Callback	Arguments (v5)
`onSettled`	`(data, error, variables, onMutateResult, context)`

Two styles: render `variables` from the pending mutation in one place (no rollback needed), or write the cache in `onMutate` and roll back on error. `context.client` is the `QueryClient`; whatever `onMutate` returns arrives as `onMutateResult`.

```
import { useMutation } from "@tanstack/react-query";
const key = ["todos"];
export const useSave = () => useMutation({
  mutationFn: (t: string) => fetch(`/api?t=${t}`),
  onMutate: async (t, { client }) => {
    await client.cancelQueries({ queryKey: key });
    const prev = client.getQueryData<string[]>(key);
    client.setQueryData(key, [...(prev ?? []),
```

```
t]);
    return { prev }; // becomes onMutateResult
  },
  onError: (_e, _t, res, { client }) =>
    client.setQueryData(key, res?.prev),
  onSettled: (_d, _e, _t, _r, { client }) =>
    client.invalidateQueries({ queryKey: key }),
});
```

> **Gotcha:** Skip `cancelQueries` and an in-flight refetch lands after your optimistic write and silently overwrites it.

Pagination and infinite queries

Option	Use
`placeholderData: keepPreviousData`	Keep old page while next loads
`initialPageParam`	Required first cursor

Option	Use
`getNextPageParam`	Next cursor, or `undefined` to stop
`fetchNextPage()`, `hasNextPage`	Load more, more exists
`maxPages`	Cap pages kept in memory

Paged lists use `useQuery` with the page in the key plus `keepPreviousData`, so the UI does not flash a spinner between pages. Endless lists use `useInfiniteQuery`, which stores `data.pages` and refetches every loaded page in order when it goes stale.

```
import {
  useInfiniteQuery,
} from "@tanstack/react-query";
type Page = { items: string[]; next?: number };
const get = (c: number): Promise<Page> =>
  fetch(`/api/feed?c=${c}`).then((r) => r.json());
```

```
function Feed() {
  const q = useInfiniteQuery({
    queryKey: ["feed"], initialPageParam: 0,
    queryFn: ({ pageParam }) => get(pageParam),
    getNextPageParam: (last) => last.next,
  });
  return <button onClick={() => q.fetchNextPage()}
    disabled={!q.hasNextPage}>More</button>;
}
```

> **Note:** `initialPageParam` is required in v5; v4 inferred the first page param from the `queryFn` default argument.

Errors and retries

Option	Default	Effect
`retry`	`3` (client), `0` (server)	Attempts after failure

Option	Default	Effect
`retryDelay`	Exponential backoff	Wait between attempts
`throwOnError`	`false`	Throw to nearest error boundary
`useSuspenseQuery`	—	Suspends; `data` always defined

A query reaches `isError` only after the last retry, so a failing endpoint shows a spinner for several seconds first. Until then the error sits in `failureReason`. Skip retries for errors that cannot succeed, such as a 404.

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

class HttpError extends Error {
  constructor(public status: number) { super(); }
}
```

```
export const client = new QueryClient({
  defaultOptions: {
    queries: {
      retry: (count, err) =>
        !(err instanceof HttpError &&
          err.status < 500) && count < 3,
    },
  },
});
```

> **Gotcha:** `fetch` does not throw on 404 or 500. Check `res.ok` and throw in the `queryFn`, or the error is cached as success data.

SSR and Server Components

Step	Where
`queryClient.query(opts)`	Server: prefetch into a new client

Step	Where
`dehydrate(queryClient)`	Server: serialise the cache
`<HydrationBoundary state>`	Client Component boundary
`useQuery(opts)`	Client: reads hydrated data
`staleTime > 0` default	Stops an instant refetch

In the Next.js App Router, a Server Component prefetches and passes the dehydrated cache to a Client Component, which reads it with the same key. Create a new `QueryClient` per request on the server and one per browser on the client.

```
import {
  dehydrate, HydrationBoundary, noop, QueryClient,
} from "@tanstack/react-query";
const posts = {
  queryKey: ["posts"],
```

```
  queryFn: () => fetch("https://x.dev/p")
    .then((r) => r.json()),
};
export default async function Page() {
  const qc = new QueryClient();
  await qc.query(posts).catch(noop);
  const state = dehydrate(qc);
  return <HydrationBoundary state={state}>
    <p>Client list here</p></HydrationBoundary>;
}
```

> **Note:** `queryClient.query()` arrived in 5.102 and deprecates `prefetchQuery` and `fetchQuery`. If a Server Component can render the data itself and nothing on the client mutates or refetches it, fetch it there and skip TanStack Query.

Further reading

- TanStack Query — Important defaults

- TanStack Query — Query options
- TanStack Query — Optimistic updates
- TanStack Query — Advanced server rendering
- TanStack Query — Devtools