Mastering TanStack Query: An In-Depth Guide

This comprehensive guide delves into TanStack Query, the powerful data-fetching library for modern web applications. Discover its features, explore in-depth concepts, and learn how to harness TanStack Query to optimize data synchronization and state management in your projects.

Table of Contents

Introduction

Getting Started

Guides & Concepts

- Default Query Fn
- Suspense
- Testing
- Does this replace [Redux, MobX, etc]?
- Migrating to v3
- Migrating to v4
- Migrating to v5

API Reference

- QueryClient
- QueryCache
- MutationCache
- QueryObserver
- InfiniteQueryObserver
- QueriesObserver
- streamedQuery
- focusManager
- onlineManager
- notifyManager
- useQuery
- useQueries
- useInfiniteQuery
- useMutation
- useIsFetching
- useIsMutating
- useMutationState
- useSuspenseQuery
- useSuspenseInfiniteQuery
- useSuspenseQueries
- QueryClientProvider
- useQueryClient
- queryOptions
- infiniteQueryOptions
- usePrefetchQuery
- usePrefetchInfiniteQuery
- QueryErrorResetBoundary
- useQueryErrorResetBoundary
- hydration

ESLint

- ESLint Plugin Query
- Exhaustive Deps
- Stable Query Client
- No Rest Destructuring
- No Unstable Deps
- Infinite Query Property Order

Community Resources

- TkDodo's Blog
- Community Projects

Examples

- Simple
- Basic
- Basic w/ GraphQL-Request
- Auto Refetching / Polling / Realtime
- Optimistic Updates (UI)
- Optimistic Updates (Cache)
- Pagination
- Load-More & Infinite Scroll
- Infinite query with Max pages
- Suspense
- Default Query Function
- Playground
- Prefetching
- Star Wars
- Rick And Morty
- Next.js Pages
- Next.js app with prefetching
- Next.js app with streaming
- React Native
- React Router
- Offline Queries and Mutations
- Algolia
- Shadow DOM
- Devtools Embedded Panel
- Chat example (streaming)

Plugins

- persistQueryClient
- createSyncStoragePersister
- createAsyncStoragePersister
- broadcastQueryClient (Experimental)
- createPersister (Experimental)

TanStack Query

Overview

Powerful *asynchronous state management* for TS/JS, React, Solid, Vue, Svelte, and Angular

Toss out that granular state management, manual refetching, and endless bowls of async-spaghetti code. TanStack Query gives you declarative, always-up-to-date auto-managed queries and mutations that **directly improve both your developer and user experiences.**

Read the Docs

(or check out our official course 👇)

> "If you're serious about *really* understanding React Query, there's no better way than with query.gg"
> — Tanner Linsley

Join now

Declarative & Automatic

Writing your data fetching logic by hand is over. Tell TanStack Query where to get your data and how fresh you need it to be and the rest is automatic. It handles **caching, background updates, and stale data out of the box with zero-configuration.**

Simple & Familiar

If you know how to work with promises or async/await, then you already know how to use TanStack Query. There's **no global state to manage, reducers, normalization systems** or heavy configurations to understand. Simply pass a function that resolves your data (or throws an error) and the rest is history.

Extensible

TanStack Query is configurable down to each observer instance of a query with knobs and options to fit every use-case. It comes wired up with **dedicated devtools, infinite-loading APIs, and first class mutation tools** that make updating your data a breeze. Don't worry though, everything is pre-configured for success!

No dependencies. All the Features.

With zero dependencies, TanStack Query is extremely lean given the dense feature set it provides. From weekend hobbies all the way to enterprise e-commerce systems (Yes, I'm lookin' at you Walmart! 😉), TanStack Query is the battle-hardened tool to help you succeed at the speed of your creativity.

Features	Description
Backend agnostic	
Dedicated Devtools	
Auto Caching	
Auto Refetching	
Window Focus Refetching	
Polling/Realtime Queries	

Features	Description
Parallel Queries	
Dependent Queries	
Mutations API	
Automatic Garbage Collection	
Paginated/Cursor Queries	
Load-More/Infinite Scroll Queries	
Scroll Recovery	
Request Cancellation	
Suspense Ready!	
Render-as-you-fetch	
Prefetching	
Variable-length Parallel Queries	
Offline Support	
SSR Support	
Data Selectors	

Trusted in Production by

Incredible Partners

> Speakeasy and TanStack collaborate to make API management effortless
> Speakeasy's **automated SDK generation** and TanStack's robust front-end tooling enable faster development. From server to client, get **powerful, type-safe, and optimized solutions**.
> *Learn More*

Sponsors

Become a Sponsor!

About Us

> TanStack is 100% privately owned, with no paid products, venture capital, or acquisition plans.
> We're a small team dedicated to creating software used by millions daily. What did you expect?

Check out our ethos to learn more about how we plan on sticking around (and staying relevant) for the long-haul.

Less code, fewer edge cases.

Instead of writing reducers, caching logic, timers, retry logic, complex async/await scripting (I could keep going...), you literally write a tiny fraction of the code you normally would. You will be surprised at how little code you're writing or how much code you're deleting when you use TanStack Query. Try it out with one of the examples below!

Frameworks: Angular | React | Solid | Svelte | Vue

Next Steps

Wow, you've come a long way!
Only one thing left to do...
Read the Docs!

Connect with Us

- Blog
- Twitter (@Tan_Stack)
- TannerLinsley Twitter (@TannerLinsley)
- GitHub
- YouTube
- Nozzle.io - Keyword Rank Tracker
- Privacy Policy
- Terms of Service

TanStack Query Docs

Navigation and Header

TanStack | Query v5

Auto

Search and Menu

Search...
+ K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview (react)
- Installation (react)
- Quick Start (react)
- Devtools (react)
- Videos & Talks (react)
- Comparison (react)
- TypeScript (react)
- GraphQL (react)
- React Native (react)

Guides & Concepts

- Important Defaults (react)
- Queries (react)
- Query Keys (react)
- Query Functions (react)
- Query Options (react)
- Network Mode (react)
- Parallel Queries (react)
- Dependent Queries (react)
- Background Fetching Indicators (react)
- Window Focus Refetching (react)
- Disabling/Pausing Queries (react)

- Query Retries (react)
- Paginated Queries (react)
- Infinite Queries (react)
- Initial Query Data (react)
- Placeholder Query Data (react)
- Mutations (react)
- Query Invalidation (react)
- Invalidation from Mutations (react)
- Updates from Mutation Responses (react)
- Optimistic Updates (react)
- Query Cancellation (react)
- Scroll Restoration (react)
- Filters (react)
- Performance & Request Waterfalls (react)
- Prefetching & Router Integration (react)
- Server Rendering & Hydration (react)
- Advanced Server Rendering (react)
- Caching (react)
- Render Optimizations (react)
- Default Query Fn (react)
- Suspense (react)
- Testing (react)
- Does this replace [Redux, MobX, etc]? (react)
- Migrating to v3 (react)
- Migrating to v4 (react)
- Migrating to v5 (react)

API Reference

- QueryClient
- QueryCache
- MutationCache
- QueryObserver
- InfiniteQueryObserver
- QueriesObserver
- streamedQuery
- focusManager
- onlineManager
- notifyManager
- useQuery
- useQueries
- useInfiniteQuery
- useMutation
- useIsFetching
- useIsMutating
- useMutationState
- useSuspenseQuery
- useSuspenseInfiniteQuery
- useSuspenseQueries
- QueryClientProvider
- useQueryClient
- queryOptions
- infiniteQueryOptions
- usePrefetchQuery
- usePrefetchInfiniteQuery
- QueryErrorResetBoundary

- useQueryErrorResetBoundary
- hydration

ESLint

- ESLint Plugin Query
- Exhaustive Deps
- Stable Query Client
- No Rest Destructuring
- No Unstable Deps
- Infinite Query Property Order

Community Resources

- TkDodo's Blog (react)
- Community Projects (react)

Examples

- Simple (react)
- Basic (react)
- Basic w/ GraphQL-Request (react)
- Auto Refetching / Polling / Realtime (react)
- Optimistic Updates (UI) (react)
- Optimistic Updates (Cache) (react)
- Pagination (react)
- Load-More & Infinite Scroll (react)
- Infinite query with Max pages (react)
- Suspense (react)
- Default Query Function (react)
- Playground (react)
- Prefetching (react)
- Star Wars (react)
- Rick And Morty (react)
- Next.js Pages (react)
- Next.js app with prefetching (react)
- Next.js app with streaming (react)
- React Native (react)
- React Router (react)
- Offline Queries and Mutations (react)
- Algolia (react)
- Shadow DOM (react)
- Devtools Embedded Panel (react)
- Chat example (streaming) (react)

Plugins

- persistQueryClient (react)
- createSyncStoragePersister (react)
- createAsyncStoragePersister (react)
- broadcastQueryClient (Experimental) (react)

- createPersister (Experimental) (react)

Supported TanStack Query Frameworks

- TanStack React Query
- TanStack Vue Query
- TanStack Angular Query
- TanStack Solid Query
- TanStack Svelte Query

Footer and Additional Links

Home

Our Partners

- React Query on Speakeasy

Other Resources

- Want to Skip the Docs?
  “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg” — *Tanner Linsley*

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
  Learn More

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!
  Learn More

Additional Query.gg Promotion

Want to Skip the Docs?
“*If you’re serious about really understanding React Query, there’s no better way than with query.gg*” — *Tanner Linsley*

TanStack Query Docs

Navigation

- TanStack
- Query v5
- Auto

Search

Search...

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview React
- Installation React
- Quick Start React
- Devtools React
- Videos & Talks React
- Comparison React
- TypeScript React
- GraphQL React
- React Native React

Guides & Concepts

- Important Defaults React
- Queries React
- Query Keys React
- Query Functions React
- Query Options React
- Network Mode React
- Parallel Queries React
- Dependent Queries React
- Background Fetching Indicators React
- Window Focus Refetching React
- Disabling/Pausing Queries React
- Query Retries React
- Paginated Queries React
- Infinite Queries React
- Initial Query Data React
- Placeholder Query Data React
- Mutations React

- Query Invalidation React
- Invalidation from Mutations React
- Updates from Mutation Responses React
- Optimistic Updates React
- Query Cancellation React
- Scroll Restoration React
- Filters React
- Performance & Request Waterfalls React
- Prefetching & Router Integration React
- Server Rendering & Hydration React
- Advanced Server Rendering React
- Caching React
- Render Optimizations React
- Default Query Fn React
- Suspense React
- Testing React
- Does this replace Redux, MobX, etc? React
- Migrating to v3 React
- Migrating to v4 React
- Migrating to v5 React

API Reference

- QueryClient core
- QueryCache core
- MutationCache core
- QueryObserver core
- InfiniteQueryObserver core
- QueriesObserver core
- streamedQuery core
- focusManager core
- onlineManager core
- notifyManager core
- useQuery react
- useQueries react
- useInfiniteQuery react
- useMutation react
- useIsFetching react
- useIsMutating react
- useMutationState react
- useSuspenseQuery react
- useSuspenseInfiniteQuery react
- useSuspenseQueries react
- QueryClientProvider react
- useQueryClient react
- queryOptions react
- infiniteQueryOptions react
- usePrefetchQuery react
- usePrefetchInfiniteQuery react
- QueryErrorResetBoundary react
- useQueryErrorResetBoundary react
- hydration react

ESLint

- ESLint Plugin Query core

- Exhaustive Deps core
- Stable Query Client core
- No Rest Destructuring core
- No Unstable Deps core
- Infinite Query Property Order core

Community Resources

- TkDodo's Blog React
- Community Projects React

Examples

- Simple React
- Basic React
- Basic w/ GraphQL-Request React
- Auto Refetching / Polling / Realtime React
- Optimistic Updates (UI) React
- Optimistic Updates (Cache) React
- Pagination React
- Load-More & Infinite Scroll React
- Infinite query with Max pages React
- Suspense React
- Default Query Function React
- Playground React
- Prefetching React
- Star Wars React
- Rick And Morty React
- Next.js Pages React
- Next.js app with prefetching React
- Next.js app with streaming React
- React Native React
- React Router React
- Offline Queries and Mutations React
- Algolia React
- Shadow DOM React
- Devtools Embedded Panel React
- Chat example (streaming) React

Plugins

- persistQueryClient React
- createSyncStoragePersister React
- createAsyncStoragePersister React
- broadcastQueryClient (Experimental) React
- createPersister (Experimental) React

Maintainers and Contributors

React, Solid Architecture, Core API, Documentation

- Tanner Linsley
  Roles: Creator
  

React, Core API, TypeScript, Documentation

- Dominik Dorfmeister
  Roles: Maintainer
  

React, Svelte Architecture, Maintainer

- Lachlan Collins
  Roles: Maintainer

React, Data Management, SSR, Hydration

- Fredrik Höglund
  Roles: Maintainer
  

React, TypeScript, Backport, Test

- Jonghyeon Ko
  Roles: Maintainer
  

Angular, React Architecture, Developer Experience, TypeScript, Reactivity

- Arnoud de Vries
  Roles: Maintainer
  

Solid Development Tools

- Aryan Deora
  Roles: Maintainer

Vue, Maintainer

- Damian Osipiuk
  Roles: Maintainer

ESLint Plugin

- Eliya Cohen
  Roles: Maintainer

All-Time Contributors

Powered by contrib.rocks
View all contributors on GitHub

Hello

TanStack

Overview

TanStack Query React Docs

On this page

Overview

TanStack Query (formerly known as React Query) is often described as the missing data-fetching library for web applications, but in more technical terms, it makes **fetching, caching, synchronizing and updating server state** in your web applications a breeze.

Motivation

Most core web frameworks **do not** come with an opinionated way of fetching or updating data in a holistic way. Because of this developers end up building either meta-frameworks which encapsulate strict opinions about data-fetching, or they invent their own ways of fetching data. This usually means cobbling together component-based state and side-effects, or using more general purpose state management libraries to store and provide asynchronous data throughout their apps.

While most traditional state management libraries are great for working with client state, they are **not so great at working with async or server state**. This is because **server state is totally different**. For starters, server state:

- Is persisted remotely in a location you may not control or own
- Requires asynchronous APIs for fetching and updating
- Implies shared ownership and can be changed by other people without your knowledge
- Can potentially become "out of date" in your applications if you're not careful

Once you grasp the nature of server state in your application, **even more challenges will arise** as you go, for example:

- Caching... (possibly the hardest thing to do in programming)
- Deduping multiple requests for the same data into a single request
- Updating "out of date" data in the background
- Knowing when data is "out of date"
- Reflecting updates to data as quickly as possible
- Performance optimizations like pagination and lazy loading data
- Managing memory and garbage collection of server state
- Memoizing query results with structural sharing

If you're not overwhelmed by that list, then that must mean that you've probably solved all of your server state problems already and deserve an award. However, if you are like a vast majority of people, you either have yet to tackle all or most of these challenges and we're only scratching the surface!

TanStack Query is hands down one of the **best** libraries for managing server state. It works amazingly well **out-of-the-box, with zero-config, and can be customized** to your liking as your application grows.

TanStack Query allows you to defeat and overcome the tricky challenges and hurdles of *server state* and control your app data before it starts to control you.

On a more technical note, TanStack Query will likely:

- Help you remove **many** lines of complicated and misunderstood code from your application and replace with just a handful of lines of TanStack Query logic
- Make your application more maintainable and easier to build new features without worrying about wiring up new server state data sources
- Have a direct impact on your end-users by making your application feel faster and more responsive than ever before
- Potentially help you save on bandwidth and increase memory performance

---

Enough talk, show me some code already!

In the example below, you can see TanStack Query in its most basic and simple form being used to fetch the GitHub stats for the TanStack Query GitHub project itself:

Open in StackBlitz

```
import {
  QueryClient,
  QueryClientProvider,
  useQuery,
} from '@tanstack/react-query'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { isPending, error, data } = useQuery({
    queryKey: ['repoData'],
    queryFn: () =>
      fetch('https://api.github.com/repos/TanStack/query').then((res) =>
        res.json()
      ),
  })

  if (isPending) return 'Loading...'

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.description}</p>
      <strong>👀 {data.subscribers_count}</strong>{' '}
      <strong>✨ {data.stargazers_count}</strong>{' '}
```

```
      <strong>🍴 {data.forks_count}</strong>
    </div>
  )
}
```

You talked me into it, so what now?

- Consider taking the official TanStack Query Course (or buying it for your whole team!)
- Learn TanStack Query at your own pace with our thoroughly designed Walkthrough Guide and API Reference

---

Edit on GitHub

On this page

Installation

On this page

Overview

You can install React Query via NPM, or a good ol' `<script>` via ESM.sh.

NPM

```
npm i @tanstack/react-query
```

or

```
pnpm add @tanstack/react-query
```

or

```
yarn add @tanstack/react-query
```

or

```
bun add @tanstack/react-query
```

React Query is compatible with React v18+ and works with ReactDOM and React Native.

> **Wanna give it a spin before you download?**
> Try out the simple or basic examples!

CDN

If you're not using a module bundler or package manager, you can also use this library via an ESM-compatible CDN such as ESM.sh. Simply add a `<script type="module">` tag to the bottom of your HTML file:

```
<script type="module">
  import React from 'https://esm.sh/react@18.2.0'
  import ReactDOM from 'https://esm.sh/react-dom@18.2.0'
  import { QueryClient } from 'https://esm.sh/@tanstack/react-query'
</script>
```

> You can find instructions on how to use React without JSX here.

Requirements

React Query is optimized for modern browsers. It is compatible with the following browsers:

```
Chrome >= 91
Firefox >= 90
Edge >= 91
```

```
Safari >= 15
iOS >= 15
Opera >= 77
```

> Depending on your environment, you might need to add polyfills. If you want to support older browsers, you need to transpile the library from `node_modules` yourselves.

Recommendations

It is recommended to also use our ESLint Plugin Query to help you catch bugs and inconsistencies while you code. You can install it via:

```
npm i -D @tanstack/eslint-plugin-query
```

or

```
pnpm add -D @tanstack/eslint-plugin-query
```

or

```
yarn add -D @tanstack/eslint-plugin-query
```

or

```
bun add -D @tanstack/eslint-plugin-query
```

Edit on GitHub

On this page

Quick Start | TanStack Query React Docs

Overview

This guide briefly illustrates the three core concepts of React Query:

- Queries
- Mutations
- Query Invalidation

If you're looking for a fully functioning example, please have a look at our simple StackBlitz example.

Example Code (TSX)

```
import {
  useQuery,
  useMutation,
  useQueryClient,
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'
import { getTodos, postTodo } from '../my-api'

// Create a client
const queryClient = new QueryClient()

function App() {
  return (
    // Provide the client to your App
    <QueryClientProvider client={queryClient}>
      <Todos />
    </QueryClientProvider>
  )
}

function Todos() {
  // Access the client
  const queryClient = useQueryClient()

  // Queries
  const query = useQuery({ queryKey: ['todos'], queryFn: getTodos })

  // Mutations
  const mutation = useMutation({
    mutationFn: postTodo,
    onSuccess: () => {
      // Invalidate and refetch
      queryClient.invalidateQueries({ queryKey: ['todos'] })
```

```
    },
  })

  return (
    <div>
      <ul>{query.data?.map((todo) => <li key={todo.id}>{todo.title}</li>)}</ul>

      <button
        onClick={() => {
          mutation.mutate({
            id: Date.now(),
            title: 'Do Laundry',
          })
        }}
      >
        Add Todo
      </button>
    </div>
  )
}

render(<App />, document.getElementById('root'))
```

(Note: The second code block is identical to the first, so omitted for brevity)

Summary

These three concepts make up most of the core functionality of React Query. The next sections of the documentation will go over each of these core concepts in great detail.

Edit on GitHub

Edit on GitHub

Additional Resources

Installation & Devtools

- Installation
- Devtools

Our Partners

- React Query Course by Query.gg *(Learn More)*

Related Tools

- TanStack Table: Supercharge your tables or build a datagrid for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, and Lit while retaining full control.

- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations for reactive, consistent, and fast UI.

Skip the Docs?

Query.gg - The Official React Query Course
"If you're serious about really understanding React Query, there's no better way than with query.gg." — Tanner Linsley

Devtools | TanStack Query React Docs

On this page

Devtools

Wave your hands in the air and shout hooray because React Query comes with dedicated devtools! 🥳

When you begin your React Query journey, you'll want these devtools by your side. They help visualize all the inner workings of React Query and will likely save you hours of debugging if you find yourself in a pinch!

> For Google Chrome users: A third-party Chrome extension is available for debugging TanStack Query directly in Chrome DevTools. This provides the same functionality as the framework-specific devtools packages. Check it out here: TanStack Query DevTools
>
> For React Native users: A third-party native macOS app is available for debugging React Query in ANY js-based application. Monitor queries across devices in real-time. Check it out here: rn-better-dev-tools
>
> Note that since version 5, the dev tools support observing mutations as well.

Install and Import the Devtools

The devtools are a separate package that you need to install:

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

or

```
pnpm add @tanstack/react-query-devtools
```

or

```
yarn add @tanstack/react-query-devtools
```

or

```
bun add @tanstack/react-query-devtools
```

For Next 13+ App Directory, you must install it as a dev dependency to ensure proper functioning.

You can import the devtools like this:

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

By default, React Query Devtools are only included in bundles when `process.env.NODE_ENV === 'development'`, so you don't need to worry about excluding them during a production build.

Floating Mode

Floating Mode will mount the devtools as a fixed, floating element in your app and provide a toggle in the corner of the screen to show and hide the devtools. This toggle state will be stored and remembered in localStorage across reloads.

Place the following code as high in your React app as you can. The closer it is to the root of the page, the better it will work!

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

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      {/* The rest of your application */}
      <ReactQueryDevtools initialIsOpen={false} />
    </QueryClientProvider>
  )
}
```

Options

- `initialIsOpen: Boolean`
  Set to `true` if you want the dev tools to default to being open.
- `buttonPosition?: "top-left" | "top-right" | "bottom-left" | "bottom-right" | "relative"`
  Defaults to `bottom-right`.
  Determines the position of the React Query logo to open and close the devtools panel.
  If `relative`, the button is placed in the location that you render the devtools.
- `position?: "top" | "bottom" | "left" | "right"`
  Defaults to `bottom`.
  Position of the React Query devtools panel.
- `client?: QueryClient`
  Use a custom `QueryClient`. Otherwise, the one from the nearest context will be used.
- `errorTypes?: { name: string; initializer: (query: Query) => TError }[]`
  Predefine errors that can be triggered on your queries. The initializer will be called when the error is toggled on from the UI; it must return an Error object.
- `styleNonce?: string`
  Pass a nonce to the style tag for Content Security Policy (CSP) compliance.
- `shadowDOMTarget?: ShadowRoot`
  Applies styles to a specific Shadow DOM instead of the document head.

Embedded Mode

Embedded mode shows the development tools as a fixed element within your application.

Place this code high in your React app:

```
import { ReactQueryDevtoolsPanel } from '@tanstack/react-query-devtools'

function App() {
  const [isOpen, setIsOpen] = React.useState(false)

  return (
    <QueryClientProvider client={queryClient}>
      {/* The rest of your application */}
      <button onClick={() => setIsOpen(!isOpen)}>
        {`${isOpen ? 'Close' : 'Open'} the devtools panel`}
      </button>
      {isOpen && (
        <ReactQueryDevtoolsPanel onClose={() => setIsOpen(false)} />
      )}
    </QueryClientProvider>
  )
}
```

Options for embedded mode include:

- `style?: React.CSSProperties`
  Custom styles for the devtools panel. Default: `{ height: '500px' }`.
- `onClose?: () => unknown`
  Callback when the panel is closed.
- Additional options similar to Floating Mode (`client`, `errorTypes`, `styleNonce`, `shadowDOMTarget`).

Devtools in production

Devtools are excluded in production builds. However, you might want to lazy load the devtools in production:

```
import * as React from 'react'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import { Example } from './Example'

const queryClient = new QueryClient()

const ReactQueryDevtoolsProduction = React.lazy(() =>
  import('@tanstack/react-query-devtools/build/modern/production.js').then(
    (d) => ({
      default: d.ReactQueryDevtools,
    })
  )
)

function App() {
  const [showDevtools, setShowDevtools] = React.useState(false)

  React.useEffect(() => {
```

```
    // @ts-expect-error
    window.toggleDevtools = () => setShowDevtools((old) => !old)
  }, [])

  return (
    <QueryClientProvider client={queryClient}>
      <Example />
      <ReactQueryDevtools initialIsOpen={false} />
      {showDevtools && (
        <React.Suspense fallback={null}>
          <ReactQueryDevtoolsProduction />
        </React.Suspense>
      )}
    </QueryClientProvider>
  )
}
```

Calling `window.toggleDevtools()` will download and show the devtools bundle.

Modern bundlers

If your bundler supports package exports, you can import like:

```
const ReactQueryDevtoolsProduction = React.lazy(() =>
  import('@tanstack/react-query-devtools/production').then((d) => ({
    default: d.ReactQueryDevtools,
  }))
)
```

Make sure your `tsconfig.json` has:

```
"moduleResolution": "nodenext"
```

(Requires TypeScript v4.7+)

Edit on GitHub

Videos & Talks | TanStack Query React Docs

Click here to view the Repository used for the above presentation

Edit on GitHub

Devtools

Devtools

Comparison

Comparison

Our Partners

Want to Skip the Docs?
Query.gg - The Official React Query Course

> "If you're serious about *really* understanding React Query, there's no better way than with query.gg"
> — Tanner Linsley

Learn More

Resources

TanStack	Description	Link
Table	Supercharge your tables or build a data grid from scratch for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, and Lit while retaining 100% control over markup and styles.	Learn More
DB	TanStack DB extends TanStack Query with collections, live queries, and optimistic mutations that keep your UI reactive, consistent, and blazing fast 🔥	Learn More

---

Additional

Want to Skip the Docs?
Query.gg - The Official React Query Course

> "If you're serious about *really* understanding React Query, there's no better way than with query.gg"
> — Tanner Linsley

Comparison | React Query vs SWR vs Apollo vs RTK Query vs React Router

---

Notes

[1] Lagged Query Data - React Query provides a way to continue to see an existing query's data while the next query loads (similar to the same UX that suspense will soon provide natively). This is extremely important when writing pagination UIs or infinite loading UIs where you do not want to show a hard loading state whenever a new query is requested. Other libraries do not have this capability and render a hard loading state for the new query (unless it has been prefetched), while the new query loads.

[2] Render Optimization - React Query has excellent rendering performance. By default, it will automatically track which fields are accessed and only re-render if one of them changes. To opt-out, set `notifyOnChangeProps` to `'all'` to re-render on any query update. React Query batches updates to minimize re-renders. For more optimized updates, set `notifyOnChangeProps` to `['data', 'error']`.

[3] Partial query matching - Because React Query uses deterministic query key serialization, you can manipulate variable groups of queries without knowing each key. For example, refetch every query starting with `todos`, regardless of variables, or target specific nested properties, with optional filters.

[4] Pre-usage Query Configuration - Configuring query/mutation defaults before use, so usage is simplified to `useQuery({ queryKey })`. SWR allows global default fetcher, but no per-query or mutation defaults.

[5] Automatic Refetch after Mutation - For automatic refetching post-mutation, a schema (like GraphQL) with heuristics for entity identification is necessary.

[6] Normalized Caching - React Query, SWR, RTK-Query do not currently support automatic normalized caching (storing entities in flat structures to prevent duplication).

[7] SWR's Immutable Mode - SWR's "immutable" mode fetches only once per cache lifetime but lacks `staleTime` or auto-revalidation.

[8] React Router cache persistence - React Router does not cache data beyond current routes; leaving a route discards its data.

Comparison Table

#	Feature/Capability	React Query	SWR Website	Apollo Client Website	RTK-Query Website	React Router Website
1	Github Repo / Stars	GitHub	GitHub	GitHub	GitHub	GitHub
2	Platform Requirements	React	React	React, GraphQL	Redux	React
3	Their Comparison		none	none	Comparison	none

#	Feature/Capability	React Query	SWR Website	Apollo Client Website	RTK-Query Website	React Router Website
4	Supported Query Syntax	Promise, REST, GraphQL	Promise, REST, GraphQL	GraphQL, any (Reactive Variables)	Promise, REST, GraphQL	Promise, REST, GraphQL
5	Supported Frameworks	React	React	React + Others	Any	React
6	Caching Strategy	Hierarchical Key -> Value	Unique Key -> Value	Normalized Schema	Unique Key -> Value	Nested Route -> value
7	Cache Key Strategy	JSON	JSON	GraphQL Query	JSON	Route Path
8	Cache Change Detection	Deep Compare Keys (Stable Serialization)	Deep Compare Keys (Stable Serialization)	Deep Compare Keys (Unstable Serialization)	Key Referential Equality (===)	Route Change
9	Data Change Detection	Deep Comparison + Structural Sharing	Deep Compare (via *stable-hash*)	Deep Compare (Unstable Serialization)	Key Referential Equality (===)	Loader Run
10	Data Memoization	Full Structural Sharing	Identity (===)	Normalized Identity	Identity (===)	Identity (===)
11	Bundle Size	bundlephobia	bundlephobia	bundlephobia	bundlephobia	bundlephobia + history
12	API Definition Location	Component, External Config	Component	GraphQL Schema	External Config	Route Tree Configuration
13	Queries	✅	✅	✅	✅	✅
14	Cache Persistence	✅	✅	✅	✅	🛑 Active Routes Only [8]
15	Devtools	✅	✅	✅	✅	🛑
16	Polling/Intervals	✅	✅	✅	✅	🛑
17	Parallel Queries	✅	✅	✅	✅	✅
18	Dependent Queries	✅	✅	✅	✅	✅
19	Paginated Queries	✅	✅	✅	✅	✅
20	Infinite Queries	✅	✅	✅	✅	🛑
21	Bi-directional Infinite Queries	✅	🔶	🔶	✅	🛑
22	Infinite Query Refetching	✅	✅	🛑	✅	🛑
23	Lagged Query Data [1]	✅	✅	✅	✅	✅
24	Selectors	✅	🛑	✅	✅	N/A
25	Initial Data	✅	✅	✅	✅	✅
26	Scroll Recovery	✅	✅	✅	✅	✅
27	Cache Manipulation	✅	✅	✅	✅	🛑
28	Outdated Query Dismissal	✅	✅	✅	✅	✅
29	Render Batching & Optimizations [2]	✅	✅	🛑	✅	✅
30	Auto Garbage Collection	✅	🛑	🛑	✅	N/A
31	Mutation Hooks	✅	✅	✅	✅	✅
32	Offline Mutation Support	✅	🛑	🟡	🛑	🛑
33	Prefetching APIs	✅	✅	✅	✅	✅
34	Query Cancellation	✅	🛑	🛑	🛑	✅
35	Partial Query Matching [3]	✅	🔶	✅	✅	N/A
36	Stale While Revalidate	✅	✅	✅	✅	🛑
37	Stale Time Configuration	✅	🛑[7]	🛑	✅	🛑

#	Feature/Capability	React Query	SWR Website	Apollo Client Website	RTK-Query Website	React Router Website
38	Pre-usage Query/Mutation Configuration[4]	✅	🛑	✅	✅	✅
39	Window Focus Refetching	✅	✅	🛑	✅	🛑
40	Network Status Refetching	✅	✅	✅	✅	🛑
41	General Cache Dehydration/Rehydration	✅	🛑	✅	✅	✅
42	Offline Caching	✅	🛑	✅	🔶	🛑
43	React Suspense	✅	✅	✅	🛑	✅
44	Abstracted/Agnostic Core	✅	🛑	✅	✅	🛑
45	Automatic Refetch after Mutation[5]	🔶	🔶	✅	✅	✅
46	Normalized Caching[6]	🛑	🛑	✅	🛑	🛑

Additional sections

Notes

(Already included above within blockquotes)

Edit on GitHub

Videos & Talks

- Videos & Talks
- TypeScript

Our Partners

- 🤝 Want to Skip the Docs? Query.gg - The Official React Query Course
  "*If you're serious about really understanding React Query, there's no better way than with query.gg — Tanner Linsley*"

Other Resources

- TanStack Table: "Supercharge your tables or build a datagrid from scratch for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, and Lit while retaining 100% control over markup and styles."
- TanStack DB: "TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥"

Note: The original HTML contains a very long, nested structure, images, links, and complex content. The above Markdown simplifies the structure, translating headings, lists, links, tables, blockquotes, and notes, while maintaining the semantic style and key information.

TypeScript | TanStack Query React Docs

Overview

React Query and TypeScript

React Query is now written in **TypeScript** to ensure type safety across your projects!

Important notes:

- Types require TypeScript **v4.7** or later.
- Changes to types are **non-breaking** and usually released as **patch** version updates.
- It's **highly recommended** to lock the react-query package to a specific patch version, as types may upgrade between releases.
- The public API remains very stable via semver.

Type Inference

Types in React Query generally flow seamlessly, reducing the need for explicit type annotations.

```
const { data } = useQuery({
  //    ^? const data: number | undefined
  queryKey: ['test'],
  queryFn: () => Promise.resolve(5),
})
```

Try on TypeScript Playground

```
const { data } = useQuery({
  //    ^? const data: string | undefined
  queryKey: ['test'],
  queryFn: () => Promise.resolve(5),
  select: (data) => data.toString(),
})
```

Try on TypeScript Playground

This works best when your `queryFn` has a well-defined return type. Since most libraries default to `any`, extract your fetch logic into typed functions:

```
const fetchGroups = (): Promise<Group[]> =>
  axios.get('/groups').then((response) => response.data)

const { data } = useQuery({ queryKey: ['groups'], queryFn: fetchGroups })
//    ^? const data: Group[] | undefined
```

Try on TypeScript Playground

Type Narrowing

React Query uses *discriminated union types* based on the `status` field, enabling safe type narrowing with flags like `isSuccess`.

```
const { data, isSuccess } = useQuery({
  queryKey: ['test'],
  queryFn: () => Promise.resolve(5),
})

if (isSuccess) {
  // data is assuredly defined
  //  ^? const data: number
}
```

Try on TypeScript Playground

This pattern allows you to execute type-safe logic after confirming the query's status.

Typing the Error Field

By default, the error type is `Error`, aligning with common expectations.

```
const { error } = useQuery({ queryKey: ['groups'], queryFn: fetchGroups })
//    ^? const error: Error
```

Try on TypeScript Playground

If you need to throw a custom error type, specify it via generics:

```
const { error } = useQuery<Group[], string>(['groups'], fetchGroups)
//    ^? const error: string | null
```

However, this can complicate inference elsewhere. A safer approach when working with subclasses like `AxiosError` is to *narrow* the error type:

```
import axios from 'axios'

const { error } = useQuery({ queryKey: ['groups'], queryFn: fetchGroups })
//    ^? const error: Error | null

if (axios.isAxiosError(error)) {
  error
  // ^? const error: AxiosError
}
```

[Try on TypeScript Playground](https://www.typescriptlang.org/play?#code/JYWwDg9gTgLgBAbzgVwM4FMCKz1QJ5wC+cAZlBCHAOQACMAhgHaoMDGA1gPRTr2swBaAI458VALAAoUJFhx6AD2ARUpcpSqLlqCZKkw8YdHADi5ZGDgBeRHGAATAFxxGyEACNcRKVNYRm8CToMKwAFmYQFqo2ABQAlM4ACurAGAA8ERYA2gC6AHzWBVoqAHQA5sExVJxl5mA6cSUwoeiMMTyokMzGVgUdXRgl9vQMcT6SfgG2uORQRNYoGNi4eDFIIisA0uh4zllUtZH1VDkANHAb+ABijM5BIeF1qoRjkpyccJ9fAHoA-OPAEhwGLFVAlVIAQSUKgAolBZjEZtA4nFEFJPkioOi4O84H8pIQgA

Registering a Global Error

````
TanStack Query v5 supports setting a **global error type** across all queries/mutation

```tsx
import '@tanstack/react-query'

declare module '@tanstack/react-query' {
  interface Register {
    defaultError: AxiosError
  }
}

const { error } = useQuery({ queryKey: ['groups'], queryFn: fetchGroups })
//    ^? const error: AxiosError | null
````

Try on TypeScript Playground

Typing Meta

You can define a global `Meta` type to keep consistency:

```
import '@tanstack/react-query'

interface MyMeta extends Record<string, unknown> {
  // your meta fields
}

declare module '@tanstack/react-query' {
  interface Register {
    queryMeta: MyMeta
    mutationMeta: MyMeta
  }
}
```

Same link for playground

Typing Query and Mutation Keys

Registering key types

```
import '@tanstack/react-query'

type QueryKey = ['dashboard' | 'marketing', ...ReadonlyArray<unknown>]

declare module '@tanstack/react-query' {
  interface Register {
    queryKey: QueryKey
    mutationKey: QueryKey
  }
}
```

Same playground link

```

Typing Query Options

You can create reusable typed options:

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

function groupOptions() {
  return queryOptions({
    queryKey: ['groups'],
    queryFn: fetchGroups,
    staleTime: 5000,
  })
}

useQuery(groupOptions())
queryClient.prefetchQuery(groupOptions())
```

Since the returned `queryKey` includes the `queryFn` type, functions like `queryClient.getQueryData` are aware of data types:

```
const data = queryClient.getQueryData(groupOptions().queryKey)
//    ^? const data: Group[] | undefined
```

Without `queryOptions`, you'd need to specify generics:

```
const data = queryClient.getQueryData<Group[]>(['groups'])
```

Further reading on type inference

Further Reading

- React Query and TypeScript
- Type-safe React Query

Typesafe Disabling of Queries with `skipToken`

You can use the `skipToken` to conditionally disable queries while maintaining type safety. See the guide: Disabling Queries

---

Other Links

Edit on GitHub

On this page

GraphQL | TanStack Query React Docs

On this page

GraphQL

Because React Query's fetching mechanisms are agnostically built on Promises, you can use React Query with literally any asynchronous data fetching client, including GraphQL!

> *Keep in mind that React Query does not support normalized caching. While a vast majority of users do not actually need a normalized cache or even benefit from it as much as they believe they do, there may be very rare circumstances that may warrant it so be sure to check with us first to make sure it's truly something you need!*

Type-Safety and Code Generation

React Query, used in combination with graphql-request^5 and GraphQL Code Generator, provides full-typed GraphQL operations:

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

import { graphql } from './gql/gql'

const allFilmsWithVariablesQueryDocument = graphql(/* GraphQL */ `
  query allFilmsWithVariablesQuery($first: Int!) {
    allFilms(first: $first) {
      edges {
        node {
          id
          title
        }
      }
    }
  }
`)

function App() {
  // `data` is fully typed!
  const { data } = useQuery({
    queryKey: ['films'],
    queryFn: async () =>
      request(
        'https://swapi-graphql.netlify.app/.netlify/functions/index',
        allFilmsWithVariablesQueryDocument,
        // variables are type-checked too!
```

```
        { first: 10 }
      ),
  })
  // ...
}

import request from 'graphql-request'
import { useQuery } from '@tanstack/react-query'

import { graphql } from './gql/gql'

const allFilmsWithVariablesQueryDocument = graphql(/* GraphQL */ `
  query allFilmsWithVariablesQuery($first: Int!) {
    allFilms(first: $first) {
      edges {
        node {
          id
          title
        }
      }
    }
  }
`)

function App() {
  // `data` is fully typed!
  const { data } = useQuery({
    queryKey: ['films'],
    queryFn: async () =>
      request(
        'https://swapi-graphql.netlify.app/.netlify/functions/index',
        allFilmsWithVariablesQueryDocument,
        // variables are type-checked too!
        { first: 10 }
      ),
  })
  // ...
}
```

You can find a complete example in the repo

Get started with the dedicated guide on GraphQL Code Generator documentation.

Our Partners

- Want to Skip the Docs?
  Query.gg - The Official React Query Course

  “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg”
  — Tanner Linsley

- Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

Learn More

- The build and publish utilities used by all of our projects. Use it if you dare!
  Learn More

Additional Resources

- TypeScript
- React Native

Community Resources

- TkDodo's Blog
- Community Projects

Examples

- Simple
- Basic
- Basic w/ GraphQL-Request
- Auto Refetching / Polling / Realtime
- Optimistic Updates (UI)
- Optimistic Updates (Cache)
- Pagination
- Load-More & Infinite Scroll
- Infinite query with Max pages
- Suspense
- Default Query Function
- Playground
- Prefetching
- Star Wars
- Rick And Morty
- Next.js Pages
- Next.js app with prefetching
- Next.js app with streaming
- React Native
- React Router
- Offline Queries and Mutations
- Algolia
- Shadow DOM
- Devtools Embedded Panel
- Chat example (streaming)

Plugins

- persistQueryClient
- createSyncStoragePersister
- createAsyncStoragePersister
- broadcastQueryClient (Experimental)
- createPersister (Experimental)

Type-Safety and Code Generation

See here for details.

Edit on GitHub

React Native | TanStack Query React Docs

React Native

React Query is designed to work out of the box with React Native.

DevTools Support

There are several options available for React Native DevTools integration:

1. **Native macOS App**: A 3rd party app for debugging React Query in any JS-based application:
   https://github.com/LovesWorking/rn-better-dev-tools

2. **Flipper Plugin**: A 3rd party plugin for Flipper users:
   https://github.com/bgaleotti/react-query-native-devtools

3. **Reactotron Plugin**: A 3rd party plugin for Reactotron users:
   https://github.com/hsndmr/reactotron-react-query

Online status management

React Query already supports auto refetch on reconnect in web browser.
To add this behavior in React Native, you need to use React Query's `onlineManager` as shown below:

```
import NetInfo from '@react-native-community/netinfo'
import { onlineManager } from '@tanstack/react-query'

onlineManager.setEventListener((setOnline) => {
  return NetInfo.addEventListener((state) => {
    setOnline(!!state.isConnected)
  })
})
```

or alternatively:

```
import { onlineManager } from '@tanstack/react-query'
import * as Network from 'expo-network'

onlineManager.setEventListener((setOnline) => {
  const eventSubscription = Network.addNetworkStateListener((state) => {
    setOnline(!!state.isConnected)
  })
  return eventSubscription.remove
})
```

Refetch on App focus

Instead of event listeners on `window`, React Native provides focus information through the AppState module.
Use the `AppState` "change" event to trigger an update when the app state changes to "active":

```
import { useEffect } from 'react'
import { AppState, Platform } from 'react-native'
import type { AppStateStatus } from 'react-native'
import { focusManager } from '@tanstack/react-query'

function onAppStateChange(status: AppStateStatus) {
  if (Platform.OS !== 'web') {
    focusManager.setFocused(status === 'active')
  }
}

useEffect(() => {
  const subscription = AppState.addEventListener('change', onAppStateChange)
  return () => subscription.remove()
}, [])
```

Refresh on Screen focus

This custom hook calls the provided `refetch` function when a React Native screen is focused again:

```
import React from 'react'
import { useFocusEffect } from '@react-navigation/native'

export function useRefreshOnFocus<T>(refetch: () => Promise<T>) {
  const firstTimeRef = React.useRef(true)

  useFocusEffect(
    React.useCallback(() => {
      if (firstTimeRef.current) {
        firstTimeRef.current = false
        return
      }

      refetch()
    }, [refetch]),
  )
}
```

In the above code, `refetch` is skipped the first time because `useFocusEffect` calls our callback on mount in addition to screen focus.

Disable queries on out of focus screens

To prevent certain queries from remaining "live" while a screen is out of focus, use the `subscribed` prop on `useQuery`.
This lets you control whether a query stays subscribed to updates.
Combined with React Navigation's `useIsFocused`, it allows seamless subscription management:

```
import React from 'react'
import { useIsFocused } from '@react-navigation/native'
```

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

function MyComponent() {
  const isFocused = useIsFocused()

  const { dataUpdatedAt } = useQuery({
    queryKey: ['key'],
    queryFn: () => fetch(...),
    subscribed: isFocused,
  })

  return <Text>DataUpdatedAt: {dataUpdatedAt}</Text>
}
```

When `subscribed` is false, the query unsubscribes from updates and won't trigger re-renders or fetch new data for that screen. When it becomes true again (e.g., on focus), the query re-subscribes and stays up to date.

Edit on GitHub

On this page

Important Defaults | TanStack Query React Docs

TanStack Query v5

On this page

Important Defaults

Out of the box, TanStack Query is configured with **aggressive but sane** defaults. **Sometimes these defaults can catch new users off guard or make learning/debugging difficult if they are unknown by the user.** Keep them in mind as you continue to learn and use TanStack Query:

- Query instances via `useQuery` or `useInfiniteQuery` by default **consider cached data as stale.**

  To change this behavior, you can configure your queries both globally and per-query using the `staleTime` option. Specifying a longer `staleTime` means queries will not refetch their data as often.

- A Query that has a `staleTime` set is considered **fresh** until that `staleTime` has elapsed.
  - set `staleTime` to e.g. `2 * 60 * 1000` to make sure data is read from the cache, without triggering any kinds of refetches, for 2 minutes, or until the Query is invalidated manually.
  - set `staleTime` to `Infinity` to never trigger a refetch until the Query is invalidated manually.
  - set `staleTime` to `'static'` to **never** trigger a refetch, even if the Query is invalidated manually.

- Stale queries are refetched automatically in the background when:
  - New instances of the query mount
  - The window is refocused
  - The network is reconnected

  Setting `staleTime` is the recommended way to avoid excessive refetches, but you can also customize the points in time for refetches by setting options like `refetchOnMount`, `refetchOnWindowFocus`, and `refetchOnReconnect`.

- Queries can optionally be configured with a `refetchInterval` to trigger refetches periodically, which is independent of the `staleTime` setting.

- Query results that have no more active instances of `useQuery`, `useInfiniteQuery` or query observers are labeled as "inactive" and remain in the cache in case they are used again at a later time.

- By default, "inactive" queries are garbage collected after **5 minutes.**

  To change this, you can alter the default `gcTime` for queries to something other than `1000 * 60 * 5` milliseconds.

- Queries that fail are **silently retried 3 times, with exponential backoff delay** before capturing and displaying an error to the UI.

To change this, you can alter the default `retry` and `retryDelay` options for queries to something other than `3` and the default exponential backoff function.

- Query results by default are **structurally shared to detect if data has actually changed** and if not, **the data reference remains unchanged** to better help with value stabilization with regards to `useMemo` and `useCallback`. If this concept sounds foreign, then don't worry about it! 99.9% of the time you will not need to disable this and it makes your app more performant at zero cost to you.

  Structural sharing only works with JSON-compatible values, any other value types will always be considered as changed. If you are seeing performance issues because of large responses, you can disable this feature with the `config.structuralSharing` flag. If you deal with non-JSON compatible values in your query responses and still want to detect if data has changed, you can provide your own custom function as `config.structuralSharing` to compute a value from the old and new responses, retaining references as required.

Further Reading

Have a look at the following articles from our Community Resources for further explanations of the defaults:

- Practical React Query
- React Query as a State Manager

Edit on GitHub

Queries | TanStack Query React Docs

On this page

Query Basics

A query is a declarative dependency on an asynchronous source of data that is tied to a **unique key**. A query can be used with any Promise based method (including GET and POST methods) to fetch data from a server. If your method modifies data on the server, we recommend using Mutations instead.

To subscribe to a query in your components or custom hooks, call the `useQuery` hook with at least:

- A **unique key for the query**
- A function that returns a promise that:
  - Resolves the data, or
  - Throws an error

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

function App() {
  const info = useQuery({ queryKey: ['todos'], queryFn: fetchTodoList });
}
```

The **unique key** you provide is used internally for refetching, caching, and sharing your queries throughout your application.

The query result returned by `useQuery` contains all of the information about the query that you'll need for templating and any other usage of the data:

```
const result = useQuery({ queryKey: ['todos'], queryFn: fetchTodoList });
```

The `result` object contains a few very important states you'll need to be aware of to be productive. A query can only be in one of the following states at any given moment:

- `isPending` or `status === 'pending'` - The query has no data yet
- `isError` or `status === 'error'` - The query encountered an error
- `isSuccess` or `status === 'success'` - The query was successful and data is available

Beyond those primary states, more information is available depending on the state of the query:

- `error` - If the query is in an `isError` state, the error is available via the `error` property.
- `data` - If the query is in an `isSuccess` state, the data is available via the `data` property.
- `isFetching` - In any state, if the query is fetching at any time (including background refetching) `isFetching` will be `true`.

For **most** queries, it's usually sufficient to check for the `isPending` state, then the `isError` state, then finally, assume that the data is available and render the successful state:

```
function Todos() {
  const { isPending, isError, data, error } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodoList,
  });

  if (isPending) {
    return <span>Loading...</span>;
  }

  if (isError) {
    return <span>Error: {error.message}</span>;
  }

  // We can assume by this point that `isSuccess === true`
  return (
    <ul>
      {data.map((todo) => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
}
```

Alternatively, using `status`:

```
function Todos() {
  const { status, data, error } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodoList,
  });

  if (status === 'pending') {
    return <span>Loading...</span>;
  }

  if (status === 'error') {
    return <span>Error: {error.message}</span>;
  }

  // also status === 'success', but "else" logic works, too
  return (
    <ul>
      {data.map((todo) => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
}
```

TypeScript will also narrow the type of `data` correctly if you've checked for `pending` and `error` before accessing it.

FetchStatus

In addition to the `status` field, you will also get an additional `fetchStatus` property with the following options:

- `fetchStatus === 'fetching'` - The query is currently fetching.
- `fetchStatus === 'paused'` - The query wanted to fetch, but it is paused. Read more about this in the Network Mode guide.
- `fetchStatus === 'idle'` - The query is not doing anything at the moment.

Why two different states?

Background refetches and stale-while-revalidate logic make all combinations for `status` and `fetchStatus` possible. For example:

- a query in **success** status will usually be in **idle** fetchStatus, but it could also be in **fetching** if a background refetch is happening.
- a query that mounts and has no data will usually be in **pending** status and **fetching** fetchStatus, but it could also be **paused** if there is no network connection.

So keep in mind that a query can be in **pending** state without actually fetching data. As a rule of thumb:

- The `status` gives information about the `data`: Do we have any or not?
- The `fetchStatus` gives information about the `queryFn`: Is it running or not?

Further Reading

For an alternative way of performing status checks, have a look at the Community Resources.

Edit on GitHub

Disabling/Pausing Queries | TanStack Query React Docs

On this page

Disabling/Pausing Queries

Overview

If you ever want to disable a query from automatically running, you can use the `enabled = false` option. The `enabled` option also accepts a callback that returns a boolean.

When `enabled` is `false`:

- If the query has cached data, then the query will be initialized in the `status === 'success'` or `isSuccess` state.
- If the query does not have cached data, then the query will start in the `status === 'pending'` and `fetchStatus === 'idle'` state.
- The query will not automatically fetch on mount.
- The query will not automatically refetch in the background.
- The query will ignore query client `invalidateQueries` and `refetchQueries` calls that would normally result in the query refetching.
- `refetch` returned from `useQuery` can be used to manually trigger the query to fetch. However, it will not work with `skipToken`.

  TypeScript users may prefer to use skipToken as an alternative to `enabled = false`.

Example

```
function Todos() {
  const { isLoading, isError, data, error, refetch, isFetching } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodoList,
    enabled: false,
  })

  return (
    <div>
      <button onClick={() => refetch()}>Fetch Todos</button>

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

```
        </>
      ) : isError ? (
        <span>Error: {error.message}</span>
      ) : isLoading ? (
        <span>Loading...</span>
      ) : (
        <span>Not ready ...</span>
      )}

      <div>{isFetching ? 'Fetching...' : null}</div>
    </div>
  )
}
```

Note: Permanently disabling a query opts out of features like background refetches, and isn't the idiomatic approach. It shifts from a declarative dependency approach to an imperative mode (e.g., fetch only on button click). It also doesn't support passing parameters to `refetch` — for lazy fetches, use `enabled` carefully or the `skipToken` approach.

Lazy Queries

`enabled` can be used to control if a query runs initially, acting as a lazy loader. For example, in a filter form, only trigger fetch once the user provides input:

```
function Todos() {
  const [filter, setFilter] = React.useState('')

  const { data } = useQuery({
    queryKey: ['todos', filter],
    queryFn: () => fetchTodos(filter),
    // Disabled as long as filter is empty
    enabled: !!filter,
  })

  return (
    <div>
      {/* Applying the filter enables the query */}
      <FiltersForm onApply={setFilter} />
      {data && <TodosTable data={data} />}
    </div>
  )
}
```

isLoading (Previously: `isInitialLoading`)

Lazy queries start with `status: 'pending'`, indicating no data yet. Since the query isn't enabled, it doesn't fetch, so you can't rely on `isLoading` for a spinner. Instead, use the derived `isLoading` flag:

```
// `isLoading` becomes true only if the query is fetching for the first time
```

This flag is computed via:

```
isPending && isFetching
```

Typesafe disabling of queries using `skipToken`

In TypeScript, you can disable a query based on conditions while keeping type safety:

> **Important:** `refetch` from `useQuery` does not work with `skipToken`.

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

function Todos() {
  const [filter, setFilter] = React.useState<string | undefined>()

  const { data } = useQuery({
    queryKey: ['todos', filter],
    // Query function is skipped if filter is undefined or empty
    queryFn: filter ? () => fetchTodos(filter) : skipToken,
  })

  return (
    <div>
      {/* Apply filter to enable the query */}
      <FiltersForm onApply={setFilter} />
      {data && <TodosTable data={data} />}
    </div>
  )
}
```

Edit on GitHub

Edit this page on GitHub

Query Retries | TanStack Query React Docs

On this page

Query Retries

When a *useQuery* query fails (the query function throws an error), TanStack Query will automatically retry the query if that query's request has not reached the max number of consecutive retries (defaults to **3**) or a function is provided to determine if a retry is allowed.

You can configure retries both on a global level and an individual query level.

- Setting **retry = false** will disable retries.
- Setting **retry = 6** will retry failing requests 6 times before showing the final error thrown by the function.
- Setting **retry = true** will infinitely retry failing requests.
- Setting **retry = (failureCount, error) => ...** allows for custom logic based on why the request failed.

  On the server, retries default to **0** to make server rendering as fast as possible.

Example

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

// Make a specific query retry a certain number of times
const result = useQuery({
  queryKey: ['todos', 1],
  queryFn: fetchTodoListPage,
  retry: 10, // Will retry failed requests 10 times before displaying an error
})
```

> **Note:** Contents of the `error` property will be part of `failureReason` response property of `useQuery` until the last retry attempt. So in above example any error contents will be part of `failureReason` property for first 9 retry attempts (overall 10 attempts) and finally they will be part of `error` after the last attempt if error persists after all retry attempts.

Retry Delay

Back to top

By default, retries in TanStack Query do not happen immediately after a request fails. As is standard, a back-off delay is gradually applied to each retry attempt.

The default `retryDelay` is set to double (starting at `1000` ms) with each attempt, but not exceeding 30 seconds:

```
// Configure for all queries
import {
```

```
  QueryCache,
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retryDelay: (attemptIndex) => Math.min(1000 * 2 ** attemptIndex, 30000),
    },
  },
})

function App() {
  return <QueryClientProvider client={queryClient}>...</QueryClientProvider>
}
```

Though it is not recommended, you can override the `retryDelay` function or integer in both the Provider and individual query options. If set to an integer, the delay will always be the same amount of time:

```
const result = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
  retryDelay: 1000, // Always wait 1000ms to retry, regardless of retries
})
```

Edit on GitHub

Paginated / Lagged Queries

On this page

Better Paginated Queries with `placeholderData`

Rendering paginated data is a very common UI pattern and in TanStack Query, it "just works" by including the page information in the query key:

```
const result = useQuery({
  queryKey: ['projects', page],
  queryFn: fetchProjects,
})
```

However, if you run this simple example, you might notice something strange:

The UI jumps in and out of the `success` and `pending` states because each new page is treated like a brand new query.

This experience is not optimal and unfortunately is how many tools today insist on working. But not TanStack Query! As you may have guessed, TanStack Query comes with an awesome feature called `placeholderData` that allows us to get around this.

Better Paginated Queries with `placeholderData`

Consider the following example where we would ideally want to increment a `pageIndex` (or cursor) for a query. If we were to use `useQuery`, **it would still technically work fine**, but the UI would jump in and out of the `success` and `pending` states as different queries are created and destroyed for each page or cursor. By setting `placeholderData` to `(previousData) => previousData` or `keepPreviousData` function exported from TanStack Query, we get a few new things:

- **The data from the last successful fetch is available while new data is being requested, even though the query key has changed.**
- When the new data arrives, the previous *data* is seamlessly swapped to show the new data.
- *isPlaceholderData* is made available to know what data the query is currently providing you.

```
import { keepPreviousData, useQuery } from '@tanstack/react-query'
import React from 'react'

function Todos() {
  const [page, setPage] = React.useState(0)

  const fetchProjects = (page = 0) =>
    fetch('/api/projects?page=' + page).then((res) => res.json())
```

```
  const { isPending, isError, error, data, isFetching, isPlaceholderData } =
    useQuery({
      queryKey: ['projects', page],
      queryFn: () => fetchProjects(page),
      placeholderData: keepPreviousData,
    })

  return (
    <div>
      {isPending ? (
        <div>Loading...</div>
      ) : isError ? (
        <div>Error: {error.message}</div>
      ) : (
        <div>
          {data.projects.map((project) => (
            <p key={project.id}>{project.name}</p>
          ))}
        </div>
      )}
      <span>Current Page: {page + 1}</span>
      <button
        onClick={() => setPage((old) => Math.max(old - 1, 0))}
        disabled={page === 0}
      >
        Previous Page
      </button>
      <button
        onClick={() => {
          if (!isPlaceholderData && data.hasMore) {
            setPage((old) => old + 1)
          }
        }}
        // Disable the Next Page button until we know a next page is available
        disabled={isPlaceholderData || !data?.hasMore}
      >
        Next Page
      </button>
      {isFetching ? <span> Loading...</span> : null}
    </div>
  )
}
```

While not as common, the `placeholderData` option also works flawlessly with the `useInfiniteQuery` hook, so you can seamlessly allow your users to continue to see cached data while infinite query keys change over time.

Lagging Infinite Query results with `placeholderData`

While not as common, the `placeholderData` option also works flawlessly with the `useInfiniteQuery` hook, so you can seamlessly allow your users to continue to see cached data while infinite query keys change over time.

Edit on GitHub

Infinite Queries

Rendering lists with "load more" or "infinite scroll"

Supporting lists that add data onto existing sets or support infinite scrolling is common UI design. TanStack Query offers `useInfiniteQuery` for such patterns.

Key differences when using `useInfiniteQuery`:

- `data` is an object with:
  - `data.pages`: array of fetched pages
  - `data.pageParams`: array of query parameters used per page
- Functions:
  - `fetchNextPage` and `fetchPreviousPage` (with `fetchNextPage` required)
- Options:
  - `initialPageParam` (required) for initial page parameter
- Pagination:
  - `getNextPageParam` and `getPreviousPageParam` to determine if more data exists and how to fetch it, supplied as an argument to query function
  - Booleans `hasNextPage` and `hasPreviousPage` indicate availability of more data
  - State booleans `isFetchingNextPage` and `isFetchingPreviousPage` distinguish background fetches

  Note: Options `initialData` and `placeholderData` must conform to an object with `data.pages` and `data.pageParams`

Example

Suppose your API returns pages of `projects`, 3 at a time, with a cursor:

```
fetch('/api/projects?cursor=0')
// { data: [...], nextCursor: 3}
fetch('/api/projects?cursor=3')
// { data: [...], nextCursor: 6}
fetch('/api/projects?cursor=6')
// { data: [...], nextCursor: 9}
fetch('/api/projects?cursor=9')
// { data: [...] }
```

You can implement a "Load More" button:

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

function Projects() {
  const fetchProjects = async ({ pageParam }) => {
    const res = await fetch('/api/projects?cursor=' + pageParam);
    return res.json();
  }

  const {
    data,
```

```
    error,
    fetchNextPage,
    hasNextPage,
    isFetching,
    isFetchingNextPage,
    status,
  } = useInfiniteQuery({
    queryKey: ['projects'],
    queryFn: fetchProjects,
    initialPageParam: 0,
    getNextPageParam: (lastPage) => lastPage.nextCursor,
  });

  return status === 'pending' ? (
    <p>Loading...</p>
  ) : status === 'error' ? (
    <p>Error: {error.message}</p>
  ) : (
    <>
      {data.pages.map((group, i) => (
        <React.Fragment key={i}>
          {group.data.map((project) => (
            <p key={project.id}>{project.name}</p>
          ))}
        </React.Fragment>
      ))}
      <div>
        <button
          onClick={() => fetchNextPage()}
          disabled={!hasNextPage || isFetching}
        >
          {isFetchingNextPage
            ? 'Loading more...'
            : hasNextPage
            ? 'Load More'
            : 'Nothing more to load'}
        </button>
      </div>
      <div>{isFetching && !isFetchingNextPage ? 'Fetching...' : null}</div>
    </>
  );
}
```

Handling refetching of infinite queries

When an infinite query becomes `stale`, each page is refetched sequentially from the first page to prevent stale cursors and duplicates.

If all pages are removed from cache, the infinite pagination restarts.

Implementing bi-directional infinite list

Use `getPreviousPageParam`, `fetchPreviousPage`, `hasPreviousPage`, and `isFetchingPreviousPage`:

```
useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: fetchProjects,
  initialPageParam: 0,
  getNextPageParam: (lastPage) => lastPage.nextCursor,
  getPreviousPageParam: (firstPage) => firstPage.prevCursor,
});
```

Showing pages in reversed order

Utilize `select` to reverse the data:

```
useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: fetchProjects,
  select: (data) => ({
    pages: [...data.pages].reverse(),
    pageParams: [...data.pageParams].reverse(),
  }),
});
```

Manually updating infinite query data

- Remove the first page:

```
queryClient.setQueryData(['projects'], (data) => ({
  pages: data.pages.slice(1),
  pageParams: data.pageParams.slice(1),
}));
```

- Remove a specific item from a page:

```
const newPagesArray =
  oldPagesArray?.pages.map((page) =>
    page.filter((val) => val.id !== updatedId),
  ) ?? [];

queryClient.setQueryData(['projects'], (data) => ({
  pages: newPagesArray,
  pageParams: data.pageParams,
}));
```

- Keep only the first page:

```
queryClient.setQueryData(['projects'], (data) => ({
  pages: data.pages.slice(0, 1),
  pageParams: data.pageParams.slice(0, 1),
}));
```

  Always maintain the data structure of `pages` and `pageParams`.

Limiting the number of pages

You can set `maxPages` (supported in certain configurations) to restrict stored pages, improving performance. When fetching new pages, only a limited number are kept:

```
useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: fetchProjects,
  initialPageParam: 0,
  getNextPageParam: (lastPage) => lastPage.nextCursor,
  getPreviousPageParam: (firstPage) => firstPage.prevCursor,
  maxPages: 3,
});
```

Handling APIs without cursors

Use `pageParam` to track pagination:

```
return useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: fetchProjects,
  initialPageParam: 0,
  getNextPageParam: (lastPage, allPages, lastPageParam) => {
    if (lastPage.length === 0) return undefined;
    return lastPageParam + 1;
  },
  getPreviousPageParam: (firstPage, allPages, firstPageParam) => {
    if (firstPageParam <= 1) return undefined;
    return firstPageParam - 1;
  },
});
```

Additional links

Edit on GitHub

On this page

Initial Query Data | TanStack Query React Docs

On this page

There are many ways to supply initial data for a query to the cache before you need it:

- **Declaratively:**
  - Provide `initialData` to a query to prepopulate its cache if empty
- **Imperatively:**
  - Prefetch the data using `queryClient.prefetchQuery`
  - Manually place the data into the cache using `queryClient.setQueryData`

Using `initialData` to prepopulate a query

There may be times when you already have the initial data for a query available in your app and can simply provide it directly to your query. If and when this is the case, you can use the `config.initialData` option to set the initial data for a query and skip the initial loading state!

> **IMPORTANT:** `initialData` is persisted to the cache, so it is not recommended to provide placeholder, partial, or incomplete data to this option. Use `placeholderData` instead.

```
const result = useQuery({
  queryKey: ['todos'],
  queryFn: () => fetch('/todos'),
  initialData: initialTodos,
});
```

`staleTime` and `initialDataUpdatedAt`

By default, `initialData` is treated as totally fresh, as if it were just fetched. This also means that it will affect how it is interpreted by the `staleTime` option.

- If you configure your query observer with `initialData` and no `staleTime` (default `staleTime: 0`), the query will immediately refetch when it mounts:

```
// Will show initialTodos immediately, but also immediately refetch todos after mount
const result = useQuery({
  queryKey: ['todos'],
  queryFn: () => fetch('/todos'),
  initialData: initialTodos,
});
```

- If you set a `staleTime` of `1000` ms, the data will be considered fresh for that time:

```
// Show initialTodos immediately, but won't refetch until another interaction after 10
const result = useQuery({
  queryKey: ['todos'],
  queryFn: () => fetch('/todos'),
  initialData: initialTodos,
  staleTime: 1000,
});
```

- For data that's not totally fresh, you can use `initialDataUpdatedAt`, passing a timestamp in milliseconds (`Date.now()`):

```
// Show initialTodos immediately, but won't refetch until after the specified timestam
const result = useQuery({
  queryKey: ['todos'],
  queryFn: () => fetch('/todos'),
  initialData: initialTodos,
  staleTime: 60 * 1000, // 1 minute
  initialDataUpdatedAt: initialTodosUpdatedTimestamp, // e.g., 1608412420052
});
```

This allows the query to decide whether to refetch based on how old the initial data is.

> **Note:** To treat your data as prefetched, consider using `prefetchQuery` for cache population, giving more control over `staleTime`.

Initial Data Function

If retrieving initial data is expensive or complex, pass a function as `initialData`. This function runs only once during query initialization:

```
const result = useQuery({
  queryKey: ['todos'],
  queryFn: () => fetch('/todos'),
  initialData: () => getExpensiveTodos(),
});
```

Initial Data from Cache

You can provide initial data based on other cached queries — for example, extracting a specific todo item from a list:

```
const result = useQuery({
  queryKey: ['todo', todoId],
  queryFn: () => fetch('/todos'),
  initialData: () => {
    // Use a todo from 'todos' query as initial data for this todo
```

```
    return queryClient.getQueryData(['todos'])?.find(d => d.id === todoId);
  },
});
```

Initial Data from the cache with `initialDataUpdatedAt`

You can improve cache consistency by passing the cache's `dataUpdatedAt` timestamp:

```
const result = useQuery({
  queryKey: ['todos', todoId],
  queryFn: () => fetch(`/todos/${todoId}`),
  initialData: () => queryClient.getQueryData(['todos'])?.find(d => d.id === todoId),
  initialDataUpdatedAt: () => queryClient.getQueryState(['todos'])?.dataUpdatedAt,
});
```

Conditional Initial Data from Cache

If the cached data is outdated, skip using it and fetch fresh data. Use `queryClient.getQueryState()` to decide:

```
const result = useQuery({
  queryKey: ['todo', todoId],
  queryFn: () => fetch(`/todos/${todoId}`),
  initialData: () => {
    const state = queryClient.getQueryState(['todos']);
    if (state && Date.now() - state.dataUpdatedAt <= 10 * 1000) {
      // Use cache if data is less than 10 seconds old
      return state.data.find(d => d.id === todoId);
    }
    // Else, fetch fresh data
  },
});
```

Further reading

For a comparison between **Initial Data** and **Placeholder Data**, see the Community Resources.

Edit on GitHub

Edit this page on GitHub

Placeholder Query Data

What is placeholder data?

Placeholder data allows a query to behave as if it already has data, similar to the *initialData* option, but **the data is not persisted to the cache**. This comes in handy for situations where you have enough partial (or fake) data to render the query successfully while the actual data is fetched in the background.

> Example: An individual blog post query could pull "preview" data from a parent list of blog posts that only include title and a small snippet of the post body. You would not want to persist this partial data to the query result of the individual query, but it is useful for showing the content layout as quickly as possible while the actual query finishes to fetch the entire object.

There are a few ways to supply placeholder data for a query to the cache before you need it:

- **Declaratively:**
  - Provide `placeholderData` to a query to prepopulate its cache if empty.
- **Imperatively:**
  - Prefetch or fetch the data using `queryClient` and the `placeholderData` option.

When we use `placeholderData`, our Query will not be in a `pending` state - it will start out as being in `success` state, because we have `data` to display - even if that data is just "placeholder" data. To distinguish it from "real" data, we will also have the `isPlaceholderData` flag set to `true` on the Query result.

Placeholder Data as a Value

```
function Todos() {
  const result = useQuery({
    queryKey: ['todos'],
    queryFn: () => fetch('/todos'),
    placeholderData: placeholderTodos,
  })
}
```

Placeholder Data Memoization

If the process for accessing a query's placeholder data is intensive or just not something you want to perform on every render, you can memoize the value:

```
function Todos() {
  const placeholderData = useMemo(() => generateFakeTodos(), [])
  const result = useQuery({
    queryKey: ['todos'],
    queryFn: () => fetch('/todos'),
    placeholderData,
  })
}
```

Placeholder Data as a Function

`placeholderData` can also be a function, where you can get access to the data and Query meta information of a "previous" successful Query. This is useful for situations where you want to use the data from one query as the placeholder data for another query. When the QueryKey changes, e.g., from `['todos', 1]` to `['todos', 2]`, we can keep displaying "old" data instead of having to show a loading spinner while data is *transitioning* from one Query to the next. For more information, see Paginated Queries.

```
const result = useQuery({
  queryKey: ['todos', id],
  queryFn: () => fetch(`/todos/${id}`),
  placeholderData: (previousData, previousQuery) => previousData,
});
```

Placeholder Data from Cache

In some circumstances, you may be able to provide the placeholder data for a query from the cached result of another. A good example of this would be searching the cached data from a blog post list query for a preview version of the post, then using that as the placeholder data for your individual post query:

```
function Todo({ blogPostId }) {
  const queryClient = useQueryClient()
  const result = useQuery({
    queryKey: ['blogPost', blogPostId],
    queryFn: () => fetch(`/blogPosts/${blogPostId}`),
    placeholderData: () => {
      // Use the smaller/preview version of the blogPost from the 'blogPosts'
      // query as the placeholder data for this blogPost query
      return queryClient
        .getQueryData(['blogPosts'])
        ?.find((d) => d.id === blogPostId)
    },
  })
}
```

Further reading

For a comparison between *Placeholder Data* and *Initial Data*, have a look at the Community Resources.

Edit on GitHub

On this page

Mutations | TanStack Query React Docs

Mutations

Unlike queries, mutations are typically used to create/update/delete data or perform server side-effects. For this purpose, TanStack Query exports a **useMutation** hook.

Example of a mutation that adds a new todo to the server:

```
function App() {
  const mutation = useMutation({
    mutationFn: (newTodo) => {
      return axios.post('/todos', newTodo)
    },
  })

  return (
    <div>
      {mutation.isPending ? (
        'Adding todo...'
      ) : (
        <>
          {mutation.isError ? (
            <div>An error occurred: {mutation.error.message}</div>
          ) : null}

          {mutation.isSuccess ? <div>Todo added!</div> : null}

          <button
            onClick={() => {
              mutation.mutate({ id: new Date(), title: 'Do Laundry' })
            }}
          >
            Create Todo
          </button>
        </>
      )}
    </div>
  )
}
```

Mutation states

A mutation can only be in one of the following states at any given moment:

- **isIdle** or *status* === *'idle'* — The mutation is currently idle or in a fresh/reset state
- **isPending** or *status* === *'pending'* — The mutation is currently running
- **isError** or *status* === *'error'* — The mutation encountered an error
- **isSuccess** or *status* === *'success'* — The mutation was successful and mutation data is available

Additional information based on state:

- **error** — If in an *error* state, the error information is available via the `error` property.
- **data** — If in a *success* state, the resulting data is available via the `data` property.

Passing variables to mutations

In the example above, variables are passed by calling the `mutate` function with a **single variable or object**.

Powerful mutations with `onSuccess` and query invalidation

When used with the `onSuccess` option and `invalidateQueries` / `setQueryData`, mutations become a very powerful tool.

IMPORTANT: The `mutate` function is asynchronous and cannot be used directly in event callbacks in *React 16 and earlier*. Wrap it in another function if needed to access the event.

Example: Wrapping mutate for legacy React

```
// This will not work in React 16 and earlier
const CreateTodo = () => {
  const mutation = useMutation({
    mutationFn: (event) => {
      event.preventDefault()
      return fetch('/api', new FormData(event.target))
    },
  })

  return <form onSubmit={mutation.mutate}>...</form>
}

// This will work
const CreateTodo = () => {
  const mutation = useMutation({
    mutationFn: (formData) => {
      return fetch('/api', formData)
    },
  })

  const onSubmit = (event) => {
    event.preventDefault()
    mutation.mutate(new FormData(event.target))
  }

  return <form onSubmit={onSubmit}>...</form>
}
```

Resetting Mutation State

You might need to clear the `error` or `data` of a mutation. Use the `reset` function:

```
const CreateTodo = () => {
  const [title, setTitle] = useState('')
```

```
  const mutation = useMutation({ mutationFn: createTodo })

  const onCreateTodo = (e) => {
    e.preventDefault()
    mutation.mutate({ title })
  }

  return (
    <form onSubmit={onCreateTodo}>
      {mutation.error && (
        <h5 onClick={() => mutation.reset()}>{mutation.error}</h5>
      )}
      <input
        type="text"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
      />
      <br />
      <button type="submit">Create Todo</button>
    </form>
  )
}
```

Mutation Side Effects

`useMutation` accepts options for handling side effects at different lifecycle stages:

```
useMutation({
  mutationFn: addTodo,
  onMutate: (variables) => {
    // Before mutation
    return { id: 1 } // Return context for rollback
  },
  onError: (error, variables, context) => {
    console.log(`Rolling back optimistic update with id ${context.id}`)
  },
  onSuccess: (data, variables, context) => {
    // Success callback
  },
  onSettled: (data, error, variables, context) => {
    // Called after success or error
  },
})
```

Returning promises in callbacks

Callbacks can be asynchronous:

```
useMutation({
  mutationFn: addTodo,
  onSuccess: async () => {
    console.log("I'm first!")
  },
  onSettled: async () => {
```

```
    console.log("I'm second!")
  },
})
```

Triggering extra callbacks on `mutate`

You can pass additional lifecycle callbacks when calling `mutate`. They won't run if the component unmounts before completion.

```
mutate(todo, {
  onSuccess: (data, variables, context) => {
    // Second callback
  },
  onError: (error, variables, context) => {
    // Second callback
  },
  onSettled: (data, error, variables, context) => {
    // Second callback
  },
})
```

Consecutive mutations

Callbacks like `onSuccess`, `onError`, and `onSettled` behave differently for consecutive mutations. When passed to `mutate`, they fire **once** and only if the component is still mounted, ensuring only the last mutation triggers them for the same scope.

Note: The order of fulfillment for asynchronous `mutationFn` may differ from call order.

```
useMutation({
  mutationFn: addTodo,
  onSuccess: (data, variables, context) => {
    // Will be called 3 times
  },
})

const todos = ['Todo 1', 'Todo 2', 'Todo 3']
todos.forEach((todo) => {
  mutate(todo, {
    onSuccess: (data, variables, context) => {
      // Executes only once, for the last mutation
    },
  })
})
```

Promises

Use `mutateAsync()` to get a promise that resolves on success or throws on error, enabling composition.

```
const mutation = useMutation({ mutationFn: addTodo })

try {
  const todo = await mutation.mutateAsync(todo)
```

```
  console.log(todo)
} catch (error) {
  console.error(error)
} finally {
  console.log('done')
}
```

Retry

By default, mutations do not retry on error. To enable retries:

```
const mutation = useMutation({
  mutationFn: addTodo,
  retry: 3,
})
```

Mutations will retry if, for example, the device is offline, upon reconnect.

Persisting mutations

Mutations can be persisted and resumed later using hydration functions.

```
const queryClient = new QueryClient()

// Define mutation default
queryClient.setMutationDefaults(['addTodo'], {
  mutationFn: addTodo,
  onMutate: async (variables) => {
    await queryClient.cancelQueries({ queryKey: ['todos'] })

    const optimisticTodo = { id: uuid(), title: variables.title }
    queryClient.setQueryData(['todos'], (old) => [...old, optimisticTodo])

    return { optimisticTodo }
  },
  onSuccess: (result, variables, context) => {
    queryClient.setQueryData(['todos'], (old) =>
      old.map((todo) =>
        todo.id === context.optimisticTodo.id ? result : todo
      )
    )
  },
  onError: (error, variables, context) => {
    queryClient.setQueryData(['todos'], (old) =>
      old.filter((todo) => todo.id !== context.optimisticTodo.id)
    )
  },
  retry: 3,
})

// In component
const mutation = useMutation({ mutationKey: ['addTodo'] })
mutation.mutate({ title: 'title' })
```

```
// Hydration
const state = dehydrate(queryClient)

// After app reload
hydrate(queryClient, state)
queryClient.resumePausedMutations()
```

Persisting Offline mutations

Persisted mutations can't be resumed unless a default mutation function is provided, due to functions not being serializable.

```
const persister = createSyncStoragePersister({
  storage: window.localStorage,
})
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      gcTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})

// Provide default mutationFn for paused mutations to resume after reload
queryClient.setMutationDefaults(['todos'], {
  mutationFn: ({ id, data }) => {
    return api.updateTodo(id, data)
  },
})

// Hydration example
export default function App() {
  return (
    <PersistQueryClientProvider
      client={queryClient}
      persistOptions={{ persister }}
      onSuccess={() => {
        queryClient.resumePausedMutations()
      }}
    >
      <RestOfTheApp />
    </PersistQueryClientProvider>
  )
}
```

Mutation Scopes

By default, all mutations run in parallel. Assigning a **scope** with an `id` allows mutations to run in series per scope.

```
const mutation = useMutation({
  mutationFn: addTodo,
```

```
  scope: {
    id: 'todo',
  },
})
```

Mutations with the same `scope.id` will queue and run serially, respecting the `isPaused` state until completed.

Further reading

For more on mutations, see #12: Mastering Mutations in React Query in the Community Resources.

Edit on GitHub

Query Invalidation | TanStack Query React Docs

TanStack Query v5

On this page

Query Invalidation

Waiting for queries to become stale before they are fetched again doesn't always work, especially when you know for a fact that a query's data is out of date because of something the user has done. For that purpose, the `QueryClient` has an `invalidateQueries` method that lets you intelligently mark queries as stale and potentially refetch them too!

```
// Invalidate every query in the cache
queryClient.invalidateQueries()

// Invalidate every query with a key that starts with `todos`
queryClient.invalidateQueries({ queryKey: ['todos'] })
```

> **Note:** Where other libraries that use normalized caches would attempt to update local queries with the new data either imperatively or via schema inference, TanStack Query gives you the tools to avoid the manual labor that comes with maintaining normalized caches and instead prescribes **targeted invalidation, background-refetching and ultimately atomic updates**.

When a query is invalidated with `invalidateQueries`, two things happen:

- It is marked as *stale*. This stale state overrides any `staleTime` configurations being used in `useQuery` or related hooks.
- If the query is currently being rendered via `useQuery` or related hooks, it will also be refetched in the background.

Query Matching with `invalidateQueries`

When using APIs like `invalidateQueries` and `removeQueries` (and others that support partial query matching), you can match multiple queries by their prefix, or get really specific and match an exact query. For info on query filters, see Query Filters.

In this example, the `todos` prefix invalidates any queries starting with `todos`:

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

// Get QueryClient from the context
const queryClient = useQueryClient()

queryClient.invalidateQueries({ queryKey: ['todos'] })

// Both queries below will be invalidated
```

```
const todoListQuery = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
})
const todoListQuery2 = useQuery({
  queryKey: ['todos', { page: 1 }],
  queryFn: fetchTodoList,
})
```

You can invalidate queries with specific variables by passing a detailed query key:

```
queryClient.invalidateQueries({
  queryKey: ['todos', { type: 'done' }],
})

// This query will be invalidated
const doneTodosQuery = useQuery({
  queryKey: ['todos', { type: 'done' }],
  queryFn: fetchTodoList,
})

// This query will NOT be invalidated
const allTodosQuery = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
})
```

For more granular control, use the `exact: true` option to invalidate only queries that exactly match:

```
queryClient.invalidateQueries({
  queryKey: ['todos'],
  exact: true,
})

// This query will be invalidated
const exactTodosQuery = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
})

// This one won't
const partialTodosQuery = useQuery({
  queryKey: ['todos', { type: 'done' }],
  queryFn: fetchTodoList,
})
```

You can also pass a predicate function for custom logic:

```
queryClient.invalidateQueries({
  predicate: (query) =>
    query.queryKey[0] === 'todos' && query.queryKey[1]?.version >= 10,
})

// Queries invalidated include:
const version20Query = useQuery({
  queryKey: ['todos', { version: 20 }],
  queryFn: fetchTodoList,
```

```
})
const version10Query = useQuery({
  queryKey: ['todos', { version: 10 }],
  queryFn: fetchTodoList,
})

// Queries NOT invalidated:
const version5Query = useQuery({
  queryKey: ['todos', { version: 5 }],
  queryFn: fetchTodoList,
})
```

Edit on GitHub

Edit this page on GitHub

Related Guides

- Mutations
- Invalidation from Mutations

Our Partners

More Resources

TanStack Table

Supercharge your tables or build a data grid for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, and Lit while retaining 100% control over markup and styles.

TanStack DB

TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥

Additional Links

Note: These are the main points and code snippets, ensuring a clean Markdown conversion of the lengthy HTML content.

Invalidations from Mutations

Invalidating queries is only half the battle. Knowing **when** to invalidate them is the other half. Usually, when a mutation in your app succeeds, it's **VERY** likely that there are related queries in your application that need to be invalidated and possibly refetched to account for the new changes from your mutation.

For example, assume we have a mutation to post a new todo:

```
const mutation = useMutation({ mutationFn: postTodo })
```

When a successful `postTodo` mutation happens, we likely want all `todos` queries to get invalidated and possibly refetched to show the new todo item. To do this, you can use `useMutation`'s `onSuccess` options and the client's `invalidateQueries` function:

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

const queryClient = useQueryClient()

// When this mutation succeeds, invalidate any queries with the `todos` or `reminders`
const mutation = useMutation({
  mutationFn: addTodo,
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ['todos'] })
    queryClient.invalidateQueries({ queryKey: ['reminders'] })
  },
})
```

You can wire up your invalidations to happen using any of the callbacks available in the useMutation hook.

Edit on GitHub

Updates from Mutation Responses

On this page

Introduction

When dealing with mutations that **update** objects on the server, it's common for the new object to be automatically returned in the response of the mutation. Instead of refetching any queries for that item and wasting a network call for data we already have, we can take advantage of the object returned by the mutation function and update the existing query with the new data immediately using the Query Client's `setQueryData` method:

Example: Updating query data after mutation

```
const queryClient = useQueryClient()

const mutation = useMutation({
  mutationFn: editTodo,
  onSuccess: (data) => {
    queryClient.setQueryData(['todo', { id: 5 }], data)
  },
})

mutation.mutate({
  id: 5,
  name: 'Do the laundry',
})

// This query will be updated with the response from the
// successful mutation
const { status, data, error } = useQuery({
  queryKey: ['todo', { id: 5 }],
  queryFn: fetchTodoById,
})
```

Reusable mutation with `onSuccess`

You may want to tie the `onSuccess` logic into a reusable mutation, for that you can create a custom hook like this:

```
const useMutateTodo = () => {
  const queryClient = useQueryClient()

  return useMutation({
    mutationFn: editTodo,
    // The second argument in onSuccess is the variables object the `mutate` function
    onSuccess: (data, variables) => {
      queryClient.setQueryData(['todo', { id: variables.id }], data)
    },
```

```
  })
}
```

Immutability

Updates via `setQueryData` must be performed in an *immutable* way. **DO NOT** attempt to write directly to the cache by mutating data (that you retrieved from the cache) in place. It might work at first but can lead to subtle bugs along the way.

Correct way to update data immutably

```
queryClient.setQueryData(['posts', { id }], (oldData) => {
  if (oldData) {
    // ❌ do not try this
    oldData.title = 'my new post title'
  }
  return oldData
})

queryClient.setQueryData(
  ['posts', { id }],
  // ✅ this is the correct way
  (oldData) =>
    oldData
      ? {
          ...oldData,
          title: 'my new post title',
        }
      : oldData,
)
```

In summary:

- Always perform immutable updates.
- Never mutate the cached data directly.
- Use the spread operator or other immutable techniques.

Edit on GitHub

Query Keys

On this page

Query Keys

Overview

At its core, TanStack Query manages query caching for you based on query keys. Query keys have to be an Array at the top level, and can be as simple as an Array with a single string, or as complex as an array of many strings and nested objects. As long as the query key is serializable using `JSON.stringify`, and **unique to the query's data**, you can use it!

Simple Query Keys

The simplest form of a key is an array with constant values. This format is useful for:

- Generic List/Index resources
- Non-hierarchical resources

```
// A list of todos
useQuery({ queryKey: ['todos'], ... })

// Something else, whatever!
useQuery({ queryKey: ['something', 'special'], ... })
```

Array Keys with variables

When a query needs more information to uniquely describe its data, you can use an array with a string and any number of serializable objects to describe it. This is useful for:

- Hierarchical or nested resources
  It's common to pass an ID, index, or other primitive to uniquely identify the item

- Queries with additional parameters
  It's common to pass an object of additional options

```
// An individual todo
useQuery({ queryKey: ['todo', 5], ... })

// An individual todo in a "preview" format
useQuery({ queryKey: ['todo', 5, { preview: true }], ... })
```

```
// A list of todos that are "done"
useQuery({ queryKey: ['todos', { type: 'done' }], ... })
```

Query Keys are hashed deterministically!

This means that no matter the order of keys in objects, all of the following queries are considered equal:

```
useQuery({ queryKey: ['todos', { status, page }], ... })
useQuery({ queryKey: ['todos', { page, status }], ... })
useQuery({ queryKey: ['todos', { page, status, other: undefined }], ... })
```

The following query keys, however, are not equal. Array item order matters:

```
useQuery({ queryKey: ['todos', status, page], ... })
useQuery({ queryKey: ['todos', page, status], ... })
useQuery({ queryKey: ['todos', undefined, page, status], ... })
```

If your query function depends on a variable, include it in your query key

Since query keys uniquely describe the data they are fetching, they should include any variables you use in your query function that *change*. For example:

```
function Todos({ todoId }) {
  const result = useQuery({
    queryKey: ['todos', todoId],
    queryFn: () => fetchTodoById(todoId),
  })
}
```

Note that query keys act as dependencies for your query functions. Adding dependent variables to your query key will ensure that queries are cached independently, and that any time a variable changes, *queries will be refetched automatically* (depending on your `staleTime` settings). See the exhaustive-deps section for more information and examples.

Further reading

For tips on organizing Query Keys in larger applications, have a look at Effective React Query Keys and check the Query Key Factory Package from the Community Resources.

Edit on GitHub

On this page

Optimistic Updates | TanStack Query React Docs

On this page

Via the UI

This is the simpler variant, as it doesn't interact with the cache directly.

```
const addTodoMutation = useMutation({
  mutationFn: (newTodo: string) => axios.post('/api/data', { text: newTodo }),
  // make sure to _return_ the Promise from the query invalidation
  // so that the mutation stays in `pending` state until the refetch is finished
  onSettled: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
});

const { isPending, submittedAt, variables, mutate, isError } = addTodoMutation;
```

You will then have access to `addTodoMutation.variables`, which contain the added todo. In your UI list, where the query is rendered, you can append another item while the mutation `isPending`:

```
<ul>
  {todoQuery.items.map((todo) => (
    <li key={todo.id}>{todo.text}</li>
  ))}
  {isPending && <li style={{ opacity: 0.5 }}>{variables}</li>}
</ul>
```

We're rendering a temporary item with a different opacity as long as the mutation is pending. Once it completes, the item will automatically no longer be rendered. If the refetch succeeded, the item appears in the list as a "normal" item.

If the mutation errors, the item will disappear, but you can also keep showing it by checking the `isError` state:

```
{
  isError && (
    <li style={{ color: 'red' }}>
      {variables}
      <button onClick={() => mutate(variables)}>Retry</button>
    </li>
  );
}
```

If the mutation and the query don't live in the same component

This approach works well if they are in the same component. You also get access to all mutations elsewhere via the `useMutationState` hook, especially when combined with a `mutationKey`:

```
// somewhere in your app
const { mutate } = useMutation({
  mutationFn: (newTodo: string) => axios.post('/api/data', { text: newTodo }),
  onSettled: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
  mutationKey: ['addTodo'],
});

// access variables somewhere else
const variables = useMutationState<string>({
  filters: { mutationKey: ['addTodo'], status: 'pending' },
  select: (mutation) => mutation.state.variables,
});
```

`variables` will be an array, as multiple mutations might run concurrently. To identify specific ones, you can select `mutation.state.submittedAt` for a unique key.

Via the cache

When you optimistically update state before mutation, there's a risk of failure. Re-fetching can revert changes, but in some cases refetching isn't reliable. You can choose to roll back updates instead.

`useMutation`'s `onMutate` handler can return a value passed to `onError` and `onSettled`, typically a rollback function:

Updating a list of todos when adding a new todo

```
const queryClient = useQueryClient();

useMutation({
  mutationFn: updateTodo,
  onMutate: async (newTodo) => {
    await queryClient.cancelQueries({ queryKey: ['todos'] });
    const previousTodos = queryClient.getQueryData(['todos']);
    queryClient.setQueryData(['todos'], (old) => [...old, newTodo]);
    return { previousTodos };
  },
  onError: (err, newTodo, context) => {
    queryClient.setQueryData(['todos'], context.previousTodos);
  },
  onSettled: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
});
```

Updating a single todo

```
useMutation({
  mutationFn: updateTodo,
```

```
  onMutate: async (newTodo) => {
    await queryClient.cancelQueries({ queryKey: ['todos', newTodo.id] });
    const previousTodo = queryClient.getQueryData(['todos', newTodo.id]);
    queryClient.setQueryData(['todos', newTodo.id], newTodo);
    return { previousTodo, newTodo };
  },
  onError: (err, newTodo, context) => {
    queryClient.setQueryData(['todos', context.newTodo.id], context.previousTodo);
  },
  onSettled: (newTodo) =>
    queryClient.invalidateQueries({ queryKey: ['todos', newTodo.id] }),
});
```

When to use what

Using `variables` and updating UI directly is simpler if there's only one place needing the optimistic result, with less code. For multiple dependent UI parts, manipulating the cache directly manages consistency automatically.

Query Cancellation | TanStack Query React Docs

On this page

Default behavior

TanStack Query provides each query function with an AbortSignal. When a query becomes out-of-date or inactive, this signal will become aborted. This means all queries are cancellable, and you can respond to the cancellation inside your query function if desired. The best part is that it allows you to continue using normal async/await syntax while benefiting from automatic cancellation.

The AbortController API is available in most runtime environments, but if your environment doesn't support it, you can use a polyfill. Several are available here.

Behavior:

By default, queries that unmount or become unused before their promises resolve are *not* canceled. This means that after the promise resolves, data is available in the cache. If you remount the component and the data isn't garbage collected, it will be available immediately.

However, if you consume the `AbortSignal`, the Promise (e.g., fetch) will be canceled (aborted), and the Query will be canceled. Cancellation will revert the query's state to its previous status.

Using `fetch`

```
const query = useQuery({
  queryKey: ['todos'],
  queryFn: async ({ signal }) => {
    const todosResponse = await fetch('/todos', {
      // Pass the signal to fetch
      signal,
    });
    const todos = await todosResponse.json();

    const todoDetails = todos.map(async ({ details }) => {
      const response = await fetch(details, {
```

```
        // Pass it to each fetch
        signal,
      });
      return response.json();
    });

    return Promise.all(todoDetails);
  },
});
```

Using `axios` v0.22.0+

```
import axios from 'axios';

const query = useQuery({
  queryKey: ['todos'],
  queryFn: ({ signal }) =>
    axios.get('/todos', {
      // Pass the signal to axios
      signal,
    }),
});
```

Using `axios` with version lower than v0.22.0

```
import axios from 'axios';

const query = useQuery({
  queryKey: ['todos'],
  queryFn: ({ signal }) => {
    // Create a new CancelToken source for this request
    const CancelToken = axios.CancelToken;
    const source = CancelToken.source();

    const promise = axios.get('/todos', {
      // Pass the cancel token
      cancelToken: source.token,
    });

    // Cancel the request if TanStack Query signals to abort
    signal?.addEventListener('abort', () => {
      source.cancel('Query was cancelled by TanStack Query');
    });

    return promise;
  },
});
```

Using `XMLHttpRequest`

```
const query = useQuery({
  queryKey: ['todos'],
  queryFn: ({ signal }) => {
    return new Promise((resolve, reject) => {
      const oReq = new XMLHttpRequest();
      oReq.addEventListener('load', () => {
        resolve(JSON.parse(oReq.responseText));
      });
      signal?.addEventListener('abort', () => {
        oReq.abort();
        reject(new Error('Request aborted'));
      });
      oReq.open('GET', '/todos');
      oReq.send();
    });
  },
});
```

Using `graphql-request`

An `AbortSignal` can be set in the client `request` method.

```
import { GraphQLClient } from 'graphql-request';

const client = new GraphQLClient(endpoint);

const query = useQuery({
  queryKey: ['todos'],
  queryFn: ({ signal }) => {
    return client.request({ document: query, signal });
  },
});
```

Using `graphql-request` with version lower than v4.0.0

An `AbortSignal` can be set in the `GraphQLClient` constructor.

```
import { GraphQLClient } from 'graphql-request';

const query = useQuery({
  queryKey: ['todos'],
  queryFn: ({ signal }) => {
    const client = new GraphQLClient(endpoint, {
      signal,
    });
    return client.request(query, variables);
  },
});
```

Manual Cancellation

You might want to cancel a query manually, e.g., if the request takes too long. You can call `queryClient.cancelQueries({ queryKey })`, which cancels the query and reverts it to its previous state. If you used the `signal` passed to the query function, TanStack Query will also cancel the Promise.

```
const query = useQuery({
  queryKey: ['todos'],
  queryFn: async ({ signal }) => {
    const resp = await fetch('/todos', { signal });
    return resp.json();
  },
});

const queryClient = useQueryClient();

return (
  <button
    onClick={(e) => {
      e.preventDefault();
      queryClient.cancelQueries({ queryKey: ['todos'] });
    }}
  >
    Cancel
  </button>
);
```

Limitations

Cancellation does not work with `useSuspenseQuery`, `useSuspenseQueries`, or `useSuspenseInfiniteQuery`.

Edit on GitHub

https://github.com/tanstack/query/edit/main/docs/framework/react/guides/query-cancellation.md

Scroll Restoration

Traditionally, when you navigate to a previously visited page on a web browser, you would find that the page would be scrolled to the exact position where you were before you navigated away from that page. This is called **scroll restoration** and has been in a bit of a regression since web applications have started moving towards client side data fetching. With TanStack Query however, that's no longer the case.

Out of the box, "scroll restoration" for all queries (including paginated and infinite queries) Just Works™ in TanStack Query. The reason for this is that query results are cached and able to be retrieved synchronously when a query is rendered. As long as your queries are being cached long enough (the default time is 5 minutes) and have not been garbage collected, scroll restoration will work out of the box all the time.

Edit on GitHub

Filters

Some methods within TanStack Query accept a **QueryFilters** or **MutationFilters** object.

Query Filters

A query filter is an object with certain conditions to match a query with:

```
// Cancel all queries
await queryClient.cancelQueries()

// Remove all inactive queries that begin with `posts` in the key
queryClient.removeQueries({ queryKey: ['posts'], type: 'inactive' })

// Refetch all active queries
await queryClient.refetchQueries({ type: 'active' })

// Refetch all active queries that begin with `posts` in the key
await queryClient.refetchQueries({ queryKey: ['posts'], type: 'active' })
```

A query filter object supports the following properties:

- **queryKey?**: `QueryKey`
  Set this property to define a query key to match on.
- **exact?**: `boolean`
  If you don't want to search queries inclusively by query key, you can pass the `exact: true` option to return only the query with the exact query key you have passed.
- **type?**: `'active' | 'inactive' | 'all'`
  Defaults to `all`. When set to `active`, it will match active queries. When set to `inactive`, it will match inactive queries.
- **stale?**: `boolean`
  When set to `true`, it will match stale queries. When `false`, it matches fresh queries.
- **fetchStatus?**: `FetchStatus`
  When set to `fetching`, it will match queries currently fetching. When `paused`, it will match queries that wanted to fetch but have been paused. When `idle`, it will match queries not fetching.
- **predicate?**: `(query: Query) => boolean`
  This function will be used as a final filter on all matching queries. If no other filters are specified, this function is evaluated against every query in the cache.

Mutation Filters

A mutation filter is an object with certain conditions to match a mutation with:

```
// Get the number of all fetching mutations
await queryClient.isMutating()

// Filter mutations by mutationKey
await queryClient.isMutating({ mutationKey: ['post'] })
```

```
// Filter mutations using a predicate function
await queryClient.isMutating({
  predicate: (mutation) => mutation.state.variables?.id === 1,
})
```

A mutation filter object supports the following properties:

- **mutationKey?**: `MutationKey`
  Set this property to define a mutation key to match on.
- **exact?**: `boolean`
  If you don't want to search mutations inclusively by mutation key, pass `exact: true` to get only the mutation with the exact mutation key.
- **status?**: `MutationStatus`
  Allows filtering mutations by their status.
- **predicate?**: `(mutation: Mutation) => boolean`
  Function used as a final filter on all matching mutations.

Utils

matchQuery

```
const isMatching = matchQuery(filters, query)
```

Returns a boolean indicating whether a query matches the provided set of query filters.

matchMutation

```
const isMatching = matchMutation(filters, mutation)
```

Returns a boolean indicating whether a mutation matches the provided set of mutation filters.

Edit on GitHub

On this page

Additional navigation

Performance & Request Waterfalls | TanStack Query React Docs

On this page

What is a Request Waterfall?

A request waterfall is what happens when a request for a resource (code, css, images, data) does not start until *after* another request for a resource has finished.

Consider a web page. Before you can load things like the CSS, JS etc., the browser first needs to load the markup. This is a request waterfall.

```
1. -> Markup
2.   -> CSS
2.   -> JS
2.   -> Image
```

If you fetch your CSS inside a JS file, you now have a double waterfall:

```
1. -> Markup
2.   -> JS
3.     -> CSS
```

If that CSS uses a background image, it's a triple waterfall:

```
1. -> Markup
2.   -> JS
3.     -> CSS
4.       -> Image
```

The best way to spot and analyze request waterfalls is by opening your browser's devtools "Network" tab.

Each waterfall represents at least one roundtrip to the server, unless cached. The negative effects of request waterfalls are highly dependent on user latency. For example, a triple waterfall with 4 server roundtrips totals 1000ms at 250ms latency, while flattening it to 2 roundtrips reduces to 500ms.

Request Waterfalls & React Query

Without Server Rendering, React Query has a double waterfall:

```
1. -> Markup
2.   -> JS
3.     -> Query
```

Let's look at patterns that can lead to request waterfalls in React Query and how to avoid them.

Single Component Waterfalls / Serial Queries

When one component fetches a query, then another, causing a waterfall.

Example:

```
// Get the user
const { data: user } = useQuery({
  queryKey: ['user', email],
  queryFn: getUserByEmail,
})

const userId = user?.id

// Then get the user's projects
const {
  status,
  fetchStatus,
  data: projects,
} = useQuery({
  queryKey: ['projects', userId],
  queryFn: getProjectsByUser,
  enabled: !!userId,
})
```

To optimize, restructure API for a combined fetch or move the dependent query to server.

React Suspense Example

Multiple queries in series with `useSuspenseQuery`:

```
function App() {
  const usersQuery = useSuspenseQuery({ queryKey: ['users'], queryFn: fetchUsers });
  const teamsQuery = useSuspenseQuery({ queryKey: ['teams'], queryFn: fetchTeams });
  const projectsQuery = useSuspenseQuery({ queryKey: ['projects'], queryFn: fetchProje
  // Renders only after all queries finish
}
```

Fix: use `useSuspenseQueries` for parallel fetching:

```
const [usersQuery, teamsQuery, projectsQuery] = useSuspenseQueries({
  queries: [
    { queryKey: ['users'], queryFn: fetchUsers },
    { queryKey: ['teams'], queryFn: fetchTeams },
    { queryKey: ['projects'], queryFn: fetchProjects },
  ],
});
```

Nested Component Waterfalls

When both parent and child queries depend on each other, leading to a waterfall.

Example: parent fetches an article, child fetches comments. If unnecessary, refactor to fetch both in parent.

```
function Article({ id }) {
  const { data: articleData, isPending: articlePending } = useQuery({
    queryKey: ['article', id],
```

```
    queryFn: getArticleById,
  });

  const { data: commentsData, isPending: commentsPending } = useQuery({
    queryKey: ['article-comments', id],
    queryFn: getArticleCommentsById,
  });

  if (articlePending) return 'Loading article...';

  return (
    <>
      <ArticleHeader articleData={articleData} />
      <ArticleBody articleData={articleData} />
      {commentsPending ? 'Loading comments...' : <Comments commentsData={commentsData}
    </>
  );
}
```

Alternatively, hoist both queries to parent to fetch in parallel.

Dependent Nested Component Waterfall

Example:

```
function Feed() {
  const { data, isPending } = useQuery({ queryKey: ['feed'], queryFn: getFeed });
  if (isPending) return 'Loading feed...';
  // render feed items...
}

function GraphFeedItem({ feedItem }) {
  const { data, isPending } = useQuery({ queryKey: ['graph', feedItem.id], queryFn: ge
  // ...
}
```

`getGraphDataById` runs only if `feedItem` is a graph and the ID is available. To flatten, include the second query in parent or prefetch.

Code Splitting

Splitting JS into smaller chunks can add request waterfalls, especially if combined with queries.

Example of lazy loading component:

```
const GraphFeedItem = React.lazy(() => import('./GraphFeedItem'));

function Feed() {
  const { data, isPending } = useQuery({ queryKey: ['feed'], queryFn: getFeed });
  if (isPending) return 'Loading feed...';

  return (
    <>
      {data.map((item) =>
        item.type === 'GRAPH' ? <React.Suspense fallback="Loading..." key={item.id}><G
      )}
```

```
    </>
  );
}
```

This can cause a 5-roundtrip waterfall:

1. getFeed()
2. load JS for `GraphFeedItem`
3. load `getGraphDataById()`

Solution: prefetch queries at router level or move queries into parent for parallel fetching.

Summary and takeaways

Request Waterfalls are common, complex, and often accidental. They can be caused by:

- Child queries added without considering parent
- Parent queries added without child's awareness
- Moving components with queries in a nested manner

Be mindful and analyze network activity regularly. While flattening every waterfall isn't always feasible, reducing high-impact ones improves performance.

Next, explore prefetching and router-based strategies to further flatten waterfalls.

Edit on GitHub: Edit request-waterfalls.md

Additional Resources

- Filters
- Prefetching & Router Integration
- Advanced Server Rendering

Prefetching & Router Integration

On this page

Prefetching & Router Integration

When you know or suspect that a certain piece of data will be needed, you can use prefetching to populate the cache with that data ahead of time, leading to a faster experience.

There are a few different prefetching patterns:

1. In event handlers
2. In components
3. Via router integration
4. During Server Rendering (another form of router integration)

In this guide, we'll take a look at the first three, while the fourth will be covered in depth in the Server Rendering & Hydration guide and the Advanced Server Rendering guide.

One specific use of prefetching is to avoid Request Waterfalls, for an in-depth background and explanation of those, see the Performance & Request Waterfalls guide.

prefetchQuery & prefetchInfiniteQuery

Before jumping into the different specific prefetch patterns, let's look at the `prefetchQuery` and `prefetchInfiniteQuery` functions. First a few basics:

- Out of the box, these functions use the default `staleTime` configured for the `queryClient` to determine whether existing data in the cache is fresh or needs to be fetched again.
- You can also pass a specific `staleTime` like this: `prefetchQuery({ queryKey: ['todos'], queryFn: fn, staleTime: 5000 })`
  - This `staleTime` is only used for the prefetch, you still need to set it for any `useQuery` call as well.
  - If you want to ignore `staleTime` and instead always return data if it's available in the cache, you can use the `ensureQueryData` function.
  - Tip: If you are prefetching on the server, set a default `staleTime` higher than `0` for that `queryClient` to avoid having to pass in a specific `staleTime` to each prefetch call.
- If no instances of `useQuery` appear for a prefetched query, it will be deleted and garbage collected after the time specified in `gcTime`.
- These functions return `Promise<void>` and thus never return query data. If that's needed, use `fetchQuery` / `fetchInfiniteQuery` instead.
- The prefetch functions never throw errors because they usually try to fetch again in a `useQuery`, which is a graceful fallback. For error catching, use `fetchQuery` / `fetchInfiniteQuery`.

Usage example for `prefetchQuery`:

```
const prefetchTodos = async () => {
  await queryClient.prefetchQuery({
    queryKey: ['todos'],
    queryFn: fetchTodos,
  });
}
```

Infinite Queries:

```
const prefetchProjects = async () => {
  await queryClient.prefetchInfiniteQuery({
    queryKey: ['projects'],
    queryFn: fetchProjects,
    initialPageParam: 0,
    getNextPageParam: (lastPage) => lastPage.nextCursor,
    pages: 3, // prefetch the first 3 pages
  });
}
```

Next, let's look at how to use these in different scenarios.

Prefetch in event handlers

A straightforward way is to do it when the user interacts with something, for example using `queryClient.prefetchQuery` on `onMouseEnter` or `onFocus`.

```
function ShowDetailsButton() {
  const queryClient = useQueryClient();

  const prefetch = () => {
    queryClient.prefetchQuery({
      queryKey: ['details'],
      queryFn: getDetailsData,
      staleTime: 60000,
    });
  };

  return (
    <button onMouseEnter={prefetch} onFocus={prefetch} onClick={...}>
      Show Details
    </button>
  );
}
```

Prefetch in components

Prefetching during component lifecycle is useful when a child or descendant will need specific data but can't render until another query completes. Example:

```
function Article({ id }) {
  const { data: articleData, isPending } = useQuery({
    queryKey: ['article', id],
```

```
    queryFn: getArticleById,
  });

  if (isPending) return 'Loading article...';

  return (
    <>
      <ArticleHeader articleData={articleData} />
      <ArticleBody articleData={articleData} />
      <Comments id={id} />
    </>
  );
}

function Comments({ id }) {
  const { data, isPending } = useQuery({
    queryKey: ['article-comments', id],
    queryFn: getArticleCommentsById,
  });

  ...
}
```

To prefetch `article-comments` without waiting, in the parent:

```
function Article({ id }) {
  const { data: articleData, isPending } = useQuery({
    queryKey: ['article', id],
    queryFn: getArticleById,
  });

  // Prefetch comments
  useQuery({
    queryKey: ['article-comments', id],
    queryFn: getArticleCommentsById,
    notifyOnChangeProps: [],
  });

  if (isPending) return 'Loading article...';

  return (
    <>
      <ArticleHeader articleData={articleData} />
      <ArticleBody articleData={articleData} />
      <Comments id={id} />
    </>
  );
}
```

This starts fetching `article-comments` immediately and flattens the waterfall.

Prefetching with Suspense

You can't use `useSuspenseQueries` for prefetch since it blocks rendering. Instead, use `usePrefetchQuery` or `usePrefetchInfiniteQuery`, then in the component that needs data, call `useSuspenseQuery`:

```
function ArticleLayout({ id }) {
  usePrefetchQuery({
    queryKey: ['article-comments', id],
    queryFn: getArticleCommentsById,
  });

  return (
    <Suspense fallback="Loading article">
      <Article id={id} />
    </Suspense>
  );
}

function Article({ id }) {
  const { data: articleData } = useSuspenseQuery({
    queryKey: ['article', id],
    queryFn: getArticleById,
  });

  ...
}
```

Prefetch inside of query function

For cases when every fetch involves related data:

```
const queryClient = useQueryClient();
const { data: articleData } = useQuery({
  queryKey: ['article', id],
  queryFn: (...args) => {
    queryClient.prefetchQuery({
      queryKey: ['article-comments', id],
      queryFn: getArticleCommentsById,
    });
    return getArticleById(...args);
  },
});
```

Prefetch in an effect

```
const queryClient = useQueryClient();

useEffect(() => {
  queryClient.prefetchQuery({
    queryKey: ['article-comments', id],
    queryFn: getArticleCommentsById,
  });
}, [queryClient, id]);
```

Note: If using `useSuspenseQuery` in the same component, this may run after the initial query resolves, which might not be desired.

Summary

Prefetch during component lifecycle:

- Before a suspense boundary with `usePrefetchQuery` / `usePrefetchInfiniteQuery`
- Use `useQuery` / `useSuspenseQueries` and ignore result
- Prefetch inside the query function
- Prefetch in an effect

Next, a more advanced topic.

Dependent Queries & Code Splitting

Sometimes, prefetching depends on previous results, e.g., fetching comments only after the article is loaded.

Example:

```
// This lazy loads the component
const GraphFeedItem = React.lazy(() => import('./GraphFeedItem'));

function Feed() {
  const { data, isPending } = useQuery({
    queryKey: ['feed'],
    queryFn: getFeed,
  });

  if (isPending) return 'Loading feed...';

  return (
    <>
      {data.map((feedItem) =>
        feedItem.type === 'GRAPH' ? (
          <GraphFeedItem key={feedItem.id} feedItem={feedItem} />
        ) : (
          <StandardFeedItem key={feedItem.id} feedItem={feedItem} />
        )
      )}
    </>
  );
}

function GraphFeedItem({ feedItem }) {
  const { data, isPending } = useQuery({
    queryKey: ['graph', feedItem.id],
    queryFn: getGraphDataById,
  });

  ...
}
```

This leads to a double request waterfall:

1. `getFeed()`
2. `getGraphDataById()` for each graph item

To mitigate this, prefetch the graph data conditionally within the feed's query function:

```
function Feed() {
  const queryClient = useQueryClient();
```

```
  const { data, isPending } = useQuery({
    queryKey: ['feed'],
    queryFn: async (...args) => {
      const feed = await getFeed(...args);
      for (const feedItem of feed) {
        if (feedItem.type === 'GRAPH') {
          queryClient.prefetchQuery({
            queryKey: ['graph', feedItem.id],
            queryFn: getGraphDataById,
          });
        }
      }
      return feed;
    },
  });

  ...
}
```

This loads code and data in parallel, reducing waterfall.

Router Integration

Integrate prefetching at the router level for SSR or client-side apps, explicitly declaring data needs per route.

Example with TanStack Router:

```
const queryClient = new QueryClient();
const router = new Router({
  routes: [
    {
      path: 'article',
      loader: async ({ context: { queryClient } }) => {
        // Prefetch comments and article, but only wait on article
        queryClient.prefetchQuery({ queryKey: ['comments'], queryFn: fetchComments });
        await queryClient.prefetchQuery({ queryKey: ['article'], queryFn: fetchArticle
      },
    },
  ],
});
```

This approach can be combined with SSR strategies.

See more in the SSR guide.

Manually Priming a Query

If data is already available, avoid fetching by directly updating the cache:

```
queryClient.setQueryData(['todos'], todos);
```

Further Reading

- Seeding the Query Cache
- Server Rendering & Hydration

Our Partners

- Query.gg - The Official React Query Course

Links

- TanStack Form
- TanStack Config

Server Rendering & Hydration

In this guide you'll learn how to use React Query with server rendering.

See the guide on Prefetching & Router Integration for some background. You might also want to check out the Performance & Request Waterfalls guide before that.

For advanced server rendering patterns, such as streaming, Server Components and the new Next.js app router, see the Advanced Server Rendering guide.

If you just want to see some code, you can skip ahead to the Full Next.js pages router example or the Full Remix example below.

Server Rendering & React Query

So what is server rendering anyway? The rest of this guide will assume you are familiar with the concept, but let's spend some time to look at how it relates to React Query. Server rendering is the act of generating the initial html on the server, so that the user has some content to look at as soon as the page loads. This can happen on demand when a page is requested (SSR). It can also happen ahead of time either because a previous request was cached, or at build time (SSG).

If you've read the Request Waterfalls guide, you might remember this:

```
1. |-&gt; Markup (without content)
2. |-&gt; JS
3. |-&gt; Query
```

With a client rendered application, these are the minimum 3 server roundtrips you will need to make before getting any content on the screen for the user. One way of viewing server rendering is that it turns the above into this:

```
1. |-&gt; Markup (with content AND initial data)
2. |-&gt; JS
```

As soon as **1.** is complete, the user can see the content and when **2.** finishes, the page is interactive and clickable. Because the markup also contains the initial data we need, step **3.** does not need to run on the client at all, at least until you want to revalidate the data for some reason.

This is all from the client's perspective. On the server, we need to **prefetch** that data before we generate/render the markup, we need to **dehydrate** that data into a serializable format we can embed in the markup, and on the client we need to **hydrate** that data into a React Query cache so we can avoid doing a new fetch on the client.

Read on to learn how to implement these three steps with React Query.

A quick note on Suspense

This guide uses the regular `useQuery` API. While we don't necessarily recommend it, it is possible to replace this with `useSuspenseQuery` instead *as long as you always prefetch all your queries*. The upside is that you get to use `<Suspense>` for loading states on the client.

If you do forget to prefetch a query when using `useSuspenseQuery`, the consequences will depend on the framework you are using. In some cases, the data will Suspend and get fetched on the server but never be hydrated to the client, where it will fetch again. This can lead to a markup hydration mismatch because the server and the client tried to render different things.

Initial setup

The first steps of using React Query is always to create a `queryClient` and wrap the application in a `<QueryClientProvider>`. When doing server rendering, it's important to create the `queryClient` instance **inside of your app**, in React state (an instance ref works fine too). **This ensures that data is not shared between different users and requests**, while still only creating the `queryClient` once per component lifecycle.

Next.js pages router:

```
// _app.tsx
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'

export default function MyApp({ Component, pageProps }) {
  // Create a new QueryClient inside React state
  const [queryClient] = React.useState(
    () =>
      new QueryClient({
        defaultOptions: {
          queries: {
            // With SSR, we usually want to set some default staleTime
            // above 0 to avoid refetching immediately on the client
            staleTime: 60 * 1000,
          },
        },
      })
  )

  return (
    <QueryClientProvider client={queryClient}>
      <Component {...pageProps} />
    </QueryClientProvider>
  )
}
```

In each route:

```
// pages/posts.tsx
import {
  dehydrate,
  HydrationBoundary,
  QueryClient,
  useQuery,
} from '@tanstack/react-query'

// This could also be getServerSideProps
export async function getStaticProps() {
  const queryClient = new QueryClient()
```

```
  await queryClient.prefetchQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
  })

  return {
    props: {
      dehydratedState: dehydrate(queryClient),
    },
  }
}

function Posts() {
  const { data } = useQuery({ queryKey: ['posts'], queryFn: getPosts })

  // ...
}

export default function PostsRoute({ dehydratedState }) {
  return (
    <HydrationBoundary state={dehydratedState}>
      <Posts />
    </HydrationBoundary>
  )
}
```

Full Remix example:

```
// app/root.tsx
import { Outlet } from '@remix-run/react'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'

export default function MyApp() {
  const [queryClient] = React.useState(
    () =>
      new QueryClient({
        defaultOptions: {
          queries: {
            staleTime: 60 * 1000,
          },
        },
      }),
  )

  return (
    <QueryClientProvider client={queryClient}>
      <Outlet />
    </QueryClientProvider>
  )
}
```

In each route, you can do:

```
// app/routes/posts.tsx
import { json } from '@remix-run/node'
```

```
import {
  dehydrate,
  HydrationBoundary,
  QueryClient,
  useQuery,
  useLoaderData,
} from '@tanstack/react-query'

export async function loader() {
  const queryClient = new QueryClient()

  await queryClient.prefetchQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
  })

  return json({ dehydratedState: dehydrate(queryClient) })
}

function Posts() {
  const { dehydratedState } = useLoaderData()

  const { data } = useQuery({ queryKey: ['posts'], queryFn: getPosts })

  // ...
}

export default function PostsRoute() {
  const { dehydratedState } = useLoaderData()
  return (
    <HydrationBoundary state={dehydratedState}>
      <Posts />
    </HydrationBoundary>
  )
}
```

Optional - Remove boilerplate

Having to wrap your routes in `<HydrationBoundary>` each time can seem verbose. To avoid this, modify your `_app.tsx`:

```
// _app.tsx
import {
  HydrationBoundary,
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'

export default function MyApp({ Component, pageProps }) {
  const [queryClient] = React.useState(() => new QueryClient())

  return (
    <QueryClientProvider client={queryClient}>
      <HydrationBoundary state={pageProps.dehydratedState}>
        <Component {...pageProps} />
```

```
      </HydrationBoundary>
    </QueryClientProvider>
  )
}
```

And in your page components, just export the data-fetching logic and your component:

```
// pages/posts.tsx
export default function Posts() { ... }
```

This way, the hydration boundary is handled globally, reducing boilerplate.

Prefetching dependent queries

You can prefetch dependent queries on the server for SSR. For example:

```
// For Remix, rename this to loader
export async function getServerSideProps() {
  const queryClient = new QueryClient()

  const user = await queryClient.fetchQuery({
    queryKey: ['user', email],
    queryFn: getUserByEmail,
  })

  if (user?.userId) {
    await queryClient.prefetchQuery({
      queryKey: ['projects', userId],
      queryFn: getProjectsByUser,
    })
  }

  return { props: { dehydratedState: dehydrate(queryClient) } }
}
```

Error handling

React Query's default behavior is graceful degradation:

- `queryClient.prefetchQuery(...)` never throws.
- `dehydrate(...)` only includes successful queries.

To handle errors explicitly, use `queryClient.fetchQuery(...)` which throws if it fails:

```
let result

try {
  result = await queryClient.fetchQuery(...)
} catch (error) {
  // Handle error as needed
}
```

To include failed queries in the dehydrated state:

```
dehydrate(queryClient, {
  shouldDehydrateQuery: (query) => {
```

```
    // Includes all queries, including failed ones
    return true
  },
})
```

Serialization

`dehydrate()` results are serialized by the framework to embed in the markup safely. Be aware of issues with non-serializable values (`Error` , `Date` , `Map` , `Set` , etc.).

For custom serialization, consider `superjson` or `serialize-javascript` , both are safe against XSS.

A note about request waterfalls

Server rendering helps flatten nested request waterfalls, e.g.:

```
1. |&gt; getFeed()
2.   |&gt; getGraphDataById()
```

becomes:

```
1. |&gt; getFeed() + getGraphDataById()
2.   |&gt; JS for <Feed> and <GraphFeedItem>
```

This parallelizes JS loading on the client, but on subsequent page navigations, the waterfall reappears. Modern prefetching strategies can mitigate this, but cross-page SPA navigation will still experience waterfalls unless you use techniques like React Server Components.

Important Caveats & Tips

Staleness is measured from when the query was fetched on the server

Queries are considered stale based on `dataUpdatedAt` . The server time (UTC) is used, so it's immune to timezones. Use higher `staleTime` to prevent unnecessary refetches.

High memory consumption on server

Creating a new `QueryClient` per request retains cached data in memory. Garbage collection (`gcTime`) defaults to `Infinity` , but can be adjusted to reduce memory usage, e.g., `2 * 1000` .

Caveat for Next.js rewrites

Next.js rewrites can cause a second hydration and mismatched data due to loss of referential equality. Be aware if using rewrites with client-side hydration.

Edit on GitHub: https://github.com/tanstack/query/edit/main/docs/framework/react/guides/ssr.md

Advanced Server Rendering

Table of Contents

Server Components & Next.js app router

We won't cover Server Components in depth here, but the short version is that they are components guaranteed to *only* run on the server, both for initial page view and *also on page transitions*. This is similar to Next.js's `getServerSideProps` / `getStaticProps` and Remix's `loader`.

How do we pass data from server to client?

Use React Query's `hydration` APIs. Remember, this approach mirrors passing prefetched data in framework loaders and can be used to enable SSR with React Query.

A quick note on terminology

The terms "server" and "client" are often used, but not always align perfectly with *Server Components* and *Client Components*.

- Server Components *only* run on the server.
- Client Components *can* run on both server (during SSR) and in the client (browser).

Server Components happen during a "loader" phase, and Client Components during the "application" phase (client or server-side rendering).

Initial setup

Create a `queryClient` and wrap your application in a `QueryClientProvider`.
The filename conventions can vary, but typically:

```
// e.g., app/providers.tsx
'use client'
import { isServer, QueryClient, QueryClientProvider } from '@tanstack/react-query';

function makeQueryClient() {
  return new QueryClient({
    defaultOptions: {
```

```
      queries: {
        // For SSR, default staleTime above 0 prevents immediate refetch
        staleTime: 60 * 1000,
      },
    },
  });
}

let browserQueryClient: QueryClient | undefined = undefined;

export default function Providers({ children }: { children: React.ReactNode }) {
  const queryClient = isServer ? makeQueryClient() : (browserQueryClient ??= makeQuery

  return (
    <QueryClientProvider client={queryClient}>
      {children}
    </QueryClientProvider>
  );
}
```

In your root layout:

```
// e.g., app/layout.tsx
import Providers from './providers';

export default function RootLayout({ children }: { children: React.ReactNode }) {
  return (
    <html lang="en">
      <head />
      <body>
        <Providers>{children}</Providers>
      </body>
    </html>
  );
}
```

Prefetching and de/hydrating data

Using `getStaticProps` (or `getServerSideProps`), prefetch data on the server:

```
// pages/posts.tsx
import { dehydrate, HydrationBoundary, QueryClient, useQuery } from '@tanstack/react-q

export async function getStaticProps() {
  const queryClient = new QueryClient();

  await queryClient.prefetchQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
  });

  return {
    props: { dehydratedState: dehydrate(queryClient) },
  };
}
```

```
function Posts() {
  const { data } = useQuery({ queryKey: ['posts'], queryFn: getPosts });
  // ...
}

export default function PostsRoute({ dehydratedState }) {
  return (
    <HydrationBoundary state={dehydratedState}>
      <Posts />
    </HydrationBoundary>
  );
}
```

For the App Router, create a server component that prefetches and hydrates:

```
// app/posts/page.tsx
import { dehydrate, HydrationBoundary, QueryClient } from '@tanstack/react-query';
import Posts from './posts';

export default async function PostsPage() {
  const queryClient = new QueryClient();

  await queryClient.prefetchQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
  });

  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <Posts />
    </HydrationBoundary>
  );
}
```

Client component with preloaded data

```
// app/posts/posts.tsx
'use client';

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

export default function Posts() {
  const { data } = useQuery({
    queryKey: ['posts'],
    queryFn: () => getPosts(),
  });
  // ...
}
```

Note: You can also use `useSuspenseQuery` to leverage the Suspense API.

Prefetching without `await` and streaming SSR

You can start prefetching early without awaiting, and stream data to the client:

```
// app/get-query-client.ts
import { QueryClient, defaultShouldDehydrateQuery } from '@tanstack/react-query';

export function getQueryClient() {
  if (isServer) {
    return new QueryClient({
      defaultOptions: {
        queries: { staleTime: 60 * 1000 },
        dehydrate: {
          shouldDehydrateQuery: (query) =>
            defaultShouldDehydrateQuery(query) || query.state.status === 'pending',
          shouldRedactErrors: (error) => false,
        },
      },
    });
  } else {
    // client
    // ...
  }
}
```

Then, in your server component:

```
// app/posts/page.tsx
import { dehydrate, HydrationBoundary } from '@tanstack/react-query';
import { getQueryClient } from './get-query-client';

export default function PostsPage() {
  const queryClient = getQueryClient();

  // No await here
  queryClient.prefetchQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
  });

  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <Posts />
    </HydrationBoundary>
  );
}
```

And on the client, use `useSuspenseQuery` to consume the prefetched data:

```
// app/posts/posts.tsx
'use client';

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

export default function Posts() {
  const { data } = useSuspenseQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
  });
```

```
  // ...
}
```

You can also serialize/deserialize custom data types with `serializeData` and `deserializeData` options.

Streaming with Server Components

Next.js auto-streams parts of the app via `<Suspense>` boundaries. React Query supports this, as queries resolve and stream as the data becomes available.

```
// app/providers.tsx
'use client';

import { isServer, QueryClient, QueryClientProvider } from '@tanstack/react-query';
import { ReactQueryStreamedHydration } from '@tanstack/react-query-next-experimental';

function makeQueryClient() {
  return new QueryClient({
    defaultOptions: {
      queries: { staleTime: 60 * 1000 },
    },
  });
}

let browserQueryClient: QueryClient | undefined;

export function Providers({ children }: { children: React.ReactNode }) {
  const queryClient = isServer ? makeQueryClient() : (browserQueryClient ??= makeQuery
  return (
    <QueryClientProvider client={queryClient}>
      <ReactQueryStreamedHydration>{children}</ReactQueryStreamedHydration>
    </QueryClientProvider>
  );
}
```

This approach streamlines data fetching, avoids manual prefetching, and enhances UX.

Final words

React Query's integration with Server Components and streaming is evolving. Given the flux, participate by providing feedback on the API or contribute to docs via GitHub.

Always choose the right tool for the job, balancing developer experience, performance, and complexity.

Edit on GitHub

Caching Examples | TanStack Query React Docs

> Please thoroughly read the Important Defaults before reading this guide

Basic Example

This caching example illustrates the story and lifecycle of:

- Query Instances with and without cache data
- Background Refetching
- Inactive Queries
- Garbage Collection

Let's assume we are using the default **gcTime** of **5 minutes** and the default **staleTime** of **0**.

1. A new instance of `useQuery({ queryKey: ['todos'], queryFn: fetchTodos })` mounts.
    - Since no other queries have been made with the `['todos']` query key, this query will show a hard loading state and make a network request to fetch the data.
    - When the network request has completed, the returned data will be cached under the `['todos']` key.
    - The hook will mark the data as stale after the configured **staleTime** (defaults to **0**, or immediately).
2. A second instance of `useQuery({ queryKey: ['todos'], queryFn: fetchTodos })` mounts elsewhere.
    - Since the cache already has data for the `['todos']` key from the first query, that data is immediately returned from the cache.
    - The new instance triggers a new network request using its query function.
        - Note that regardless of whether both `fetchTodos` query functions are identical or not, both queries' **status** are updated (including **isFetching**, **isPending**, and other related values) because they have the same query key.
    - When the request completes successfully, the cache's data under the `['todos']` key is updated with the new data, and both instances are updated with the new data.
3. Both instances of the `useQuery({ queryKey: ['todos'], queryFn: fetchTodos })` query are unmounted and no longer in use.
    - Since there are no more active instances of this query, a garbage collection timeout is set using **gcTime** to delete and garbage collect the query (defaults to **5 minutes**).
4. Before the cache timeout has completed, another instance of `useQuery({ queryKey: ['todos'], queryFn: fetchTodos })` mounts.
    - The query immediately returns the available cached data while the `fetchTodos` function is being run in the background. When it completes successfully, it will populate the cache with fresh data.
5. The final instance of `useQuery({ queryKey: ['todos'], queryFn: fetchTodos })` unmounts.
    - No more instances of this query appear within **5 minutes**.
        - The cached data under the `['todos']` key is deleted and garbage collected.

Edit on GitHub

Render Optimizations

Structural Sharing

React Query uses a technique called *structural sharing* to ensure that as many references as possible will be kept intact between re-renders. If data is fetched over the network, usually, you'll get a completely new reference by JSON parsing the response. However, React Query will keep the original reference if *nothing* changed in the data. If a subset changed, React Query will keep the unchanged parts and only replace the changed parts.

> **Note:** This optimization only works if the `queryFn` returns JSON compatible data. You can turn it off by setting `structuralSharing: false` globally or on a per-query basis, or you can implement your own structural sharing by passing a function to it.

Referential Identity

The top level object returned from `useQuery`, `useInfiniteQuery`, `useMutation`, and the array returned from `useQueries` is **not referentially stable**. It will be a new reference on every render. However, the `data` properties returned from these hooks will be as stable as possible.

Tracked Properties

React Query will only trigger a re-render if one of the properties returned from `useQuery` is actually *used*. This is done by using a Proxy object. This avoids unnecessary re-renders, e.g., because properties like `isFetching` or `isStale` may change often but are not used in the component.

You can customize this feature by setting `notifyOnChangeProps` manually globally or on a per-query basis. To disable, set `notifyOnChangeProps: 'all'`.

> **Note:** The get trap of a proxy is invoked by accessing a property, either via destructuring or directly. Using object rest destructuring disables this optimization. There is an ESLint rule to guard against this.

Select

You can use the `select` option to subscribe to a subset of the data. This is useful for optimized data transformations or avoiding unnecessary re-renders.

```
export const useTodos = (select) => {
  return useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodos,
    select,
  });
};

export const useTodoCount = () => {
  return useTodos((data) => data.length);
};
```

A component using `useTodoCount` will only re-render if the length of todos changes, **not** if other properties (e.g., the name of a todo) change.

> **Note:** `select` operates on successfully cached data and is not the place to throw errors. Errors should come from the `queryFn`. Returning an error from `select` results in `data` being `undefined` and `isSuccess` being `true`.

Memoization

The `select` function will only re-run if:

- The `select` function itself changed referentially
- `data` changed

Inline `select` functions run on every render. To avoid this, wrap with `useCallback` or extract to a stable function reference:

```
// wrapped in useCallback
export const useTodoCount = () => {
  return useTodos(useCallback((data) => data.length, []));
};
```

Or

```
// stable function reference
const selectTodoCount = (data) => data.length;

export const useTodoCount = () => {
  return useTodos(selectTodoCount);
};
```

Further Reading

For an in-depth guide about these topics, read React Query Render Optimizations from the Community Resources.

Edit on GitHub

Query Functions | TanStack Query React Docs

On this page

Query Functions

A query function can be literally any function that *returns a promise*. The promise should either **resolve the data** or **throw an error**.

Valid query function configurations:

```
useQuery({ queryKey: ['todos'], queryFn: fetchAllTodos })
useQuery({ queryKey: ['todos', todoId], queryFn: () => fetchTodoById(todoId) })
useQuery({
  queryKey: ['todos', todoId],
  queryFn: async () => {
    const data = await fetchTodoById(todoId)
    return data
  },
})
useQuery({
  queryKey: ['todos', todoId],
  queryFn: ({ queryKey }) => fetchTodoById(queryKey[1]),
})
```

Handling and Throwing Errors

For TanStack Query to determine a query has errored, the query function **must throw** or return a **rejected Promise**. Any error thrown in the query function will be persisted on the *error* state.

```
const { error } = useQuery({
  queryKey: ['todos', todoId],
  queryFn: async () => {
    if (somethingGoesWrong) {
      throw new Error('Oh no!')
    }
    if (somethingElseGoesWrong) {
      return Promise.reject(new Error('Oh no!'))
    }

    return data
```

```
  },
})
```

Usage with *fetch* and other clients that do not throw by default

While utilities like *axios* or *graphql-request* automatically throw errors, *fetch* does not. You need to throw errors yourself:

```
useQuery({
  queryKey: ['todos', todoId],
  queryFn: async () => {
    const response = await fetch('/todos/' + todoId)
    if (!response.ok) {
      throw new Error('Network response was not ok')
    }
    return response.json()
  },
})
```

Query Function Variables

Query keys are not only for identifying data but are passed into your query function as part of *QueryFunctionContext*. This allows extraction of variables:

```
function Todos({ status, page }) {
  const result = useQuery({
    queryKey: ['todos', { status, page }],
    queryFn: fetchTodoList,
  })
}

// Access variables in your query function
function fetchTodoList({ queryKey }) {
  const [_key, { status, page }] = queryKey
  return new Promise()
}
```

QueryFunctionContext

This object is passed to each query function and contains:

- `queryKey: QueryKey` — Query Keys
- `client: QueryClient` — QueryClient
- `signal?: AbortSignal` — AbortSignal, used for Query Cancellation
- `meta: Record<string, unknown> | undefined` — Optional additional info

Additionally, for **Infinite Queries**:

- `pageParam: TPageParam` — current page parameter
- `direction: 'forward' | 'backward'` — deprecated; used for page fetch direction

Edit on GitHub

https://github.com/tanstack/query/edit/main/docs/framework/react/guides/query-functions.md

---

On this page

Additional Resources

- Query Keys
- Query Options

Our Partners

- React Query Course - Query.gg

Want to Skip the Docs?

Query.gg - The Official React Query Course
“If you’re serious about *really* understanding React Query, there’s no better way than with query.gg” — Tanner Linsley

Learn More

Related Tools

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

- TanStack Config
  The build and publish utilities used by all of our projects.

Want to Skip the Docs?

Query.gg - The Official React Query Course
“If you’re serious about *really* understanding React Query, there’s no better way than with query.gg” — Tanner Linsley

Learn More

Default Query Function

If you find yourself wishing for a way to share the same query function throughout your app and only use query keys to identify what to fetch, you can do that by providing a **default query function** to TanStack Query:

```
// Define a default query function that will receive the query key
const defaultQueryFn = async ({ queryKey }) => {
  const { data } = await axios.get(
    `https://jsonplaceholder.typicode.com${queryKey[0]}`
  )
  return data
}

// Provide the default query function to your app with defaultOptions
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      queryFn: defaultQueryFn,
    },
  },
})

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <YourApp />
    </QueryClientProvider>
  )
}

// All you have to do now is pass a key!
function Posts() {
  const { status, data, error, isFetching } = useQuery({ queryKey: ['/posts'] })
  // ...
}

// You can even leave out the queryFn and just go straight into options
function Post({ postId }) {
  const { status, data, error, isFetching } = useQuery({
    queryKey: [`/posts/${postId}`],
    enabled: !!postId,
  })
  // ...
}
```

If you ever want to override the default `queryFn`, just provide your own as usual.

Edit on GitHub

Suspense

React Query can also be used with React's Suspense for Data Fetching APIs. For this, we have dedicated hooks:

- useSuspenseQuery
- useSuspenseInfiniteQuery
- useSuspenseQueries

Additionally, you can use the `useQuery().promise` and `React.use()` (Experimental).

When using suspense mode, *status* states and *error* objects are not needed and are then replaced by usage of the `<React.Suspense>` component (including the `fallback` prop) and React error boundaries for catching errors.

Please read the Resetting Error Boundaries and look at the Suspense Example for more information on how to set up suspense mode.

If you want mutations to propagate errors to the nearest error boundary (similar to queries), you can set the `throwOnError` option to `true` as well.

Enabling suspense mode for a query:

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

const { data } = useSuspenseQuery({ queryKey, queryFn })
```

This works nicely in TypeScript because `data` is guaranteed to be defined (errors and loading states are handled by Suspense and Error Boundaries).

Limitations

- You can't conditionally enable/disable the Query with suspense; typically, all queries inside one component are fetched sequentially.
- `placeholderData` doesn't exist for this Query. To prevent the UI from being replaced by a fallback during an update, wrap your updates changing the `QueryKey` into startTransition.

`throwOnError` default

Not all errors are thrown to the nearest Error Boundary by default — errors are only thrown if there's no other data to show. If the cache has data, the component renders even if data is *stale*. The default is:

```
throwOnError: (error, query) => typeof query.state.data === 'undefined'
```

To manually throw errors and handle them with Error Boundaries:

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

const { data, error, isFetching } = useSuspenseQuery({ queryKey, queryFn })

if (error && !isFetching) {
  throw error
}
// continue rendering data
```

Resetting Error Boundaries

Whether using `suspense` or `throwOnError`, you need a way to reset queries after errors:

- Using `<QueryErrorResetBoundary>` component:

```
import { QueryErrorResetBoundary } from '@tanstack/react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App = () => (
  <QueryErrorResetBoundary>
    {({ reset }) => (
      <ErrorBoundary
        onReset={reset}
        fallbackRender={({ resetErrorBoundary }) => (
          <div>
            There was an error!
            <Button onClick={() => resetErrorBoundary()}>Try again</Button>
          </div>
        )}
      >
        <Page />
      </ErrorBoundary>
    )}
  </QueryErrorResetBoundary>
)
```

- Using `useQueryErrorResetBoundary()` hook:

```
import { useQueryErrorResetBoundary } from '@tanstack/react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App = () => {
  const { reset } = useQueryErrorResetBoundary()
  return (
    <ErrorBoundary
      onReset={reset}
      fallbackRender={({ resetErrorBoundary }) => (
        <div>
          There was an error!
          <Button onClick={() => resetErrorBoundary()}>Try again</Button>
        </div>
      )}
    >
      <Page />
    </ErrorBoundary>
  )
}
```

This resets errors globally or in the closest boundary.

Fetch-on-render vs Render-as-you-fetch

React Query in suspense mode works well as a **Fetch-on-render** solution, triggering queries upon component mount.

To implement **Render-as-you-fetch**, prefetch data on routing callbacks or user interactions to load queries before mounting components.

Suspense on the Server with Streaming

If using Next.js, you can use the experimental `@tanstack/react-query-next-experimental` package to fetch data on the server with streaming:

Wrap your app with `ReactQueryStreamedHydration`:

```
// app/providers.tsx
'use client'

import {
  isServer,
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'
import * as React from 'react'
import { ReactQueryStreamedHydration } from '@tanstack/react-query-next-experimental'

function makeQueryClient() {
  return new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 60 * 1000,
      },
    },
  })
}

let browserQueryClient: QueryClient | undefined = undefined

function getQueryClient() {
  if (isServer) {
    return makeQueryClient()
  } else {
    if (!browserQueryClient) browserQueryClient = makeQueryClient()
    return browserQueryClient
  }
}

export function Providers({ children }: { children: React.ReactNode }) {
  const queryClient = getQueryClient()
  return (
    <QueryClientProvider client={queryClient}>
      <ReactQueryStreamedHydration>
        {children}
      </ReactQueryStreamedHydration>
    </QueryClientProvider>
  )
}
```

For more details, see the NextJs Suspense Streaming Example and Advanced SSR guide.

Using `useQuery().promise` and `React.use()` (Experimental)

Set `experimental_prefetchInRender: true` in your `QueryClient`:

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      experimental_prefetchInRender: true,
    },
  },
})
```

Usage example:

```
import React from 'react'
import { useQuery } from '@tanstack/react-query'
import { fetchTodos, type Todo } from './api'

function TodoList({ query }: { query: UseQueryResult<Todo[]> }) {
  const data = React.use(query.promise)

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

export function App() {
  const query = useQuery({ queryKey: ['todos'], queryFn: fetchTodos })

  return (
    <>
      <h1>Todos</h1>
      <React.Suspense fallback={<div>Loading...</div>}>
        <TodoList query={query} />
      </React.Suspense>
    </>
  )
}
```

See the complete example on GitHub.

For full documentation, edit on GitHub.

Testing

React Query works by means of hooks - either the ones we offer or custom ones that wrap around them.

With React 17 or earlier, writing unit tests for these custom hooks can be done by means of the React Hooks Testing Library.

Install this by running:

```
npm install @testing-library/react-hooks react-test-renderer --save-dev
```

(The `react-test-renderer` library is needed as a peer dependency of `@testing-library/react-hooks`, and needs to correspond to the version of React that you are using.)

Note: when using React 18 or later, `renderHook` is available directly through the `@testing-library/react` package, and `@testing-library/react-hooks` is no longer required.

Our First Test

Once installed, a simple test can be written. Given the following custom hook:

```
export function useCustomHook() {
  return useQuery({ queryKey: ['customHook'], queryFn: () => 'Hello' })
}
```

We can write a test for this as follows:

```
import { renderHook, waitFor } from '@testing-library/react'
import { QueryClient, QueryClientProvider } from 'react-query'

const queryClient = new QueryClient()
const wrapper = ({ children }) => (
  <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>
)

const { result } = renderHook(() => useCustomHook(), { wrapper })

await waitFor(() => expect(result.current.isSuccess).toBe(true))

expect(result.current.data).toEqual('Hello')
```

Note: We provide a custom wrapper that builds the `QueryClient` and `QueryClientProvider`. This helps to ensure that our test is completely isolated from any other tests.

It is possible to write this wrapper only once, but if so we need to ensure that the `QueryClient` gets cleared before every test, and that tests don't run in parallel otherwise one test will influence the results of others.

Turn off retries

The library defaults to three retries with exponential backoff, which means that your tests are likely to timeout if you want to test an erroneous query. The easiest way to turn retries off is via the `QueryClientProvider`. Let's extend the above example:

```
const queryClient = new QueryClient({
  defaultOptions: {
```

```
    queries: {
      // ✅ turns retries off
      retry: false,
    },
  },
})
const wrapper = ({ children }) => (
  <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>
)
```

This will set the defaults for all queries in the component tree to "no retries". It is important to know that this will only work if your actual `useQuery` has no explicit retries set. If you have a query that wants 5 retries, this will still take precedence, because defaults are only taken as a fallback.

Set gcTime to Infinity with Jest

If you use Jest, you can set the `gcTime` to `Infinity` to prevent "Jest did not exit one second after the test run completed" error message. This is the default behavior on the server, and is only necessary to set if you are explicitly setting a `gcTime`.

Testing Network Calls

The primary use for React Query is to cache network requests, so it's important that we can test our code is making the correct network requests in the first place.

There are plenty of ways that these can be tested, but for this example we are going to use nock.

Given the following custom hook:

```
function useFetchData() {
  return useQuery({
    queryKey: ['fetchData'],
    queryFn: () => request('/api/data'),
  })
}
```

We can write a test for this as follows:

```
const queryClient = new QueryClient()
const wrapper = ({ children }) => (
  <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>
)

const expectation = nock('http://example.com').get('/api/data').reply(200, {
  answer: 42,
})

const { result } = renderHook(() => useFetchData(), { wrapper })

await waitFor(() => expect(result.current.isSuccess).toBe(true))

expect(result.current.data).toEqual({ answer: 42 })
```

Note: When using React 18, the semantics of `waitFor` have changed as noted above.

Testing Load More / Infinite Scroll

First, we need to mock our API response:

```
function generateMockedResponse(page) {
  return {
    page: page,
    items: [/* ... */]
  }
}
```

Then, our `nock` configuration needs to differentiate responses based on the page, and we'll be using `uri` to do this. `uri`'s value here will be something like `"/?page=1"` or `"/?page=2"`:

```
const expectation = nock('http://example.com')
  .persist()
  .query(true)
  .get('/api/data')
  .reply(200, (uri) => {
    const url = new URL(`http://example.com${uri}`)
    const { page } = Object.fromEntries(url.searchParams)
    return generateMockedResponse(page)
  })
```

(Notice the `.persist()`, because we'll be calling from this endpoint multiple times)

Now we can safely run our tests; the trick here is to await for the data assertion to pass:

```
const { result } = renderHook(() => useInfiniteQueryCustomHook(), {
  wrapper,
})

await waitFor(() => expect(result.current.isSuccess).toBe(true))

expect(result.current.data.pages).toStrictEqual(generateMockedResponse(1))

result.current.fetchNextPage()

await waitFor(() =>
  expect(result.current.data.pages).toStrictEqual([
    ...generateMockedResponse(1),
    ...generateMockedResponse(2),
  ]),
)

expectation.done()
```

Note: When using React 18, the semantics of `waitFor` have changed as noted above.

Further reading

For additional tips and an alternative setup using mock-service-worker, have a look at Testing React Query from the Community Resources.

Edit on GitHub

Does TanStack Query replace Redux, MobX or other global state managers?

A Contrived Example

Here we have some "global" state being managed by a global state library:

```
const globalState = {
  projects,
  teams,
  tasks,
  users,
  themeMode,
  sidebarStatus,
}
```

Currently, the global state manager is caching 4 types of server-state: `projects`, `teams`, `tasks`, and `users`. If we were to move these server-state assets to TanStack Query, our remaining global state would look more like this:

```
const globalState = {
  themeMode,
  sidebarStatus,
}
```

This also means that with a few hook calls to `useQuery` and `useMutation`, we also get to remove any boilerplate code that was used to manage our server state e.g.

- Connectors
- Action Creators
- Middlewares
- Reducers
- Loading/Error/Result states
- Contexts

With all of those things removed, you may ask yourself, **"Is it worth it to keep using our client state manager for this tiny global state?"**

And that's up to you!

But TanStack Query's role is clear. It removes asynchronous wiring and boilerplate from your application and replaces it with just a few lines of code.

What are you waiting for, give it a go already!

Migrating to React Query 3

Previous versions of React Query were awesome and brought some amazing new features, more magic, and an overall better experience to the library. They also brought on massive adoption and likewise a lot of refining fire (issues/contributions) to the library and brought to light a few things that needed more polish to make the library even better. v3 contains that very polish.

Overview

- More scalable and testable cache configuration
- Better SSR support
- Data-lag (previously usePaginatedQuery) anywhere!
- Bi-directional Infinite Queries
- Query data selectors!
- Fully configure defaults for queries and/or mutations before use
- More granularity for optional rendering optimization
- New *useQueries* hook! (Variable-length parallel query execution)
- Query filter support for the *useIsFetching()* hook!
- Retry/offline/replay support for mutations
- Observe queries/mutations outside of React
- Use the React Query core logic anywhere you want!
- Bundled/Colocated Devtools via *react-query/devtools*
- Cache Persistence to web storage (experimental via *react-query/persistQueryClient-experimental* and *react-query/createWebStoragePersistor-experimental*)

Breaking Changes

The `QueryCache` has been split into a `QueryClient` and lower-level `QueryCache` and `MutationCache` instances.

The `QueryCache` contains all queries, the `MutationCache` contains all mutations, and the `QueryClient` can be used to set configuration and to interact with them.

This has some benefits:

- Allows for different types of caches.
- Multiple clients with different configurations can use the same cache.
- Clients can be used to track queries, which can be used for shared caches on SSR.
- The client API is more focused towards general usage.
- Easier to test the individual components.

When creating a `new QueryClient()`, a `QueryCache` and `MutationCache` are automatically created for you if you don't supply them.

```
import { QueryClient } from 'react-query';

const queryClient = new QueryClient();
```

`ReactQueryConfigProvider` and `ReactQueryCacheProvider` have both been replaced by `QueryClientProvider`

Default options for queries and mutations can now be specified in `QueryClient`:

Notice that it's now `defaultOptions` instead of `defaultConfig`

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      // query options
    },
    mutations: {
      // mutation options
    },
  },
});
```

The `QueryClientProvider` component is now used to connect a `QueryClient` to your application:

```
import { QueryClient, QueryClientProvider } from 'react-query';

const queryClient = new QueryClient();

function App() {
  return <QueryClientProvider client={queryClient}>...</QueryClientProvider>;
}
```

The default `QueryCache` is gone. For real this time!

As previously noted with a deprecation, there is no longer a default `QueryCache` that is created or exported from the main package. **You must create your own via `new QueryClient()` or `new QueryCache()`** (which you can then pass to `new QueryClient({ queryCache })`).

```
import { QueryClient } from 'react-query';

const queryClient = new QueryClient();
```

`makeQueryCache` utility removed

It's been a long time coming, but it's finally gone :)

`QueryCache.prefetchQuery()` moved to `QueryClient.prefetchQuery()`

The new `QueryClient.prefetchQuery()` function is async but **does not return the data from the query.** If you need the data, use `fetchQuery()`:

```
// Prefetch a query:
await queryClient.prefetchQuery('posts', fetchPosts);

// Fetch a query:
try {
  const data = await queryClient.fetchQuery('posts', fetchPosts);
} catch (error) {
  // Error handling
}
```

`ReactQueryErrorResetBoundary` and `QueryCache.resetErrorBoundaries()` replaced by `QueryErrorResetBoundary` and `useQueryErrorResetBoundary()`

These provide the same experience but with more control. See:

- QueryErrorResetBoundary
- useQueryErrorResetBoundary

`QueryCache.getQuery()` replaced by `QueryCache.find()`

Use `QueryCache.find()` to look up individual queries.

`QueryCache.getQueries()` moved to `QueryCache.findAll()`

Use `QueryCache.findAll()` to look up multiple queries.

`QueryCache.isFetching` moved to `QueryClient.isFetching()`

Note: it's now a function, not a property.

`useQueryCache` replaced with `useQueryClient`

Returns the provided `queryClient` for the component tree, mainly a renaming.

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

Query key parts/pieces are no longer automatically spread to the query function

Inline functions are now recommended for passing parameters:

```
// Old:
useQuery(['post', id], (_key, id) => fetchPost(id));

// New:
useQuery(['post', id], () => fetchPost(id));
```

Alternatively, you can use the `QueryFunctionContext`:

```
useQuery(['post', id], (context) => fetchPost(context.queryKey[1]));
```

Infinite Query Page params're now passed via `QueryFunctionContext.pageParam`

Previously added as the last query key param, now passed explicitly:

```
// Old:
useInfiniteQuery(['posts'], (_key, pageParam = 0) => fetchPosts(pageParam));

// New:
useInfiniteQuery(['posts'], ({ pageParam = 0 }) => fetchPosts(pageParam));
```

`usePaginatedQuery()` removed in favor of `keepPreviousData` option

Use `keepPreviousData: true` with `useQuery` or `useInfiniteQuery`.

```
import { useQuery } from 'react-query';

function Page({ page }) {
  const { data } = useQuery(['page', page], fetchPage, {
    keepPreviousData: true,
  });
}
```

`useInfiniteQuery()` now supports bi-directional fetching

Options renamed and added:

- `getFetchMore` → `getNextPageParam`
- `queryResult.canFetchMore` → `queryResult.hasNextPage`
- `queryResult.fetchMore` → `queryResult.fetchNextPage`
- `queryResult.isFetchingMore` → `queryResult.isFetchingNextPage`

Additional properties:

- `getPreviousPageParam`
- `queryResult.hasPreviousPage`
- `queryResult.fetchPreviousPage`
- `queryResult.isFetchingPreviousPage`

Data now contains `pages` and `pageParams`:

```
const { pages, pageParams } = data;
```

Infinite Query data contains pages and pageParams

Manipulate data easily:

```
queryClient.setQueryData(['projects'], (data) => ({
  pages: data.pages.slice(1),
  pageParams: data.pageParams.slice(1),
}));
```

`useMutation` now returns an object, not an array

```
// Old:
const [mutate, { status, reset }] = useMutation();
```

```
// New:
const { mutate, status, reset } = useMutation();
```

`mutation.mutate` no longer returns a promise

`mutate` now accepts callbacks, and `mutateAsync()` returns a promise.

```
const { mutate, mutateAsync } = useMutation({ mutationFn: addTodo });
```

Using callbacks:

```
mutate('todo', {
  onSuccess: (data) => console.log(data),
  onError: (error) => console.error(error),
});
```

Using async/await:

```
try {
  const data = await mutateAsync('todo');
  console.log(data);
} catch (error) {
  console.error(error);
}
```

The object syntax for `useQuery` now uses a collapsed config

```
// Old:
useQuery({
  queryKey: 'posts',
  queryFn: fetchPosts,
  config: { staleTime: Infinity },
});

// New:
useQuery({
  queryKey: 'posts',
  queryFn: fetchPosts,
  staleTime: Infinity,
});
```

`QueryOptions.enabled` must be a boolean

`enabled` only disables a query when `false`. Casting examples:

```
// Correct:
useQuery(['user'], fetchUser, { enabled: !!userId });
```

`QueryOptions.initialStale` removed

Initial data is now treated as regular data. To avoid refetch on mount, define `staleTime`.

`QueryOptions.forceFetchOnMount` replaced with `refetchOnMount: 'always'`

```
useQuery(['user'], fetchUser, { refetchOnMount: 'always' });
```

`refetchOnMount` applies only to the component where set

Setting `refetchOnMount: false` in one component won't affect others.

`QueryOptions.queryFnParamsFilter` removed

Query functions now receive a `QueryFunctionContext`, parameters can be filtered inside the function.

`QueryOptions.notifyOnStatusChange` replaced

Use `notifyOnChangeProps` and `notifyOnChangePropsExclusions` for granular re-render control:

```
const { data } = useQuery(['user'], fetchUser, {
  notifyOnChangeProps: ['data', 'error'],
});
```

Or to prevent re-render when `isStale` changes:

```
const { data } = useQuery(['user'], fetchUser, {
  notifyOnChangePropsExclusions: ['isStale'],
});
```

`QueryResult.clear()` renamed to `QueryResult.remove()`

Removes the query from cache:

```
queryResult.remove();
```

`QueryResult.updatedAt` split into:

- `dataUpdatedAt`
- `errorUpdatedAt`

`setConsole()` replaced by `setLogger()`

```
import { setLogger } from 'react-query';

setLogger({
  error: (error) => {
    Sentry.captureException(error);
  },
});
```

React Native automatic logger

No more manual override needed. Errors are handled automatically.

TypeScript Changes

`QueryStatus` is now a union type, not an enum:

```
// Old:
import { QueryStatus } from 'react-query';

// New:
import { QueryStatus } from 'react-query';

// Usage:
if (status === 'loading') { ... }
```

Replace enum references:

Old	New
`QueryStatus.Idle`	`'idle'`
`QueryStatus.Loading`	`'loading'`
`QueryStatus.Error`	`'error'`
`QueryStatus.Success`	`'success'`

New features

Query Data Selectors

Transform the query result via `select`:

```
const { data } = useQuery(['user'], fetchUser, {
  select: (user) => user.username,
});
```

`useQueries()` hook for variable-length parallel queries

```
import { useQueries } from 'react-query';

function Overview() {
  const results = useQueries([
    { queryKey: ['post', 1], queryFn: fetchPost },
    { queryKey: ['post', 2], queryFn: fetchPost },
  ]);
  return (
    <ul>
      {results.map(({ data }) => data && <li key={data.id}>{data.title}</li>)}
    </ul>
  );
}
```

Retry offline mutations

```
const mutation = useMutation({
  mutationFn: addTodo,
  retry: 3,
});
```

Persist mutations

Mutations can be persisted to storage and resumed later. See docs.

QueryObserver, InfiniteQueryObserver, QueriesObserver

Create observers to watch queries outside React:

```
const observer = new QueryObserver(queryClient, { queryKey: 'posts' });

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

Similarly for InfiniteQueryObserver and QueriesObserver.

Set default options for specific queries and mutations

```
queryClient.setQueryDefaults(['posts'], { queryFn: fetchPosts });
queryClient.setMutationDefaults(['addPost'], { mutationFn: addPost });
```

`useIsFetching()` filter

```
const fetches = useIsFetching({ queryKey: ['posts'] });
```

Core separation

The core logic is now separated from React, usable standalone:

```
import { QueryClient } from 'react-query/core';
```

Devtools

Devtools are integrated into the main package:

```
// import { ReactQueryDevtools } from 'react-query/devtools'
```

Additional Info

Edit on GitHub

On this page

Migrating to React Query 4

Breaking Changes

v4 is a major version, so there are some breaking changes to be aware of:

react-query is now @tanstack/react-query

You will need to un-/install dependencies and change the imports:

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

TSX Example:

```
- import { useQuery } from 'react-query' // [!code --]
- import { ReactQueryDevtools } from 'react-query/devtools' // [!code --]

+ import { useQuery } from '@tanstack/react-query' // [!code ++]
+ import { ReactQueryDevtools } from '@tanstack/react-query-devtools' // [!code ++]
```

Codemod

To make the import migration easier, v4 comes with a codemod.

> The codemod is a best efforts attempt to help you migrate the breaking change. Please review the generated code thoroughly! There are edge cases that cannot be found by the code mod, so keep an eye on the log output.

You can apply it using:

For `.js` or `.jsx` files:

```
npx jscodeshift ./path/to/src/ \
  --extensions=js,jsx \
  --transform=./node_modules/@tanstack/react-query/codemods/v4/replace-import-specifie
```

For `.ts` or `.tsx` files:

```
npx jscodeshift ./path/to/src/ \
  --extensions=ts,tsx \
  --parser=tsx \
  --transform=./node_modules/@tanstack/react-query/codemods/v4/replace-import-specifie
```

> Note: Use `tsx` parser for TypeScript to ensure proper application. After applying, run Prettier or ESLint.

Query Keys (and Mutation Keys) need to be an Array

In v3, keys could be strings or arrays. Now, all keys must be arrays for consistency.

Example:

```
- useQuery('todos', fetchTodos) // [!code --]
+ useQuery(['todos'], fetchTodos) // [!code ++]
```

Codemod for easier migration:

```
npx jscodeshift ./path/to/src/ \
  --extensions=js,jsx \
  --transform=./node_modules/@tanstack/react-query/codemods/v4/key-transformation.js
```

> Note: Remember to review the code after running the codemod.

The idle state has been removed

Changes due to new `fetchStatus`. For example:

```
- status: 'idle' // [!code --]
+ status: 'loading'  // [!code ++]
+ fetchStatus: 'idle' // [!code ++]
```

Disables queries now start in `loading` but should check `isInitialLoading` instead of `isLoading`.

New API for `useQueries`

Now accepts an object with a `queries` array:

```
- useQueries([
-   { queryKey1, queryFn1, options1 },
-   { queryKey2, queryFn2, options2 },
- ]) // [!code --]
+ useQueries({
+   queries: [
+     { queryKey1, queryFn1, options1 },
+     { queryKey2, queryFn2, options2 },
+   ],
+ }) // [!code ++]
```

Undefined is an illegal cache value for successful queries

Disallows returning `undefined` as cache data. For example:

```
useQuery(['key'], () =>
  axios.get(url).then((result) => console.log(result.data))
)
```

This will now cause an error, transforming `undefined` into a failed Promise at runtime.

Queries and mutations per default need network connection to run

React Query now defaults to pausing queries/mutations offline, controlled via `networkMode: 'offlineFirst'`.

Example:

```
new QueryClient({
  defaultOptions: {
    queries: {
```

```
      networkMode: 'offlineFirst',
    },
    mutations: {
      networkMode: 'offlineFirst',
    },
  },
})
```

`notifyOnChangeProps` property no longer accepts `"tracked"` as a value

Defaults to tracking properties. To emulate v3 behavior, use `"all"`.

`notifyOnChangePropsExclusion` has been removed

Defaults have been aligned; this option is no longer needed.

Consistent behavior for `cancelRefetch`

- Applied to functions like `refetchQueries`, `invalidateQueries`, `resetQueries`, and methods from Hooks.
- Defaults now to `true` for all. For example:

```
queryClient.refetchQueries({ queryKey: ['todos'] })
// Aborts previous fetch and starts new
```

- To disable, pass `{ cancelRefetch: false }`.

Query Filters

Instead of `active` and `inactive` booleans, use:

```
type?: 'active' | 'inactive' | 'all' // [!code ++]
```

Defaults to `'all'`.

`refetchActive` / `refetchInactive`

Replaced with:

```
refetchType?: 'active' | 'inactive' | 'all' | 'none' // [!code ++]
```

Defaults:

Option	Default	Description
`active`	`true`	Refetch active queries
`inactive`	`false`	Refetch inactive queries

`onSuccess` is no longer called from `setQueryData`

Now tied to an actual request. To listen to data changes, use `useEffect`:

```
const { data } = useQuery({ queryKey, queryFn });
React.useEffect(() => mySideEffectHere(data), [data]);
```

`persistQueryClient` and related plugins are now production-ready

Renamed to:

- `createSyncStoragePersister`
- `createAsyncStoragePersister`

and imports updated accordingly.

The `<src/react>` directory renamed to `<src/reactjs>`

Update imports if your project directly accesses `'react-query/react'`:

```
- import { QueryClientProvider } from 'react-query/react'; // [!code --]
+ import { QueryClientProvider } from '@tanstack/react-query/reactjs'; // [!code ++]
```

New Features 🚀

Support for React 18

First-class React 18 support with concurrent features.

Proper offline support

New `networkMode` addresses offline scenarios, improving user experience when offline.

Tracked Queries per default

Default tracking for enhanced rendering efficiency.

Bailing out of updates with `setQueryData`

Return `undefined` from updater to prevent updates:

```
queryClient.setQueryData(['todo', id], (previousTodo) =>
  previousTodo ? { ...previousTodo, done: true } : undefined
)
```

Mutation Cache Garbage Collection

Mutations now garbage collected automatically after 5 minutes.

Custom Contexts for Multiple Providers

Example showcase:

```
// Data package with custom context
const context = React.createContext<QueryClient | undefined>(undefined);
const queryClient = new QueryClient();

export const useUser = () => {
  return useQuery(USER_KEY, USER_FETCHER, { context });
```

```
};

export const ContainerDataProvider = ({ children }: { children: React.ReactNode }) =>
  <QueryClientProvider client={queryClient} context={context}>
    {children}
  </QueryClientProvider>
);

// Usage in app:
<ContainerDataProvider>
  <AppDataProvider>
    <MyComponentDataProvider>
      <MyComponent />
    </MyComponentDataProvider>
  </AppDataProvider>
</ContainerDataProvider>
```

Additional Resources

Edit on GitHub

On this page

Migrating to TanStack Query v5 | TanStack Query React Docs

On this page

Breaking Changes

v5 is a major version, so there are some breaking changes to be aware of:

Supports a single signature, one object

useQuery and friends used to have many overloads in TypeScript, with different invocation styles. Now, only the object format is supported.

```
// Old overloads
useQuery(key, fn, options)
// Supported now
useQuery({ queryKey, queryFn, ...options })

// Similarly for other hooks:
useInfiniteQuery({ queryKey, queryFn, ...options })
useMutation({ mutationFn, ...options })
useIsFetching({ queryKey, ...filters })
useIsMutating({ mutationKey, ...filters })
```

queryClient.getQueryData now accepts queryKey only as an Argument

The argument has changed:

```
// Old
queryClient.getQueryData(queryKey, filters)
// New
queryClient.getQueryData(queryKey)
```

queryClient.getQueryState now accepts queryKey only as an Argument

```
// Old
queryClient.getQueryState(queryKey, filters)
// New
queryClient.getQueryState(queryKey)
```

Codemod

To aid migration, a codemod script is provided. Review the GitHub link for details.

Callbacks on useQuery and QueryObserver removed

`onSuccess`, `onError`, `onSettled` are no longer supported on queries (but remain on mutations). See RFC #5279.

refetchInterval callback only gets query

```
// Old
refetchInterval: (data, query) => ...
// Now
refetchInterval: (query) => ...
```

The `remove` method removed from useQuery

Previously used for imperative removal; replace with:

```
queryClient.removeQueries({ queryKey })
```

Minimum TypeScript version is 4.7

Type inference improvements require TS 4.7+.

isDataEqual option removed from useQuery

Replace with `structuralSharing`:

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

- isDataEqual: (oldData, newData) => check
+ structuralSharing: (oldData, newData) => check ? oldData : replaceEqualDeep(oldData,
```

Deprecated custom logger removed

Logging in development mode no longer supports custom loggers.

Supported Browsers

Updated browserslist for better performance and size. Details [here](https://...

Private class fields and methods

Now using ECMAScript private class features, ensuring true privacy at runtime.

Rename `cacheTime` to `gcTime`

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
-      cacheTime: 10 * MINUTE,
+      gcTime: 10 * MINUTE,
    },
  },
})
```

The `useErrorBoundary` option renamed to `throwOnError`

Reflects more accurate behavior.

Error type defaults to `Error`

```
useQuery<number, Error>({
  queryKey: ['key'],
  queryFn: fetchFn
})
```

ESLint rule `prefer-query-object-syntax` removed

Only object syntax supported.

`keepPreviousData` replaced with `placeholderData`

Use an identity function:

```
useQuery({
  queryKey,
  queryFn,
  placeholderData: (prevData, prevQuery) => prevData
})
```

Window focus refetching

Now listens to `visibilitychange` event only.

Network status

Starts with `online: true`. Listens to online/offline events.

Removed custom context prop

Use `queryClient` directly:

```
useQuery({ queryKey, queryFn }, queryClient)
```

Removed `refetchPage` in favor of `maxPages`

```
// Old
refetchPage: (lastPage, allPages) => ...
// New
maxPages: number
```

New `dehydrate` API

Replacing boolean options with functions like `shouldDehydrateQuery`.

Infinite queries need `initialPageParam`

```
useInfiniteQuery({
  queryFn: ({ pageParam }) => ...,
  initialPageParam: 0
})
```

Manual infinite queries mode removed

`getNextPageParam` is now required.

Returning `null` signals no more pages

```
// in getNextPageParam
return null; // indicates end
```

No retries on server

Default retry count is 0.

Status renamed: loading → pending, isLoading → isPending

```
// Before
status: 'loading'; isLoading: true
// After
status: 'pending'; isPending: true
```

hashQueryKey renamed to hashKey

May now hash mutation keys.

Minimum React version 18.0

Required due to `useSyncExternalStore`.

Removed `contextSharing` from `QueryClientProvider`

Share `queryClient` object directly.

No longer using `unstable_batchedUpdates`

If needed, set your batching function via `notifyManager.setBatchNotifyFunction()`.

Hydration API changes

Renamed `Hydrate` to `HydrationBoundary`; `useHydrate` removed; only query hydration occurs.

```
// Old
import { Hydrate } from '@tanstack/react-query'
<Hydrate state={dehydratedState}>...</Hydrate>

// New
import { HydrationBoundary } from '@tanstack/react-query'
<HydrationBoundary state={dehydratedState}>...</HydrationBoundary>
```

Query default options

`getQueryDefaults` now merges defaults with matching registrations. Register defaults from most generic to most specific.

```
// New
queryClient.setQueryDefaults(['todo'], { retry: false, staleTime: 60000 })
queryClient.setQueryDefaults(['todo', 'detail'], { retry: true, retryDelay: 1000, stal
```

New Features 🚀

Simplified optimistic updates

Use mutation variables directly:

```
const mutation = useMutation({
  mutationFn: (...),
  onSettled: () => queryClient.invalidateQueries(['todos'])
})

if (mutation.isPending) {
  // show optimistic UI with mutation.variables
}
```

Limited infinite queries with maxPages

Limits number of stored pages, improving memory and refetch speed.

```
useInfiniteQuery({ queryKey, getNextPageParam, maxPages: 3 })
```

Infinite queries can prefetch multiple pages

Use the `pages` option. See prefetching guide

New `combine` option for `useQueries`

Facilitates combining multiple queries with various configurations.

Experimental fine grained storage persister

Details in createPersister docs

Typesafe way to create query options

See TypeScript docs

New hooks for suspense

Available hooks:

```
const { data: post } = useSuspenseQuery({ queryKey: ['post', id], queryFn: fetchPost }
```

These hooks ensure `data` is never `undefined` at the type level.

Edit on GitHub

Query Options

Overview

One of the best ways to share `queryKey` and `queryFn` between multiple places, yet keep them co-located, is to use the `queryOptions` helper. At runtime, this helper just returns whatever you pass into it, but it has many advantages when using it with TypeScript. You can define all possible options for a query in one place, gaining type inference and safety for all of them.

Example Usage

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

function groupOptions(id: number) {
  return queryOptions({
    queryKey: ['groups', id],
    queryFn: () => fetchGroups(id),
    staleTime: 5 * 1000,
  })
}

// usage:

useQuery(groupOptions(1))
useSuspenseQuery(groupOptions(5))
useQueries({
  queries: [groupOptions(1), groupOptions(2)],
})
queryClient.prefetchQuery(groupOptions(23))
queryClient.setQueryData(groupOptions(42).queryKey, newGroups)
```

For **Infinite Queries**, a separate infiniteQueryOptions helper is available.

You can still override some options at the component level. A common pattern is to create per-component select functions:

```
// Type inference still works, so query.data will be the return type of select instead

const query = useQuery({
  ...groupOptions(1),
  select: (data) => data.groupName,
})
```

Edit on GitHub

Related Guides

- Query Functions
- Network Mode

Our Partners

Want to Skip the Docs?

Query.gg - The Official React Query Course
"*If you're serious about really understanding React Query, there's no better way than with query.gg*" — TANNER LINSLEY

Learn More

Related Resources

TanStack Form: Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

TanStack Config: The build and publish utilities used by all our projects. Use it if you dare!

Network Mode

TanStack Query provides three different network modes to distinguish how Queries and Mutations should behave if you have no network connection. This mode can be set for each Query / Mutation individually, or globally via the query / mutation defaults.

Since TanStack Query is most often used for data fetching in combination with data fetching libraries, the default network mode is online.

Network Mode: online

In this mode, Queries and Mutations will not fire unless you have network connection. This is the default mode. If a fetch is initiated for a query, it will always stay in the *state* (*pending*, *error*, *success*) it is in if the fetch cannot be made because there is no network connection. However, a fetchStatus is exposed additionally. This can be either:

- *fetching*: The `queryFn` is really executing - a request is in-flight.
- *paused*: The query is not executing - it is *paused* until you have connection again.
- *idle*: The query is not fetching and not paused.

The flags `isFetching`, `isPaused`, are derived from this state and exposed for convenience.

> Keep in mind that it might not be enough to check for *pending* state to show a loading spinner. Queries can be in *state: 'pending'*, but *fetchStatus: 'paused'* if they are mounting for the first time, and you have no network connection.

If a query runs because you are online, but you go offline while the fetch is still happening, TanStack Query will also pause the retry mechanism. Paused queries will then continue to run once you re-gain network connection. This is independent of `refetchOnReconnect` (which also defaults to `true` in this mode), because it is not a *refetch*, but rather a *continue*. If the query has been cancelled in the meantime, it will not continue.

Network Mode: always

In this mode, TanStack Query will always fetch and ignore the online / offline state. This is likely the mode you want to choose if you use TanStack Query in an environment where you don't need an active network connection for your Queries to work - e.g., if you just read from *AsyncStorage*, or if you just want to return `Promise.resolve(5)` from your `queryFn`.

- Queries will never be *paused* because you have no network connection.
- Retries will also not pause - your Query will go to *error* state if it fails.
- `refetchOnReconnect` defaults to `false` in this mode, because reconnecting to the network is not a good indicator anymore that stale queries should be refetched. You can still turn it on if you want.

Network Mode: offlineFirst

This mode is the middle ground between the first two options, where TanStack Query will run the `queryFn` once, but then pause retries. This is very handy if you have a service worker that intercepts a request for caching like in an offline-first PWA, or if you use HTTP caching via the Cache-Control header.

In those situations, the first fetch might succeed because it comes from an offline storage / cache. However, if there is a cache miss, the network request will go out and fail, in which case this mode behaves like an *online* query - pausing retries.

Devtools

The TanStack Query Devtools will show Queries in a *paused* state if they would be fetching, but there is no network connection. There is also a toggle button to *Mock offline behavior*. Please note that this button will *not* actually mess with your network connection (you can do that in the browser devtools), but it will set the OnlineManager in an offline state.

Signature

- `networkMode: 'online' | 'always' | 'offlineFirst'`
  - optional
  - defaults to `'online'`

Edit on GitHub

On this page

On this page

Parallel Queries

On this page

"Parallel" queries are queries that are executed in parallel, or at the same time so as to maximize fetching concurrency.

Manual Parallel Queries

Overview

When the number of parallel queries does not change, there is **no extra effort** to use parallel queries. Just use any number of TanStack Query's `useQuery` and `useInfiniteQuery` hooks side-by-side!

Example (TSX)

```
function App () {
  // The following queries will execute in parallel
  const usersQuery = useQuery({ queryKey: ['users'], queryFn: fetchUsers })
  const teamsQuery = useQuery({ queryKey: ['teams'], queryFn: fetchTeams })
  const projectsQuery = useQuery({ queryKey: ['projects'], queryFn: fetchProjects })
  ...
}
```

Note

> When using React Query in suspense mode, this pattern of parallelism does not work, since the first query would throw a promise internally and would suspend the component before the other queries run. To get around this, you'll either need to use the `useSuspenseQueries` hook (which is suggested) or orchestrate your own parallelism with separate components for each `useSuspenseQuery` instance.

Dynamic Parallel Queries with `useQueries`

Overview

If the number of queries you need to execute is changing from render to render, you cannot use manual querying since that would violate the rules of hooks. Instead, TanStack Query provides a `useQueries` hook, which you can use to dynamically execute as many queries in parallel as you'd like.

Usage

`useQueries` accepts an **options object** with a **queries key** whose value is an **array of query objects**. It returns an **array of query results**.

Example (TSX)

```
function App({ users }) {
  const userQueries = useQueries({
    queries: users.map((user) => {
      return {
        queryKey: ['user', user.id],
        queryFn: () => fetchUserById(user.id),
      }
    }),
  })
}
```

Edit on GitHub

On this page

Manual Parallel Queries

Overview

When the number of parallel queries does not change, there is **no extra effort** to use parallel queries. Just use any number of TanStack Query's `useQuery` and `useInfiniteQuery` hooks side-by-side!

```
function App () {
  // The following queries will execute in parallel
  const usersQuery = useQuery({ queryKey: ['users'], queryFn: fetchUsers })
  const teamsQuery = useQuery({ queryKey: ['teams'], queryFn: fetchTeams })
  const projectsQuery = useQuery({ queryKey: ['projects'], queryFn: fetchProjects })
  ...
}
```

> When using React Query in suspense mode, this pattern of parallelism does not work, since the first query would throw a promise internally and would suspend the component before the other queries run. To get around this, you'll either need to use the `useSuspenseQueries` hook (which is suggested) or orchestrate your own parallelism with separate components for each `useSuspenseQuery` instance.

Dynamic Parallel Queries with `useQueries`

Overview

If the number of queries you need to execute is changing from render to render, you cannot use manual querying since that would violate the rules of hooks. Instead, TanStack Query provides a `useQueries` hook, which you can use to dynamically execute as many queries in parallel as you'd like.

Usage

`useQueries` accepts an **options object** with a **queries key** whose value is an **array of query objects**. It returns an **array of query results**.

Example (TSX)

```
function App({ users }) {
  const userQueries = useQueries({
    queries: users.map((user) => {
      return {
        queryKey: ['user', user.id],
```

```
        queryFn: () => fetchUserById(user.id),
      }
    }),
  })
}
```

Edit on GitHub

Additional Links

- Network Mode
- Dependent Queries

Our Partners

https://www.speakeasy.com/product/react-query?utm_source=tanstack&utm_campaign=tanstack

Want to Skip the Docs?

Query.gg - The Official React Query Course

> "If you're serious about *really* understanding React Query, there's no better way than with query.gg"
> — Tanner Linsley

Learn More

More Resources

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!

Dependent Queries

On this page

useQuery dependent Query

Dependent (or serial) queries depend on previous ones to finish before they can execute. To achieve this, it's as easy as using the **enabled** option to tell a query when it is ready to run:

```
// Get the user
const { data: user } = useQuery({
  queryKey: ['user', email],
  queryFn: getUserByEmail,
})

const userId = user?.id

// Then get the user's projects
const {
  status,
  fetchStatus,
  data: projects,
} = useQuery({
  queryKey: ['projects', userId],
  queryFn: getProjectsByUser,
  // The query will not execute until the userId exists
  enabled: !!userId,
})
```

The `projects` query will start in:

```
status: 'pending'
isPending: true
fetchStatus: 'idle'
```

As soon as the `user` is available, the `projects` query will be **enabled** and will then transition to:

```
status: 'pending'
isPending: true
fetchStatus: 'fetching'
```

Once we have the projects, it will go to:

```
status: 'success'
isPending: false
fetchStatus: 'idle'
```

---

useQueries dependent Query

Dynamic parallel query - *useQueries* can depend on a previous query also, here's how to achieve this:

```
// Get the users' ids
const { data: userIds } = useQuery({
  queryKey: ['users'],
  queryFn: getUsersData,
  select: (users) => users.map((user) => user.id),
})

// Then get the users' messages
const usersMessages = useQueries({
  queries: userIds
    ? userIds.map((id) => {
        return {
          queryKey: ['messages', id],
          queryFn: () => getMessagesByUsers(id),
        }
      })
    : [], // if userIds is undefined, an empty array will be returned
})
```

Note: `useQueries` returns an **array of query results**.

A note about performance

Dependent queries by definition constitute a form of request waterfall, which hurts performance. If we pretend both queries take the same amount of time, doing them serially instead of in parallel always takes twice as much time, which is especially hurtful when it happens on a client that has high latency. If you can, it's always better to restructure the backend APIs so that both queries can be fetched in parallel, though that might not always be practically feasible.

In the example above, instead of first fetching `getUserByEmail` to be able to `getProjectsByUser`, introducing a new `getProjectsByUserEmail` query would flatten the waterfall.

Edit on GitHub

On this page

Background Fetching Indicators

On this page

Introduction

A query's *status* === *'pending'* state is sufficient enough to show the initial hard-loading state for a query, but sometimes you may want to display an additional indicator that a query is refetching in the background. To do this, queries also supply you with an `isFetching` boolean that you can use to show that it's in a fetching state, regardless of the state of the `status` variable:

```
function Todos() {
  const {
    status,
    data: todos,
    error,
    isFetching,
  } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodos,
  })

  return status === 'pending' ? (
    <span>Loading...</span>
  ) : status === 'error' ? (
    <span>Error: {error.message}</span>
  ) : (
    <>
      {isFetching ? <div>Refreshing...</div> : null}

      <div>
        {todos.map((todo) => (
          <Todo todo={todo} />
        ))}
      </div>
    </>
  )
}
```

Displaying Global Background Fetching Loading State

In addition to individual query loading states, if you would like to show a global loading indicator when **any** queries are fetching (including in the background), you can use the `useIsFetching` hook:

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

function GlobalLoadingIndicator() {
```

```
  const isFetching = useIsFetching()

  return isFetching ? (
    <div>Queries are fetching in the background...</div>
  ) : null
}
```

Edit on GitHub

Edit this page on GitHub

Window Focus Refetching | TanStack Query React Docs

On this page

Window Focus Refetching

If a user leaves your application and returns and the query data is stale, **TanStack Query automatically requests fresh data for you in the background**. You can disable this globally or per-query using the `refetchOnWindowFocus` option:

Disabling Globally

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      refetchOnWindowFocus: false, // default: true
    },
  },
})

function App() {
  return <QueryClientProvider client={queryClient}>... </QueryClientProvider>
}
```

Disabling Per-Query

```
useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
  refetchOnWindowFocus: false,
})
```

Custom Window Focus Event

In rare circumstances, you may want to manage your own window focus events that trigger TanStack Query to revalidate. To do this, TanStack Query provides a `focusManager.setEventListener` function that supplies you the callback that should be fired when the window is focused and allows you to set up your own events. When calling `focusManager.setEventListener`, the previously set handler is removed (which in most cases will be the default handler) and your new handler is used instead. For example, this is the default handler:

```
focusManager.setEventListener((handleFocus) => {
  // Listen to visibilitychange
  if (typeof window !== 'undefined' && window.addEventListener) {
    const visibilitychangeHandler = () => {
      handleFocus(document.visibilityState === 'visible')
    }
    window.addEventListener('visibilitychange', visibilitychangeHandler, false)
    return () => {
      // Be sure to unsubscribe if a new handler is set
      window.removeEventListener('visibilitychange', visibilitychangeHandler)
    }
  }
})
```

Managing Focus in React Native

Instead of event listeners on `window`, React Native provides focus information through the AppState module. You can use the AppState "change" event to trigger an update when the app state changes to "active":

```
import { AppState } from 'react-native'
import { focusManager } from '@tanstack/react-query'

function onAppStateChange(status: AppStateStatus) {
  if (Platform.OS !== 'web') {
    focusManager.setFocused(status === 'active')
  }
}

useEffect(() => {
  const subscription = AppState.addEventListener('change', onAppStateChange)
  return () => subscription.remove()
}, [])
```

Managing focus state

You can override or reset the default focus state programmatically:

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

// Override the default focus state
focusManager.setFocused(true)

// Fallback to the default focus check
focusManager.setFocused(undefined)
```

Edit on GitHub

Edit this page on GitHub

QueryClient | TanStack Query Docs

Overview

The **QueryClient** can be used to interact with a cache:

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

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: Infinity,
    },
  },
})

await queryClient.prefetchQuery({ queryKey: ['posts'], queryFn: fetchPosts })
```

Its available methods are:

- queryClient.fetchQuery
- queryClient.fetchInfiniteQuery
- queryClient.prefetchQuery
- queryClient.prefetchInfiniteQuery
- queryClient.getQueryData
- queryClient.ensureQueryData
- queryClient.ensureInfiniteQueryData
- queryClient.getQueriesData
- queryClient.setQueryData
- queryClient.getQueryState
- queryClient.setQueriesData
- queryClient.invalidateQueries
- queryClient.refetchQueries
- queryClient.cancelQueries
- queryClient.removeQueries
- queryClient.resetQueries
- queryClient.isFetching
- queryClient.isMutating
- queryClient.getDefaultOptions
- queryClient.setDefaultOptions
- queryClient.getQueryDefaults
- queryClient.setQueryDefaults
- queryClient.getMutationDefaults
- queryClient.setMutationDefaults
- queryClient.getQueryCache
- queryClient.getMutationCache
- queryClient.clear
- queryClient.resumePausedMutations

Options

- `queryCache?: QueryCache`
  Optional. The query cache this client is connected to.

- `mutationCache?: MutationCache`
  Optional. The mutation cache this client is connected to.
- `defaultOptions?: DefaultOptions`
  Optional. Define defaults for all queries and mutations using this queryClient. You can also define defaults to be used for hydration.

---

queryClient.fetchQuery

`fetchQuery` is an asynchronous method to fetch and cache a query. It resolves with the data or throws an error. Use `prefetchQuery` if you only want to fetch without handling the result.

If the query exists and isn't invalidated or older than `staleTime`, cached data is returned; otherwise, a fetch occurs.

```
try {
  const data = await queryClient.fetchQuery({ queryKey, queryFn })
} catch (error) {
  console.log(error)
}
```

Specify `staleTime` to only fetch when data is older:

```
try {
  const data = await queryClient.fetchQuery({
    queryKey,
    queryFn,
    staleTime: 10000,
  })
} catch (error) {
  console.log(error)
}
```

Options
Same as `useQuery`, with extra options like `enabled`, `refetchInterval`, `refetchOnWindowFocus`, etc.

Returns
`Promise<TData>`

---

queryClient.fetchInfiniteQuery

`fetchInfiniteQuery` fetches and caches an infinite query:

```
try {
  const data = await queryClient.fetchInfiniteQuery({ queryKey, queryFn })
  console.log(data.pages)
} catch (error) {
  console.log(error)
}
```

Same options as `fetchQuery`.

Returns
`Promise<InfiniteData<TData, TPageParam>>`

---

queryClient.prefetchQuery

Prefetches a query without returning data or throwing errors:

```
await queryClient.prefetchQuery({ queryKey, queryFn })
```

You can use with default `queryFn`:

```
await queryClient.prefetchQuery({ queryKey })
```

Options same as `fetchQuery`.

Returns a `Promise<void>`: resolves immediately if no fetch needed, or after fetch completes, without data or error.

---

queryClient.prefetchInfiniteQuery

Prefetches an infinite query:

```
await queryClient.prefetchInfiniteQuery({ queryKey, queryFn })
```

Same options as `fetchQuery`.

Returns a `Promise<void>`.

---

queryClient.getQueryData

Synchronous, gets cached data for a query or `undefined` if none.

```
const data = queryClient.getQueryData(queryKey)
```

Options: `queryKey: QueryKey`

Returns: data or `undefined`.

---

queryClient.ensureQueryData

Async, gets cached data or fetches if missing.

```
const data = await queryClient.ensureQueryData({ queryKey, queryFn })
```

Options: same as `fetchQuery`

Returns: `Promise<TData>`

---

queryClient.ensureInfiniteQueryData

Async, gets infinite query data or fetches:

```
const data = await queryClient.ensureInfiniteQueryData({
  queryKey,
```

```
  queryFn,
  initialPageParam,
  getNextPageParam,
})
```

Part of options same as `fetchInfiniteQuery`. Returns `Promise<InfiniteData<TData, TPageParam>>`.

---

queryClient.getQueriesData

Synchronous; returns cached data of multiple queries matching filters.

```
const data = queryClient.getQueriesData(filters)
```

Options: `filters: QueryFilters`

Returns: Array of `[queryKey: QueryKey, data: TQueryFnData | undefined]`

---

queryClient.setQueryData

Sync; update cache immediately. If query doesn't exist, it will be created.

```
queryClient.setQueryData(queryKey, updater)
```

`updater` can be data or a function `(oldData) => newData`.
Use immutability: **do not** mutate old data directly.

Options

- `queryKey: QueryKey`
- `updater: TQueryFnData | undefined | ((oldData: TQueryFnData | undefined) => TQueryFnData | undefined)`

Using an updater value

```
setQueryData(queryKey, newData)
```

Using an updater function

```
setQueryData(queryKey, (oldData) => newData)
```

Returns immediately with updated cache.

---

queryClient.getQueryState

Sync; get current state of a query or `undefined` if none.

```
const state = queryClient.getQueryState(queryKey)
console.log(state.dataUpdatedAt)
```

Options: `queryKey: QueryKey`

---

queryClient.setQueriesData

Sync; update multiple queries cache based on filter or partial key match.

```
queryClient.setQueriesData(filters, updater)
```

Same as `setQueryData` but for multiple.

Options

- `filters: QueryFilters`
- `updater: TQueryFnData | ((oldData: TQueryFnData | undefined) => TQueryFnData)`

---

queryClient.invalidateQueries

Invalidate and refetch specific queries, identified by key or predicate:

```
await queryClient.invalidateQueries({
  queryKey: ['posts'],
  exact,
  refetchType: 'active',
});
```

Options control which queries are invalidated and whether to refetch:

- `refetchType: 'active' | 'inactive' | 'all' | 'none'` (defaults to `'active'`)

Refetches happen via `refetchQueries`.

---

queryClient.refetchQueries

Refetch queries matching criteria:

```
// all queries:
await queryClient.refetchQueries()

// stale queries:
await queryClient.refetchQueries({ stale: true })

// partial match:
await queryClient.refetchQueries({ queryKey: ['posts'], type: 'active' })

// exact match:
await queryClient.refetchQueries({ queryKey: ['posts', 1], type: 'active', exact: true
```

Options: `filters: QueryFilters`, `options: RefetchOptions`

Returns a promise resolving after all refetches complete. Doesn't throw by default; set `throwOnError: true` to throw on errors.

---

queryClient.cancelQueries

Cancel outgoing queries based on key or predicate:

```
await queryClient.cancelQueries({ queryKey: ['posts'], exact: true })
```

Options: `filters: QueryFilters`

Returns: void.

---

queryClient.removeQueries

Remove queries from cache:

```
queryClient.removeQueries({ queryKey, exact: true })
```

Options: `filters: QueryFilters`

Returns: void.

---

queryClient.resetQueries

Reset queries to initial state:

```
queryClient.resetQueries({ queryKey, exact: true })
```

Resets data to initial or default, refetches if active.

Options: `filters: QueryFilters`, `options: ResetOptions` (e.g., `throwOnError`, `cancelRefetch`)

Returns: Promise that resolves when done.

---

queryClient.isFetching

Returns number of queries currently fetching, including background fetches.

```
if (queryClient.isFetching()) {
  console.log('At least one query is fetching!')
}
```

Use hook `useIsFetching` to subscribe in components.

Options: `filters: QueryFilters`

Returns: number of fetching queries.

---

queryClient.isMutating

Number of current mutations:

```
if (queryClient.isMutating()) {
  console.log('At least one mutation is fetching!')
}
```

Use hook `useIsMutating`.

Options: `filters: MutationFilters`

Returns: number of mutations.

---

queryClient.getDefaultOptions

Get default options for this client:

```
const defaultOptions = queryClient.getDefaultOptions()
```

---

queryClient.setDefaultOptions

Set default options:

```
queryClient.setDefaultOptions({
  queries: {
    staleTime: Infinity,
  },
})
```

---

queryClient.getQueryDefaults

Get defaults for specific queries:

```
const defaultOptions = queryClient.getQueryDefaults(['posts'])
```

Defaults are merged if multiple match.

---

queryClient.setQueryDefaults

Set default options for queries:

```
queryClient.setQueryDefaults(['posts'], { queryFn: fetchPosts })

function Component() {
  const { data } = useQuery({ queryKey: ['posts'] })
}
```

Order of registration affects merging. More specific defaults override more generic ones.

---

queryClient.getMutationDefaults

Get defaults for mutations:

```
const defaultOptions = queryClient.getMutationDefaults(['addPost'])
```

---

queryClient.setMutationDefaults

Set defaults for mutations:

```
queryClient.setMutationDefaults(['addPost'], { mutationFn: addPost })

function Component() {
  const { data } = useMutation({ mutationKey: ['addPost'] })
}
```

Order affects merging.

---

queryClient.getQueryCache

```
const queryCache = queryClient.getQueryCache()
```

---

queryClient.getMutationCache

```
const mutationCache = queryClient.getMutationCache()
```

---

queryClient.clear

Clear all caches:

```
queryClient.clear()
```

---

queryClient.resumePausedMutations

Resume paused mutations (e.g., due to network loss):

```
queryClient.resumePausedMutations()
```

Edit on GitHub

On this page

- queryClient.isFetching
- queryClient.isMutating
- queryClient.getDefaultOptions
- queryClient.setDefaultOptions
- queryClient.getQueryDefaults
- queryClient.setQueryDefaults
- queryClient.getMutationDefaults
- queryClient.setMutationDefaults
- queryClient.getQueryCache
- queryClient.getMutationCache
- queryClient.clear
- queryClient.resumePausedMutations

QueryCache | TanStack Query Docs

On this page

QueryCache

The **QueryCache** is the storage mechanism for TanStack Query. It stores all the data, meta information, and state of queries it contains.

Normally, you will not interact with the QueryCache directly and instead use the `QueryClient` for a specific cache.

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

const queryCache = new QueryCache({
  onError: (error) => {
    console.log(error)
  },
  onSuccess: (data) => {
    console.log(data)
  },
  onSettled: (data, error) => {
    console.log(data, error)
  },
})

const query = queryCache.find(['posts'])
```

Its available methods are:

- find
- findAll
- subscribe
- clear

Options

- `onError?: (error: unknown, query: Query) => void`
  - Optional
  - This function will be called if some query encounters an error.
- `onSuccess?: (data: unknown, query: Query) => void`
  - Optional
  - This function will be called if some query is successful.

- `onSettled?: (data: unknown | undefined, error: unknown | null, query: Query) => void`
  - Optional
  - This function will be called if some query is settled (either successful or errored).

---

queryCache.find

`find` is a slightly more advanced synchronous method that can be used to get an existing query instance from the cache. This instance contains **all** the state for the query and all of the instances and underlying guts of the query as well. If the query does not exist, `undefined` will be returned.

> **Note:** This is not typically needed for most applications, but can come in handy when needing more information about a query in rare scenarios (e.g., looking at `query.state.dataUpdatedAt` timestamp to decide whether a query is fresh enough to be used as an initial value)

```
const query = queryCache.find(queryKey)
```

Options

- `filters?: QueryFilters`
  Query Filters

Returns

- `Query`
  The query instance from the cache.

---

queryCache.findAll

`findAll` is an even more advanced synchronous method that can be used to get existing query instances from the cache that partially match the query key. If queries do not exist, an empty array will be returned.

> **Note:** This is not typically needed for most applications, but can come in handy when needing more information about queries in rare scenarios.

```
const queries = queryCache.findAll(queryKey)
```

Options

- `queryKey?: QueryKey`
  Query Keys
- `filters?: QueryFilters`
  Query Filters

Returns

- `Query[]`
  Query instances from the cache.

---

queryCache.subscribe

The `subscribe` method can be used to subscribe to the query cache as a whole and be informed of safe/known updates to the cache like query states changing or queries being updated, added, or removed.

```
const callback = (event) => {
  console.log(event.type, event.query)
}

const unsubscribe = queryCache.subscribe(callback)
```

Options

- `callback: (event: QueryCacheNotifyEvent) => void`
  This function will be called with the query cache any time it is updated via its tracked update mechanisms (e.g., `query.setState`, `queryClient.removeQueries`, etc). Out of scope mutations to the cache are not encouraged and will not fire subscription callbacks.

Returns

- `unsubscribe: Function`
  This function will unsubscribe the callback from the query cache.

---

queryCache.clear

The `clear` method can be used to clear the cache entirely and start fresh.

```
queryCache.clear()
```

Further reading

To get a better understanding of how the QueryCache works internally, see #18: Inside React Query from the Community Resources.

Edit on GitHub

useQuery | TanStack Query React Docs

Overview

TanStack Query v5
Auto
Framework

React
Version
Latest
Search...
+ K
Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview
  react
- Installation
  react
- Quick Start
  react
- Devtools
  react
- Videos & Talks
  react
- Comparison
  react
- TypeScript
  react
- GraphQL
  react
- React Native
  react

Guides & Concepts

- Important Defaults
  react
- Queries
  react
- Query Keys
  react
- Query Functions
  react
- Query Options
  react

- Network Mode
  react
- Parallel Queries
  react
- Dependent Queries
  react
- Background Fetching Indicators
  react
- Window Focus Refetching
  react
- Disabling/Pausing Queries
  react
- Query Retries
  react
- Paginated Queries
  react
- Infinite Queries
  react
- Initial Query Data
  react
- Placeholder Query Data
  react
- Mutations
  react
- Query Invalidation
  react
- Invalidation from Mutations
  react
- Updates from Mutation Responses
  react
- Optimistic Updates
  react
- Query Cancellation
  react
- Scroll Restoration
  react
- Filters
  react
- Performance & Request Waterfalls
  react
- Prefetching & Router Integration
  react
- Server Rendering & Hydration
  react
- Advanced Server Rendering
  react
- Caching
  react
- Render Optimizations
  react
- Default Query Fn
  react
- Suspense
  react
- Testing
  react
- Does this replace [Redux, MobX, etc]?
  react

- Migrating to v3
  react
- Migrating to v4
  react
- Migrating to v5
  react

API Reference

- QueryClient
  core
- QueryCache
  core
- MutationCache
  core
- QueryObserver
  core
- InfiniteQueryObserver
  core
- QueriesObserver
  core
- streamedQuery
  core
- focusManager
  core
- onlineManager
  core
- notifyManager
  core
- useQuery
  react
- useQueries
  react
- useInfiniteQuery
  react
- useMutation
  react
- useIsFetching
  react
- useIsMutating
  react
- useMutationState
  react
- useSuspenseQuery
  react
- useSuspenseInfiniteQuery
  react
- useSuspenseQueries
  react
- QueryClientProvider
  react
- useQueryClient
  react
- queryOptions
  react
- infiniteQueryOptions
  react
- usePrefetchQuery
  react

- usePrefetchInfiniteQuery
  react
- QueryErrorResetBoundary
  react
- useQueryErrorResetBoundary
  react
- hydration
  react

ESLint

- ESLint Plugin Query
  core
- Exhaustive Deps
  core
- Stable Query Client
  core
- No Rest Destructuring
  core
- No Unstable Deps
  core
- Infinite Query Property Order
  core

Community Resources

- TkDodo's Blog
  react
- Community Projects
  react

Examples

- Simple
  react
- Basic
  react
- Basic w/ GraphQL-Request
  react
- Auto Refetching / Polling / Realtime
  react
- Optimistic Updates (UI)
  react
- Optimistic Updates (Cache)
  react
- Pagination
  react
- Load-More & Infinite Scroll
  react
- Infinite query with Max pages
  react
- Suspense
  react
- Default Query Function
  react
- Playground
  react
- Prefetching
  react
- Star Wars

react
- Rick And Morty
  react
- Next.js Pages
  react
- Next.js app with prefetching
  react
- Next.js app with streaming
  react
- React Native
  react
- React Router
  react
- Offline Queries and Mutations
  react
- Algolia
  react
- Shadow DOM
  react
- Devtools Embedded Panel
  react
- Chat example (streaming)
  react

Plugins

- persistQueryClient
  react
- createSyncStoragePersister
  react
- createAsyncStoragePersister
  react
- broadcastQueryClient (Experimental)
  react
- createPersister (Experimental)
  react

useQuery

tsx

```
const {
  data,
  dataUpdatedAt,
  error,
  errorUpdatedAt,
  failureCount,
  failureReason,
  fetchStatus,
  isError,
  isFetched,
  isFetchedAfterMount,
  isFetching,
  isInitialLoading,
  isLoading,
  isLoadingError,
  isPaused,
  isPending,
```

```
  isPlaceholderData,
  isRefetchError,
  isRefetching,
  isStale,
  isSuccess,
  promise,
  refetch,
  status,
} = useQuery(
  {
    queryKey,
    queryFn,
    gcTime,
    enabled,
    networkMode,
    initialData,
    initialDataUpdatedAt,
    meta,
    notifyOnChangeProps,
    placeholderData,
    queryKeyHashFn,
    refetchInterval,
    refetchIntervalInBackground,
    refetchOnMount,
    refetchOnReconnect,
    refetchOnWindowFocus,
    retry,
    retryOnMount,
    retryDelay,
    select,
    staleTime,
    structuralSharing,
    subscribed,
    throwOnError,
  },
  queryClient,
)

const {
  data,
  dataUpdatedAt,
  error,
  errorUpdatedAt,
  failureCount,
  failureReason,
  fetchStatus,
  isError,
  isFetched,
  isFetchedAfterMount,
  isFetching,
  isInitialLoading,
  isLoading,
  isLoadingError,
  isPaused,
  isPending,
  isPlaceholderData,
  isRefetchError,
  isRefetching,
```

```
  isStale,
  isSuccess,
  promise,
  refetch,
  status,
} = useQuery(
  {
    queryKey,
    queryFn,
    gcTime,
    enabled,
    networkMode,
    initialData,
    initialDataUpdatedAt,
    meta,
    notifyOnChangeProps,
    placeholderData,
    queryKeyHashFn,
    refetchInterval,
    refetchIntervalInBackground,
    refetchOnMount,
    refetchOnReconnect,
    refetchOnWindowFocus,
    retry,
    retryOnMount,
    retryDelay,
    select,
    staleTime,
    structuralSharing,
    subscribed,
    throwOnError,
  },
  queryClient,
)
```

Parameter1 (Options)

- queryKey: unknown[]
  - **Required**
  - The query key to use for this query.
  - The query key will be hashed into a stable hash. See Query Keys for more information.
  - The query will automatically update when this key changes (as long as enabled is not set to false).
- queryFn: (context: QueryFunctionContext) => Promise<TData>
  - **Required, but only if no default query function has been defined** See Default Query Function for more information.
  - The function that the query will use to request data.
  - Receives a QueryFunctionContext
  - Must return a promise that will either resolve data or throw an error. The data cannot be undefined.
- enabled: boolean | (query: Query) => boolean
  - Set this to false to disable this query from automatically running.
  - Can be used for Dependent Queries.
- networkMode: 'online' | 'always' | 'offlineFirst'
  - optional
  - defaults to 'online'
  - see Network Mode for more information.
- retry: boolean | number | (failureCount: number, error: TError) => boolean
  - If false, failed queries will not retry by default.

    - If true, failed queries will retry infinitely.
    - If set to a number, e.g. 3, failed queries will retry until the failed query count meets that number.
    - defaults to 3 on the client and 0 on the server
- retryOnMount: boolean
    - If set to false, the query will not be retried on mount if it contains an error. Defaults to true.
- retryDelay: number | (retryAttempt: number, error: TError) => number
    - This function receives a retryAttempt integer and the actual Error and returns the delay to apply before the next attempt in milliseconds.
    - A function like attempt => Math.min(attempt > 1 ? 2 ** attempt * 1000 : 1000, 30 * 1000) applies exponential backoff.
    - A function like attempt => attempt * 1000 applies linear backoff.
- staleTime: number | 'static' ((query: Query) => number | 'static')
    - Optional
    - Defaults to 0
    - The time in milliseconds after which data is considered stale. This value only applies to the hook it is defined on.
    - If set to Infinity, the data will not be considered stale unless manually invalidated
    - If set to a function, the function will be executed with the query to compute a staleTime.
    - If set to 'static', the data will never be considered stale
- gcTime: number | Infinity
    - Defaults to 5 * 60 * 1000 (5 minutes) or Infinity during SSR
    - The time in milliseconds that unused/inactive cache data remains in memory. When a query's cache becomes unused or inactive, that cache data will be garbage collected after this duration. When different garbage collection times are specified, the longest one will be used.
    - Note: the maximum allowed time is about 24 days. See more.
    - If set to Infinity, will disable garbage collection
- queryKeyHashFn: (queryKey: QueryKey) => string
    - Optional
    - If specified, this function is used to hash the queryKey to a string.
- refetchInterval: number | false | ((query: Query) => number | false | undefined)
    - Optional
    - If set to a number, all queries will continuously refetch at this frequency in milliseconds
    - If set to a function, the function will be executed with the query to compute a frequency
- refetchIntervalInBackground: boolean
    - Optional
    - If set to true, queries that are set to continuously refetch with a refetchInterval will continue to refetch while their tab/window is in the background
- refetchOnMount: boolean | "always" | ((query: Query) => boolean | "always")
    - Optional
    - Defaults to true
    - If set to true, the query will refetch on mount if the data is stale.
    - If set to false, the query will not refetch on mount.
    - If set to "always", the query will always refetch on mount (except when staleTime: 'static' is used).
    - If set to a function, the function will be executed with the query to compute the value
- refetchOnWindowFocus: boolean | "always" | ((query: Query) => boolean | "always")
    - Optional
    - Defaults to true
    - If set to true, the query will refetch on window focus if the data is stale.
    - If set to false, the query will not refetch on window focus.
    - If set to "always", the query will always refetch on window focus (except when staleTime: 'static' is used).
    - If set to a function, the function will be executed with the query to compute the value
- refetchOnReconnect: boolean | "always" | ((query: Query) => boolean | "always")
    - Optional
    - Defaults to true
    - If set to true, the query will refetch on reconnect if the data is stale.
    - If set to false, the query will not refetch on reconnect.

    - If set to "always", the query will always refetch on reconnect (except when staleTime: 'static' is used).
    - If set to a function, the function will be executed with the query to compute the value
- notifyOnChangeProps: string[] | "all" | (() => string[] | "all" | undefined)
    - Optional
    - If set, the component will only re-render if any of the listed properties change.
    - If set to ['data', 'error'] for example, the component will only re-render when the data or error properties change.
    - If set to "all", the component will opt-out of smart tracking and re-render whenever a query is updated.
    - If set to a function, the function will be executed to compute the list of properties.
    - By default, access to properties will be tracked, and the component will only re-render when one of the tracked properties change.
- select: (data: TData) => unknown
    - Optional
    - This option can be used to transform or select a part of the data returned by the query function. It affects the returned data value, but does not affect what gets stored in the query cache.
    - The select function will only run if data changed, or if the reference to the select function itself changes. To optimize, wrap the function in useCallback.
- initialData: TData | () => TData
    - Optional
    - If set, this value will be used as the initial data for the query cache (as long as the query hasn't been created or cached yet)
    - If set to a function, the function will be called **once** during the shared/root query initialization, and be expected to synchronously return the initialData
    - Initial data is considered stale by default unless a staleTime has been set.
    - initialData **is persisted** to the cache
- initialDataUpdatedAt: number | (() => number | undefined)
    - Optional
    - If set, this value will be used as the time (in milliseconds) of when the initialData itself was last updated.
- placeholderData: TData | (previousValue: TData | undefined; previousQuery: Query | undefined,) => TData
    - Optional
    - If set, this value will be used as the placeholder data for this particular query observer while the query is still in the pending state.
    - placeholderData is **not persisted** to the cache
    - If you provide a function for placeholderData, as a first argument you will receive previously watched query data if available, and the second argument will be the complete previousQuery instance.
- structuralSharing: boolean | (oldData: unknown | undefined, newData: unknown) => unknown)
    - Optional
    - Defaults to true
    - If set to false, structural sharing between query results will be disabled.
    - If set to a function, the old and new data values will be passed through this function, which should combine them into resolved data for the query. This way, you can retain references from the old data to improve performance even when that data contains non-serializable values.
- subscribed: boolean
    - Optional
    - Defaults to true
    - If set to false, this instance of useQuery will not be subscribed to the cache. This means it won't trigger the queryFn on its own, and it won't receive updates if data gets into cache by other means.
- throwOnError: undefined | boolean | (error: TError, query: Query) => boolean
    - Set this to true if you want errors to be thrown in the render phase and propagate to the nearest error boundary
    - Set this to false to disable suspense's default behavior of throwing errors to the error boundary.
    - If set to a function, it will be passed the error and the query, and it should return a boolean indicating whether to show the error in an error boundary (true) or return the error as state (false)

- meta: Record<string, unknown>
  - Optional
  - If set, stores additional information on the query cache entry that can be used as needed. It will be accessible wherever the query is available, and is also part of the QueryFunctionContext provided to the queryFn.

Parameter2 (QueryClient)

- queryClient?: QueryClient
  - Use this to use a custom QueryClient. Otherwise, the one from the nearest context will be used.

Returns

- status: QueryStatus
  - Will be:
    - pending if there's no cached data and no query attempt was finished yet.
    - error if the query attempt resulted in an error. The corresponding error property has the error received from the attempted fetch
    - success if the query has received a response with no errors and is ready to display its data. The corresponding data property on the query is the data received from the successful fetch or if the query's enabled property is set to false and has not been fetched yet data is the first initialData supplied to the query on initialization.
- isPending: boolean
  - A derived boolean from the status variable above, provided for convenience.
- isSuccess: boolean
  - A derived boolean from the status variable above, provided for convenience.
- isError: boolean
  - A derived boolean from the status variable above, provided for convenience.
- isLoadingError: boolean
  - Will be true if the query failed while fetching for the first time.
- isRefetchError: boolean
  - Will be true if the query failed while refetching.
- data: TData
  - Defaults to undefined.
  - The last successfully resolved data for the query.
- dataUpdatedAt: number
  - The timestamp for when the query most recently returned the status as "success".
- error: null | TError
  - Defaults to null
  - The error object for the query, if an error was thrown.
- errorUpdatedAt: number
  - The timestamp for when the query most recently returned the status as "error".
- isStale: boolean
  - Will be true if the data in the cache is invalidated or if the data is older than the given staleTime.
- isPlaceholderData: boolean
  - Will be true if the data shown is the placeholder data.
- isFetched: boolean
  - Will be true if the query has been fetched.
- isFetchedAfterMount: boolean
  - Will be true if the query has been fetched after the component mounted.
  - This property can be used to not show any previously cached data.
- fetchStatus: FetchStatus
  - fetching: Is true whenever the queryFn is executing, which includes initial pending as well as background refetches.
  - paused: The query wanted to fetch, but has been paused.
  - idle: The query is not fetching.

    - see Network Mode for more information.
- isFetching: boolean
    - A derived boolean from the fetchStatus variable above, provided for convenience.
- isPaused: boolean
    - A derived boolean from the fetchStatus variable above, provided for convenience.
- isRefetching: boolean
    - Is true whenever a background refetch is in-flight, which *does not* include initial pending
    - Is the same as isFetching && !isPending
- isLoading: boolean
    - Is true whenever the first fetch for a query is in-flight
    - Is the same as isFetching && isPending
- isInitialLoading: boolean
    - **deprecated**
    - An alias for isLoading, will be removed in the next major version.
- failureCount: number
    - The failure count for the query.
    - Incremented every time the query fails.
    - Reset to 0 when the query succeeds.
- failureReason: null | TError
    - The failure reason for the query retry.
    - Reset to null when the query succeeds.
- errorUpdateCount: number
    - The sum of all errors.
- refetch: (options: { throwOnError: boolean, cancelRefetch: boolean }) => Promise<UseQueryResult>
    - A function to manually refetch the query.
    - If the query errors, the error will only be logged. If you want an error to be thrown, pass the throwOnError: true option
    - cancelRefetch?: boolean
        - Defaults to true
            - Per default, a currently running request will be cancelled before a new request is made
        - When set to false, no refetch will be made if there is already a request running.
- promise: Promise<TData>
    - A stable promise that will be resolved with the data of the query.
    - Requires the experimental_prefetchInRender feature flag to be enabled on the QueryClient.

Edit on GitHub

useQueries | TanStack Query React Docs

On this page

---

useQueries

The `useQueries` hook can be used to fetch a variable number of queries:

```
const ids = [1, 2, 3]
const results = useQueries({
  queries: ids.map((id) => ({
    queryKey: ['post', id],
    queryFn: () => fetchPost(id),
    staleTime: Infinity,
  })),
})
```

Options

The `useQueries` hook accepts an options object with a **queries** key whose value is an array of query option objects, identical to the useQuery hook (excluding the `queryClient` option, as the QueryClient can be passed in on the top level).

- `queryClient?: QueryClient`
  - Use this to provide a custom QueryClient. Otherwise, the one from the nearest context will be used.
- `combine?: (result: UseQueriesResults) => TCombinedResult`
  - Use this to combine the results of the queries into a single value.

  Having the same query key more than once in the array of query objects may cause some data to be shared between queries. To avoid this, consider de-duplicating the queries and mapping the results back to the desired structure.

placeholderData

The `placeholderData` option exists for `useQueries` as well, but it doesn't get information passed from previously rendered Queries like `useQuery` does, because the input to `useQueries` can vary each render.

Returns

The `useQueries` hook returns an array with all the query results, in the same order as the input.

Combine

If you want to combine `data` (or other Query information) from the results into a single value, you can use the `combine` option. The combined result will be as referentially stable as possible.

```
const ids = [1, 2, 3]
const combinedQueries = useQueries({
  queries: ids.map((id) => ({
    queryKey: ['post', id],
    queryFn: () => fetchPost(id),
  })),
  combine: (results) => ({
    data: results.map((result) => result.data),
    pending: results.some((result) => result.isPending),
  }),
})
```

In this example, `combinedQueries` will be an object with `data` and `pending` properties. Note that other properties of individual query results will be lost.

Memoization

The `combine` function will only re-run if:

- The `combine` function itself changed referentially
- Any of the query results changed

An inline `combine` function, as shown above, will run on every render. To prevent this, wrap the `combine` function in `useCallback`, or define it as a stable function reference if it has no dependencies.

Related hooks

- useQuery
- useInfiniteQuery

Our Partners

useInfiniteQuery | TanStack Query React Docs

Example

```
const {
  fetchNextPage,
  fetchPreviousPage,
  hasNextPage,
  hasPreviousPage,
  isFetchingNextPage,
  isFetchingPreviousPage,
  promise,
  ...result
} = useInfiniteQuery({
  queryKey,
  queryFn: ({ pageParam }) => fetchPage(pageParam),
  initialPageParam: 1,
  ...options,
  getNextPageParam: (lastPage, allPages, lastPageParam, allPageParams) =>
    lastPage.nextCursor,
  getPreviousPageParam: (firstPage, allPages, firstPageParam, allPageParams) =>
    firstPage.prevCursor,
});
```

Options

The options for **useInfiniteQuery** are identical to the useQuery hook with the following additions:

- **queryFn: (context: QueryFunctionContext) => Promise**
  - **Required**, if no default query function has been defined defaultQueryFn
  - The function to request data.
  - Receives a QueryFunctionContext.
  - Must return a promise resolving data or throwing an error.
- **initialPageParam: TPageParam**
  - **Required**
  - The default page param for the first page.
- **getNextPageParam: (lastPage, allPages, lastPageParam, allPageParams) => TPageParam | undefined | null**
  - **Required**
  - Receives the last page of data and all pages, as well as pageParam info.
  - Returns a single value passed to the query function as the last parameter.
  - Return `undefined` or `null` if no next page.
- **getPreviousPageParam: (firstPage, allPages, firstPageParam, allPageParams) => TPageParam | undefined | null**

- Similar to above, but for previous pages.

- **maxPages: number | undefined**
  - Limits the number of pages stored in data.
  - When reached, older pages are removed based on direction.
  - `undefined` or `0` for unlimited.
  - Default: `undefined`.
  - `getNextPageParam` and `getPreviousPageParam` must be properly defined if `maxPages` > 0.

Returns

The properties returned by **useInfiniteQuery** are similar to useQuery, with additional properties and some differences:

- **data.pages: TData[]**
  - Array of all pages.
- **data.pageParams: unknown[]**
  - Array of all page parameters.
- **isFetchingNextPage: boolean**
  - `true` while fetching next page via `fetchNextPage()`.
- **isFetchingPreviousPage: boolean**
  - `true` while fetching previous page via `fetchPreviousPage()`.
- **fetchNextPage: (options?: FetchNextPageOptions) => Promise**
  - Fetches next page.
  - Options include `cancelRefetch`.
- **fetchPreviousPage: (options?: FetchPreviousPageOptions) => Promise**
  - Fetches previous page.
- **hasNextPage: boolean**
  - Indicates available next page.
- **hasPreviousPage: boolean**
  - Indicates available previous page.
- **isFetchNextPageError: boolean**
  - True if next page fetch failed.
- **isFetchPreviousPageError: boolean**
  - True if previous page fetch failed.
- **isRefetching: boolean**
  - True if in background refetch (excludes initial and next/previous fetches).
  - Same as `isFetching && !isPending && !isFetchingNextPage && !isFetchingPreviousPage`.

- **isRefetchError: boolean**
  - True if refetch failed.
- **promise: Promise**
  - Resolves to query result. Can be used with React's `use()`.

Note: Imperative fetch calls should be used carefully, only in response to user actions or with conditions like `hasNextPage && !isFetching`.

Edit on GitHub

Edit on GitHub

Additional hooks

- useQueries
- useMutation

Our Partners

- React Query Course by Query.gg

More Resources

Want to Skip the Docs?
Query.gg - The Official React Query Course
"If you're serious about really understanding React Query, there's no better way than with query.gg" — Tanner Linsley

Learn more at query.gg

TanStack Products

- TanStack Table: Supercharge your tables or build a data grid for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, Lit.
- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations.
  Learn more on the respective pages.

useMutation | TanStack Query React Docs

Example Usage

```
const {
  data,
  error,
  isError,
  isIdle,
  isPending,
  isPaused,
  isSuccess,
  failureCount,
  failureReason,
  mutate,
  mutateAsync,
  reset,
  status,
  submittedAt,
  variables,
} = useMutation(
  {
    mutationFn,
    gcTime,
    meta,
    mutationKey,
    networkMode,
    onError,
    onMutate,
    onSettled,
    onSuccess,
    retry,
    retryDelay,
    scope,
    throwOnError,
  },
  queryClient,
)

mutate(variables, {
  onError,
  onSettled,
  onSuccess,
})
```

Parameter1 (Options)

- **mutationFn:** *(variables: TVariables) => Promise<TData>*
  Required, if no default mutation function is set
  Function performing an async task, returning a promise.
  `variables` is passed to `mutationFn`.

- **gcTime:** `number | Infinity`
  Time in ms for unused cache data to remain before GC.
  Default: Long durations, or `Infinity` to disable GC.
  Note: max delay ~24 days. More.

- **mutationKey:** `unknown[]`
  Optional, inherits defaults via `setMutationDefaults`.

- **networkMode:** `'online' | 'always' | 'offlineFirst'`
  Optional, defaults to `'online'`.
  More on Network Mode.

- **onMutate:** `(variables: TVariables) => Promise<TContext | void> | TContext | void`
  Optional, fires before mutation, useful for optimistic updates.
  Return value can help rollback if mutation fails.

- **onSuccess:** `(data: TData, variables: TVariables, context: TContext) => Promise<unknown> | unknown`
  Optional, fires on success of mutation.

- **onError:** `(err: TError, variables: TVariables, context?: TContext) => Promise<unknown> | unknown`
  Optional, fires on error.

- **onSettled:** `(data: TData, error: TError, variables: TVariables, context?: TContext) => Promise<unknown> | unknown`
  Optional, fires after success or error.

- **retry:** `boolean | number | (failureCount: number, error: TError) => boolean`
  Defaults to `0`.

  - `false` disables retries
  - `true` retries infinitely
  - Number specifies max retries

- **retryDelay:** `number | (retryAttempt: number, error: TError) => number`
  Delay before retry in ms.
  e.g., exponential or linear backoff functions.

- **scope:** `{ id: string }`
  Optional, defaults to unique id, mutations with same id run in series.

- **throwOnError:** `undefined | boolean | (error: TError) => boolean`
  Controls whether errors are thrown or handled internally.

- **meta:** `Record<string, unknown>`
  Additional info stored on mutation cache.

Parameter2 (QueryClient)

- **queryClient?: QueryClient**
  Use a custom QueryClient; otherwise, from context.

Returns

- **mutate:** `(variables: TVariables, { onSuccess, onSettled, onError }) => void`
  Trigger mutation.
  - **variables:** *Optional*, passed to `mutationFn`
  - **onSuccess:** *(data, variables, context) => void*
    Fires on success; ignore return value.
  - **onError:** *(err, variables, context) => void*
    Fires on error; ignore return.
  - **onSettled:** *(data, error, variables, context) => void*
    Fires after success/error; ignore return.
- **mutateAsync:** `(variables, { onSuccess, onSettled, onError }) => Promise<TData>`
  Same as `mutate` but returns a promise.
- **status:** `string`
  - `'idle'`: before execution
  - `'pending'`: executing
  - `'error'`: error occurred
  - `'success'`: successful
- **isIdle**, **isPending**, **isSuccess**, **isError:** boolean, derived from `status`
- **isPaused:** boolean
  True if paused (see Network Mode)
- **data:** `undefined | unknown`
  Last successful data.
- **error:** `null | TError`
  Error object if any.
- **reset:** `() => void`
  Reset internal mutation state.
- **failureCount:** `number`
  Count of failed attempts.
- **failureReason:** `null | TError`
  Reason for last failure.
- **submittedAt:** `number`
  Timestamp when submitted.
- **variables:** `undefined | TVariables`
  Variables passed to mutation.

---

useIsFetching | TanStack Query React Docs

useIsFetching

Description

`useIsFetching` is an optional hook that returns the *number* of the queries that your application is loading or fetching in the background (useful for app-wide loading indicators).

Usage Examples

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

// How many queries are fetching?
const isFetching = useIsFetching()

// How many queries matching the posts prefix are fetching?
const isFetchingPosts = useIsFetching({ queryKey: ['posts'] })
```

Options

- **filters?: QueryFilters:** Query Filters
- **queryClient?: QueryClient**
  Use this to use a custom QueryClient. Otherwise, the one from the nearest context will be used.

Returns

- **isFetching: number**
  Will be the *number* of the queries that your application is currently loading or fetching in the background.

Additional Information

Edit on GitHub

useIsMutating

useIsMutating is an optional hook that returns the `number` of mutations that your application is fetching (useful for app-wide loading indicators).

Example

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

// How many mutations are fetching?
const isMutating = useIsMutating()

// How many mutations matching the posts prefix are fetching?
const isMutatingPosts = useIsMutating({ mutationKey: ['posts'] })
```

Options

- **filters?:** `MutationFilters`
  Mutation Filters
- **queryClient?:** `QueryClient`
  Use this to use a custom QueryClient. Otherwise, the one from the nearest context will be used.

Returns

- **isMutating:** `number`
  Will be the *number* of the mutations that your application is currently fetching.
- **Note:**
  The value indicates how many mutations are in progress.

Edit on GitHub

useMutationState | TanStack Query React Docs

---

useMutationState is a hook that gives you access to all mutations in the **MutationCache**. You can pass **filters** to narrow down your mutations, and **select** to transform the mutation state.

Example 1: Get all variables of all running mutations

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

const variables = useMutationState({
  filters: { status: 'pending' },
  select: (mutation) => mutation.state.variables,
})
```

Example 2: Get all data for specific mutations via the `mutationKey`

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

const mutationKey = ['posts']

// Some mutation that we want to get the state for
const mutation = useMutation({
  mutationKey,
  mutationFn: (newPost) => {
    return axios.post('/posts', newPost)
  },
})

const data = useMutationState({
  // this mutation key needs to match the mutation key of the given mutation (see abov
  filters: { mutationKey },
  select: (mutation) => mutation.state.data,
})
```

Example 3: Access the latest mutation data via the `mutationKey`

Each invocation of `mutate` adds a new entry to the mutation cache for `gcTime` milliseconds.

To access the latest invocation, check for the last item that `useMutationState` returns.

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

const mutationKey = ['posts']
```

```
// Some mutation that we want to get the state for
const mutation = useMutation({
  mutationKey,
  mutationFn: (newPost) => {
    return axios.post('/posts', newPost)
  },
})

const data = useMutationState({
  // this mutation key needs to match the mutation key of the given mutation (see abov
  filters: { mutationKey },
  select: (mutation) => mutation.state.data,
})

// Latest mutation data
const latest = data[data.length - 1]
```

Options

- **filters**: `MutationFilters`
  Mutation Filters
- **select**: `(mutation: Mutation) => TResult`
  Use this to transform the mutation state.
- **queryClient**: `QueryClient`
  Use a custom QueryClient, otherwise the one from the nearest context is used.

Returns

- An `Array<TResult>` containing the `select` result for each matching mutation.

Edit on GitHub

useSuspenseQuery | TanStack Query React Docs

useSuspenseQuery

```
const result = useSuspenseQuery(options)
```

Options

The same as for useQuery, except for:

- `throwOnError`
- `enabled`
- `placeholderData`

Returns

Same object as useQuery, except that:

- `data` is guaranteed to be defined
- `isPlaceholderData` is missing
- `status` is either **success** or **error**
    - the derived flags are set accordingly.

Caveat

Cancellation does not work.

Edit on GitHub

useSuspenseInfiniteQuery | TanStack Query React Docs

Example

```
const result = useSuspenseInfiniteQuery(options)
```

Options

The same as for useInfiniteQuery, except for:

- *suspense*
- *throwOnError*
- *enabled*
- *placeholderData*

Returns

Same object as useInfiniteQuery, except that:

- `data` is guaranteed to be defined
- `isPlaceholderData` is missing
- `status` is either **success** or **error**
  - the derived flags are set accordingly.

Caveat

Cancellation does not work.

Edit on GitHub

Edit on GitHub

Related Hooks

- useSuspenseQuery
- useSuspenseQueries

Our Partners

— *Tanner Linsley*

Additional Resources

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.
The build and publish utilities used by all of our projects. Use it if you dare!

useSuspenseQueries | TanStack Query React Docs

useSuspenseQueries

```
const result = useSuspenseQueries(options)
```

Options

The same as for useQueries, except that each *query* can't have:

- *suspense*
- *throwOnError*
- *enabled*
- *placeholderData*

Returns

Same structure as useQueries, except that for each *query*:

- *data* is guaranteed to be defined
- *isPlaceholderData* is missing
- *status* is either *success* or *error*
    - The derived flags are set accordingly.

Caveats

Keep in mind that the component will only re-mount after **all queries** have finished loading. Hence, if a query has gone stale in the time it took for all the queries to complete, it will be fetched again at re-mount. To avoid this, make sure to set a high enough *staleTime*.

Cancellation does not work.

Edit on GitHub

Related hooks

- useSuspenseInfiniteQuery
- QueryClientProvider

Our Partners

> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg”
> —Tanner Linsley

Learn More

Additional Resources

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!

MutationCache | TanStack Query Docs

MutationCache

The **MutationCache** is the storage for mutations.

Normally, you will not interact with the MutationCache directly and instead use the QueryClient.

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

const mutationCache = new MutationCache({
  onError: (error) => {
    console.log(error);
  },
  onSuccess: (data) => {
    console.log(data);
  },
});
```

Its available methods are:

- getAll
- subscribe
- clear

Options

- `onError?: (error: unknown, variables: unknown, context: unknown, mutation: Mutation) => Promise<unknown> | unknown`
  - Optional. This function will be called if some mutation encounters an error.
  - If you return a Promise from it, it will be awaited.
- `onSuccess?: (data: unknown, variables: unknown, context: unknown, mutation: Mutation) => Promise<unknown> | unknown`
  - Optional. This function will be called if some mutation is successful.
  - If you return a Promise from it, it will be awaited.
- `onSettled?: (data: unknown | undefined, error: unknown | null, variables: unknown, context: unknown, mutation: Mutation) => Promise<unknown> | unknown`
  - Optional. Called when mutation is settled (either succeeded or errored).
  - Awaited if a Promise is returned.
- `onMutate?: (variables: unknown, mutation: Mutation) => Promise<unknown> | unknown`
  - Optional. Called before mutation executes.
  - Awaited if a Promise is returned.

Global callbacks

The `onError`, `onSuccess`, `onSettled`, and `onMutate` callbacks on the MutationCache can be used to handle these events globally. These are different from `defaultOptions` provided to the QueryClient because:

- `defaultOptions` can be overridden by each Mutation — global callbacks will **always** be called.
- `onMutate` does not allow returning a context value.

---

mutationCache.getAll

Returns all mutations within the cache.

> **Note:** This is not typically needed for most applications, but can be useful in rare scenarios for more information about a mutation.

```
const mutations = mutationCache.getAll();
```

Returns

Mutation[]

Mutation instances from the cache

---

mutationCache.subscribe

Subscribe to the mutation cache and get notified of updates such as mutation states changing or mutations being added, updated, or removed.

```
const callback = (event) => {
  console.log(event.type, event.mutation);
};

const unsubscribe = mutationCache.subscribe(callback);
```

Options

- `callback: (mutation?: MutationCacheNotifyEvent) => void`
  - Called with the mutation cache events when it updates.

Returns

unsubscribe: Function

Unsubscribe callback from the cache

---

mutationCache.clear

Clears the cache entirely and starts fresh.

```
mutationCache.clear();
```

Edit on GitHub

Related References

- QueryCache
- QueryObserver
- OnlineManager
- NotifyManager

Our Partners

Want to Skip the Docs?
Query.gg - The Official React Query Course
"If you're serious about really understanding React Query, there's no better way than with query.gg" — Tanner Linsley

---

More Resources

TanStack Form

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

TanStack Config

The build and publish utilities used by all of our projects. Use it if you dare!

On this page

QueryClientProvider | TanStack Query React Docs

TanStack Query v5

Use the `QueryClientProvider` component to connect and provide a `QueryClient` to your application:

```
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'

const queryClient = new QueryClient()

function App() {
  return <QueryClientProvider client={queryClient}>... </QueryClientProvider>
}
```

Options

- **client:** `QueryClient`
  - **Required**
  - The `QueryClient` instance to provide

Edit on GitHub

Quick Links

- useSuspenseQueries
- useQueryClient

Our Partners

Want to Skip the Docs?

Query.gg - The Official React Query Course

> "If you're serious about *really* understanding React Query, there's no better way than with query.gg"
> — Tanner Linsley

Learn More

Additional Resources

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit,

and Svelte.
Learn More

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!
  Learn More

useQueryClient | TanStack Query React Docs

Overview

The `useQueryClient` hook returns the current `QueryClient` instance.

Usage

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

const queryClient = useQueryClient(queryClient?: QueryClient)
```

Options

- `queryClient?: QueryClient`
  Use this to use a custom `QueryClient`*. Otherwise, the one from the nearest context will be used.*

Edit on GitHub

Edit on GitHub

Related Components

- QueryClientProvider
- queryOptions

Our Partners

Learn more about our partners

Want to Skip the Docs?

Query.gg - The Official React Query Course
"*If you're serious about really understanding React Query, there's no better way than with query.gg*" — Tanner Linsley

---

Additional Resources

Learn More
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

The build and publish utilities used by all of our projects. Use it if you dare!

Query Options - TanStack Query React Docs

queryOptions

```
queryOptions({
  queryKey,
  ...options,
})
```

Options

You can generally pass everything to `queryOptions` that you can also pass to useQuery. Some options will have no effect when forwarded to functions like `queryClient.prefetchQuery`, but TypeScript will still accept those excess properties.

`queryKey: QueryKey`

- **Required**
- The query key to generate options for.

`experimental_prefetchInRender?: boolean`

- Optional
- Defaults to `false`.
- When set to `true`, queries will be prefetched during render, useful for certain optimizations.
- Needed for the experimental `useQuery().promise` functionality.

Edit on GitHub

infiniteQueryOptions

TSX Example

```
infiniteQueryOptions({
  queryKey,
  ...options,
})
```

```
infiniteQueryOptions({
  queryKey,
  ...options,
})
```

Options

You can generally pass everything to **infiniteQueryOptions** that you can also pass to useInfiniteQuery. Some options will have no effect when forwarded to functions like `queryClient.prefetchInfiniteQuery`, but TypeScript will still be fine with those excess properties.

- **queryKey: QueryKey**
  - **Required**
  - The query key to generate options for.

See useInfiniteQuery for more information.

Additional Links

- Edit on GitHub

Related Queries

- queryOptions
- usePrefetchQuery

Our Partners

TanStack Form


Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

Learn More

TanStack Config


The build and publish utilities used by all of our projects. Use it if you dare!

Learn More

usePrefetchQuery | TanStack Query React Docs

usePrefetchQuery

`usePrefetchQuery(options)`

Options

You can pass everything to `usePrefetchQuery` that you can pass to queryClient.prefetchQuery. Remember that some of them are required as below:

- **queryKey: QueryKey**
  - **Required**
  - The query key to prefetch during render
- **queryFn: (context: QueryFunctionContext) ⇒ Promise**
  - **Required, but only if no default query function has been defined**
  - See Default Query Function for more information.

Returns

The `usePrefetchQuery` does not return anything; it is used just to fire a prefetch during render, before a suspense boundary that wraps a component using useSuspenseQuery.

Edit on GitHub:
https://github.com/tanstack/query/edit/main/docs/framework/react/reference/usePrefetchQuery.md

usePrefetchInfiniteQuery | TanStack Query React Docs

---

usePrefetchInfiniteQuery

```
usePrefetchInfiniteQuery(options)
```

Options

You can pass everything to *usePrefetchInfiniteQuery* that you can pass to queryClient.prefetchInfiniteQuery. Remember that some of them are required as below:

- **queryKey: QueryKey**
  Required
  The query key to prefetch during render

- **queryFn: (context: QueryFunctionContext) => Promise<TData>**
  Required, but only if no default query function has been defined
  See Default Query Function for more information.

- **initialPageParam: TPageParam**
  Required
  The default page param to use when fetching the first page.

- **getNextPageParam: (lastPage, allPages, lastPageParam, allPageParams) => TPageParam | undefined | null**
  Required
  When new data is received for this query, this function receives both the last page of the infinite list of data and the full array of all pages, as well as pageParam information.
  It should return a **single variable** that will be passed as the last optional parameter to your query function.
  Return `undefined` or `null` to indicate there is no next page available.

Returns

The `usePrefetchInfiniteQuery` does not return anything; it should be used just to fire a prefetch during render before a Suspense boundary that wraps a component using useSuspenseInfiniteQuery.

Edit on GitHub

https://github.com/tanstack/query/edit/main/docs/framework/react/reference/usePrefetchInfiniteQuery.md

---

Related Hooks

- usePrefetchQuery
- QueryErrorResetBoundary

Our Partners

https://www.speakeasy.com/product/react-query?utm_source=tanstack&utm_campaign=tanstack

Want to Skip the Docs?

> **Query.gg - The Official React Query Course**
> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg”
> —Tanner Linsley

Learn More

Additional Resources

- TanStack Form: Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
- TanStack Config: The build and publish utilities used by all of our projects. Use it if you dare!

QueryErrorResetBoundary | TanStack Query React Docs

Overview

When using **suspense** or **throwOnError** in your queries, you need a way to let queries know that you want to try again when re-rendering after some error occurred. With the **QueryErrorResetBoundary** component you can reset any query errors within the boundaries of the component.

Example (tsx)

```
import { QueryErrorResetBoundary } from '@tanstack/react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App = () => (
  <QueryErrorResetBoundary>
    {({ reset }) => (
      <ErrorBoundary
        onReset={reset}
        fallbackRender={({ resetErrorBoundary }) => (
          <div>
            There was an error!
            <Button onClick={() => resetErrorBoundary()}>Try again</Button>
          </div>
        )}
      >
        <Page />
      </ErrorBoundary>
    )}
  </QueryErrorResetBoundary>
)
```

Edit on GitHub

Edit QueryErrorResetBoundary on GitHub

Navigation

- usePrefetchInfiniteQuery
- useQueryErrorResetBoundary

Our Partners

Learn more about our partners

Additional Resources

Want to Skip the Docs?

Query.gg - The Official React Query Course

"*If you're serious about really understanding React Query, there's no better way than with query.gg*" — Tanner Linsley

---

Learn more about TanStack Form

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

Learn more about TanStack Config

The build and publish utilities used by all of our projects. Use it if you dare!

useQueryErrorResetBoundary | TanStack Query React Docs

Overview

This hook will reset any query errors within the closest `QueryErrorResetBoundary`. If there is no boundary defined, it will reset them globally:

Example Usage (TypeScript)

```
import { useQueryErrorResetBoundary } from '@tanstack/react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App = () => {
  const { reset } = useQueryErrorResetBoundary()
  return (
    <ErrorBoundary
      onReset={reset}
      fallbackRender={({ resetErrorBoundary }) => (
        <div>
          There was an error!
          <Button onClick={() => resetErrorBoundary()}>Try again</Button>
        </div>
      )}
    >
      <Page />
    </ErrorBoundary>
  )
}
```

Example Usage (JavaScript)

```
import { useQueryErrorResetBoundary } from '@tanstack/react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App = () => {
  const { reset } = useQueryErrorResetBoundary()
  return (
    <ErrorBoundary
      onReset={reset}
      fallbackRender={({ resetErrorBoundary }) => (
        <div>
          There was an error!
          <Button onClick={() => resetErrorBoundary()}>Try again</Button>
        </div>
      )}
    >
      <Page />
    </ErrorBoundary>
```

```
  )
}
```

Edit on GitHub

Related Links

- QueryErrorResetBoundary
- hydration

Our Partners

Want to Skip the Docs?

Additional Resources

hydration

dehydrate

dehydrate creates a frozen representation of a *cache* that can later be hydrated with *HydrationBoundary* or *hydrate*. This is useful for passing prefetched queries from server to client or persisting queries to localStorage or other persistent locations. It only includes currently successful queries by default.

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

const dehydratedState = dehydrate(queryClient, {
  shouldDehydrateQuery,
  shouldDehydrateMutation,
})
```

Options

- `client: QueryClient`
  - **Required**
  - The `queryClient` that should be dehydrated
- `options: DehydrateOptions`
  - Optional
  - `shouldDehydrateMutation: (mutation: Mutation) => boolean`
    - Optional
    - Whether to dehydrate mutations.
    - The function is called for each mutation in the cache
    - Return `true` to include this mutation in dehydration, or `false` otherwise
    - Defaults to only including paused mutations
    - To extend the function while retaining default behavior, import and execute `defaultShouldDehydrateMutation`
  - `shouldDehydrateQuery: (query: Query) => boolean`
    - Optional
    - Whether to dehydrate queries.
    - The function is called for each query in the cache
    - Return `true` to include this query in dehydration, or `false` otherwise
    - Defaults to only including successful queries
    - To extend the function, import and execute `defaultShouldDehydrateQuery`
  - `serializeData?: (data: any) => any`
    - A function to transform (serialize) data during dehydration
  - `shouldRedactErrors?: (error: unknown) => boolean`
    - Optional
    - Whether to redact errors from the server during dehydration
    - The function is called for each error in the cache
    - Return `true` to redact this error, or `false` otherwise
    - Defaults to redacting all errors

Returns

- `dehydratedState: DehydratedState`
  - Includes everything needed to hydrate the `queryClient` later
  - Should not rely on the exact format; it's not part of the public API
  - Not serialized; serialization is up to the user

Limitations

Some storage systems (like browser Web Storage API) require values to be JSON serializable. For non-serializable values (like `Error` or `undefined`), you need to serialize them yourself. Since only successful queries are included by default, to include `Errors`, specify `shouldDehydrateQuery: () => true`. Example:

```
// server
const state = dehydrate(client, { shouldDehydrateQuery: () => true }) // include Error
const serializedState = mySerialize(state) // transform Error instances to objects

// client
const state = myDeserialize(serializedState) // transform objects back to Error instan
hydrate(client, state)
```

hydrate

hydrate adds a previously dehydrated state into a *cache*.

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

hydrate(queryClient, dehydratedState, options)
```

Options

- `client: QueryClient`
  - **Required**
  - The `queryClient` to hydrate the state into
- `dehydratedState: DehydratedState`
  - **Required**
  - The state to hydrate into the client
- `options: HydrateOptions`
  - Optional
  - `defaultOptions: DefaultOptions`
    - Optional
    - `mutations: MutationOptions`
      - Default mutation options for hydrated mutations
    - `queries: QueryOptions`
      - Default query options for hydrated queries
    - `deserializeData?: (data: any) => any`
      - Function to transform (deserialize) data before storing in cache
- `queryClient?: QueryClient`
  - Use a custom `QueryClient`; defaults to the nearest context

Limitations

If queries already exist, `hydrate` only overwrites them if the new data is newer. Otherwise, it does not apply.

HydrationBoundary

HydrationBoundary adds a dehydrated state into the *queryClient* that `useQueryClient()` can access. If data exists, the new queries are merged based on timestamp.

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

function App() {
  return <HydrationBoundary state={dehydratedState}>...</HydrationBoundary>
}
```

Note: Only *queries* can be dehydrated with an *HydrationBoundary*.

Options

- `state: DehydratedState`
  - The state to hydrate
- `options: HydrateOptions`
  - Optional
  - `defaultOptions: QueryOptions`
    - Default options for hydrated queries
- `queryClient?: QueryClient`
  - Use a custom `QueryClient`; defaults to nearest context

Edit on GitHub

On this page:

QueryObserver | TanStack Query Docs

QueryObserver

The *QueryObserver* can be used to observe and switch between queries.

Example (TypeScript)

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

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

Options

The options for the *QueryObserver* are exactly the same as those of useQuery.

Edit on GitHub

Edit QueryObserver.md on GitHub

InfiniteQueryObserver | TanStack Query Docs

On this page

InfiniteQueryObserver

The **InfiniteQueryObserver** can be used to observe and switch between infinite queries.

```
const observer = new InfiniteQueryObserver(queryClient, {
  queryKey: ['posts'],
  queryFn: fetchPosts,
  getNextPageParam: (lastPage, allPages) => lastPage.nextCursor,
  getPreviousPageParam: (firstPage, allPages) => firstPage.prevCursor,
})

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

Options

The options for the **InfiniteQueryObserver** are exactly the same as those of useInfiniteQuery.

Edit on GitHub

Related

- QueryObserver
- QueriesObserver

Our Partners

Additional Resources

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

The build and publish utilities used by all of our projects. Use it if you dare!

QueriesObserver | TanStack Query Docs

QueriesObserver

The **QueriesObserver** can be used to observe multiple queries.

Example (TSX)

```
const observer = new QueriesObserver(queryClient, [
  { queryKey: ['post', 1], queryFn: fetchPost },
  { queryKey: ['post', 2], queryFn: fetchPost },
])

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

Options

The options for the **QueriesObserver** are exactly the same as those of useQueries.

Edit on GitHub

https://github.com/tanstack/query/edit/main/docs/reference/QueriesObserver.md

---

Related References

- InfiniteQueryObserver
- streamedQuery

Our Partners

https://www.speakeasy.com/product/react-query?utm_source=tanstack&utm_campaign=tanstack

Want to Skip the Docs?

Query.gg - The Official React Query Course

> "If you're serious about *really* understanding React Query, there's no better way than with query.gg"
> —Tanner Linsley

Learn More

More Tools

TanStack Form

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

Learn More

TanStack Config

The build and publish utilities used by all of our projects. Use it if you dare!

Learn More

streamedQuery

streamedQuery is a helper function to create a query function that streams data from an AsyncIterable. Data will be an Array of all the chunks received. The query will be in a *pending* state until the first chunk of data is received, but will go to *success* after that. The query will stay in fetchStatus *fetching* until the stream ends.

To see *streamedQuery* in action, take a look at our chat example in the examples/react/chat directory on GitHub.

```
import { experimental_streamedQuery as streamedQuery } from '@tanstack/react-query'

const query = queryOptions({
  queryKey: ['data'],
  queryFn: streamedQuery({
    queryFn: fetchDataInChunks,
  }),
})
```

> **Note:** *streamedQuery* is currently marked as *experimental* because we want to gather feedback from the community. If you've tried out the API and have feedback for us, please provide it in this GitHub discussion.

Options

- **queryFn:** *(context: QueryFunctionContext) => Promise<AsyncIterable>*
  Required
  The function that returns a Promise of an AsyncIterable of data to stream in. Receives a QueryFunctionContext.

- **refetchMode?: 'append' | 'reset' | 'replace'**
  Optional. Defines how refetches are handled. Defaults to `'reset'`.
  - When set to `'reset'`, the query will erase all data and go back into *pending* state.
  - When set to `'append'`, data will be appended to existing data.
  - When set to `'replace'`, all data will be written to the cache once the stream ends.

- **maxChunks?: number**
  Optional. The maximum number of chunks to keep in the cache. Defaults to `undefined` (all chunks kept). If `undefined` or `0`, the number of chunks is unlimited. Exceeding this number will remove the oldest chunks.

Edit on GitHub

FocusManager | TanStack Query Docs

On this page

FocusManager

The **FocusManager** manages the focus state within TanStack Query.

It can be used to change the default event listeners or to manually change the focus state.

Its available methods are:

- `setEventListener`
- `subscribe`
- `setFocused`
- `isFocused`

focusManager.setEventListener

setEventListener can be used to set a custom event listener:

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

focusManager.setEventListener((handleFocus) => {
  // Listen to visibilitychange
  if (typeof window !== 'undefined' && window.addEventListener) {
    window.addEventListener('visibilitychange', handleFocus, false)
  }

  return () => {
    // Be sure to unsubscribe if a new handler is set
    window.removeEventListener('visibilitychange', handleFocus)
  }
})
```

focusManager.subscribe

subscribe can be used to subscribe to changes in the visibility state. It returns an unsubscribe function:

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

const unsubscribe = focusManager.subscribe((isVisible) => {
```

```
  console.log('isVisible', isVisible)
})
```

---

focusManager.setFocused

setFocused can be used to manually set the focus state. Set `undefined` to fall back to the default focus check.

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

// Set focused
focusManager.setFocused(true)

// Set unfocused
focusManager.setFocused(false)

// Fallback to the default focus check
focusManager.setFocused(undefined)
```

Options

- `focused: boolean | undefined`

---

focusManager.isFocused

isFocused can be used to get the current focus state:

```
const isFocused = focusManager.isFocused()
```

Edit on GitHub

OnlineManager | TanStack Query Docs

Overview

The **OnlineManager** manages the online state within TanStack Query. It can be used to change the default event listeners or to manually change the online state.

> Per default, the `onlineManager` assumes an active network connection, and listens to the `online` and `offline` events on the `window` object to detect changes.
>
> In previous versions, `navigator.onLine` was used to determine the network status. However, it doesn't work well in Chromium-based browsers. There are a lot of issues around false negatives, which lead to Queries being wrongfully marked as **offline**.
>
> To circumvent this, we now always start with `online: true` and only listen to `online` and `offline` events to update the status.
>
> This should reduce the likelihood of false negatives, however, it might mean false positives for offline apps that load via serviceWorkers, which can work even without an internet connection.

Available methods:

- onlineManager.setEventListener
- onlineManager.subscribe
- onlineManager.setOnline
- onlineManager.isOnline

---

onlineManager.setEventListener

setEventListener can be used to set a custom event listener:

```
import NetInfo from '@react-native-community/netinfo'
import { onlineManager } from '@tanstack/react-query'

onlineManager.setEventListener((setOnline) => {
  return NetInfo.addEventListener((state) => {
    setOnline(!!state.isConnected)
  })
})
```

---

onlineManager.subscribe

subscribe can be used to subscribe to changes in the online state. It returns an unsubscribe function:

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

const unsubscribe = onlineManager.subscribe((isOnline) => {
```

```
  console.log('isOnline', isOnline)
})
```

---

onlineManager.setOnline

setOnline can be used to manually set the online state:

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

// Set to online
onlineManager.setOnline(true)

// Set to offline
onlineManager.setOnline(false)
```

Options:

- `online: boolean`

---

onlineManager.isOnline

isOnline can be used to get the current online state:

```
const isOnline = onlineManager.isOnline()
```

Edit on GitHub

On this page

NotifyManager | TanStack Query Docs

On this page

NotifyManager

The `notifyManager` handles scheduling and batching callbacks in TanStack Query.
It exposes the following methods:

---

notifyManager.batch

`batch` can be used to batch all updates scheduled inside the passed callback.
This is mainly used internally to optimize queryClient updating.

```
function batch<T>(callback: () => T): T
```

---

notifyManager.batchCalls

`batchCalls` is a higher-order function that takes a callback and wraps it.
All calls to the wrapped function schedule the callback to be run on the next batch.

```
type BatchCallsCallback<T extends Array<unknown>> = (...args: T) => void

function batchCalls<T extends Array<unknown>>(
  callback: BatchCallsCallback<T>,
): BatchCallsCallback<T>
```

---

notifyManager.schedule

`schedule` schedules a function to be run on the next batch.
By default, the batch is run with a `setTimeout`, but this can be configured.

```
function schedule(callback: () => void): void
```

notifyManager.setNotifyFunction

`setNotifyFunction` overrides the notify function.
This function is passed the callback when it should be executed.
The default notifyFunction just calls it.

This can be used, for example, to wrap notifications with `React.act` while running tests:

```
import { notifyManager } from '@tanstack/react-query'
import { act } from 'react-dom/test-utils'

notifyManager.setNotifyFunction(act)
```

---

notifyManager.setBatchNotifyFunction

`setBatchNotifyFunction` sets the function to use for batched updates.

If your framework supports a custom batching function, you can set it via:

```
import { notifyManager } from '@tanstack/query-core'
import { batch } from 'solid-js'

notifyManager.setBatchNotifyFunction(batch)
```

---

notifyManager.setScheduler

`setScheduler` configures a custom callback that schedules when the next batch runs.
The default behavior is `setTimeout(callback, 0)`.

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

// Schedule batches in the next microtask
notifyManager.setScheduler(queueMicrotask)

// Schedule batches before the next frame is rendered
notifyManager.setScheduler(requestAnimationFrame)

// Schedule batches some time in the future
notifyManager.setScheduler((cb) => setTimeout(cb, 10))
```

On this page

ESLint Plugin Query

TanStack Query comes with its own ESLint plugin. This plugin is used to enforce best practices and to help you avoid common mistakes.

Installation

The plugin is a separate package that you need to install:

```
npm i -D @tanstack/eslint-plugin-query
```

or

```
pnpm add -D @tanstack/eslint-plugin-query
```

or

```
yarn add -D @tanstack/eslint-plugin-query
```

or

```
bun add -D @tanstack/eslint-plugin-query
```

Flat Config (`eslint.config.js`)

Recommended setup

To enable all of the recommended rules for our plugin, add the following config:

```
import pluginQuery from '@tanstack/eslint-plugin-query'

export default [
  ...pluginQuery.configs['flat/recommended'],
  // Any other config...
]
```

Custom setup

Alternatively, you can load the plugin and configure only the rules you want to use:

```
import pluginQuery from '@tanstack/eslint-plugin-query'

export default [
  {
    plugins: {
      '@tanstack/query': pluginQuery,
    },
    rules: {
      '@tanstack/query/exhaustive-deps': 'error',
    },
  },
  // Any other config...
]
```

Legacy Config (`.eslintrc`)

Recommended setup

To enable all of the recommended rules for our plugin, add `plugin:@tanstack/query/recommended` in extends:

```
{
  "extends": ["plugin:@tanstack/query/recommended"]
}
```

Custom setup

Alternatively, add `@tanstack/query` to the plugins section, and configure the rules you want to use:

```
{
  "plugins": ["@tanstack/query"],
  "rules": {
    "@tanstack/query/exhaustive-deps": "error"
  }
}
```

Rules

- @tanstack/query/exhaustive-deps
- @tanstack/query/no-rest-destructuring
- @tanstack/query/stable-query-client
- @tanstack/query/no-unstable-deps
- @tanstack/query/infinite-query-property-order
- @tanstack/query/no-void-query-fn

On this page

Exhaustive dependencies for query keys | TanStack Query Docs

On this page

Rule Details

Query keys should be seen like a dependency array to your query function: Every variable that is used inside the `queryFn` should be added to the query key.
This makes sure that queries are cached independently and that queries are refetched automatically when the variables change.

Examples of *incorrect* code for this rule:

```
/* eslint "@tanstack/query/exhaustive-deps": "error" */

useQuery({
  queryKey: ['todo'],
  queryFn: () => api.getTodo(todoId),
})

const todoQueries = {
  detail: (id) => ({ queryKey: ['todo'], queryFn: () => api.getTodo(id) }),
}
```

```
/* eslint "@tanstack/query/exhaustive-deps": "error" */

useQuery({
  queryKey: ['todo'],
  queryFn: () => api.getTodo(todoId),
})

const todoQueries = {
  detail: (id) => ({ queryKey: ['todo'], queryFn: () => api.getTodo(id) }),
}
```

Examples of *correct* code for this rule:

```
useQuery({
  queryKey: ['todo', todoId],
  queryFn: () => api.getTodo(todoId),
})

const todoQueries = {
  detail: (id) => ({ queryKey: ['todo', id], queryFn: () => api.getTodo(id) }),
}
```

```
useQuery({
  queryKey: ['todo', todoId],
  queryFn: () => api.getTodo(todoId),
})

const todoQueries = {
  detail: (id) => ({ queryKey: ['todo', id], queryFn: () => api.getTodo(id) }),
}
```

When Not To Use It

If you don't care about the rules of the query keys, then you will not need this rule.

Attributes

- ✅ Recommended
- 🔧 Fixable

Edit on GitHub

On this page

Stable Query Client

The QueryClient contains the QueryCache, so you'd only want to create one instance of the QueryClient for the lifecycle of your application - *not* a new instance on every render.

> Exception: It's allowed to create a new QueryClient inside an async Server Component, because the async function is only called once on the server.

Rule Details

Examples of **incorrect** code for this rule:

```
/* eslint "@tanstack/query/stable-query-client": "error" */

function App() {
  const queryClient = new QueryClient()
  return (
    <QueryClientProvider client={queryClient}>
      <Home />
    </QueryClientProvider>
  )
}
```

```
/* eslint "@tanstack/query/stable-query-client": "error" */

function App() {
  const queryClient = new QueryClient()
  return (
    <QueryClientProvider client={queryClient}>
      <Home />
    </QueryClientProvider>
  )
}
```

Examples of **correct** code for this rule:

```
function App() {
  const [queryClient] = useState(() => new QueryClient())
  return (
    <QueryClientProvider client={queryClient}>
      <Home />
    </QueryClientProvider>
  )
}
```

```
function App() {
  const [queryClient] = useState(() => new QueryClient())
  return (
    <QueryClientProvider client={queryClient}>
      <Home />
    </QueryClientProvider>
  )
}
```

```
const queryClient = new QueryClient()
function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Home />
    </QueryClientProvider>
  )
}
```

```
const queryClient = new QueryClient()
async function App() {
  const queryClient = new QueryClient()
  await queryClient.prefetchQuery(options)
}
```

Attributes

- ✅ Recommended
- 🔧 Fixable

Edit on GitHub

On this page

Additional Resources

- [Exhaustive Deps](https://query.lol/>, /query/latest/docs/eslint/exhaustive-deps)
- [No Rest Destructuring](https://query.lol/>, /query/latest/docs/eslint/no-rest-destructuring)

Our Partners

Disallow object rest destructuring on query results

Use object rest destructuring on query results automatically subscribes to every field of the query result, which may cause unnecessary re-renders.
This makes sure that you only subscribe to the fields that you actually need.

Rule Details

Examples of *incorrect* code for this rule:

```
/* eslint "@tanstack/query/no-rest-destructuring": "warn" */

const useTodos = () => {
  const { data: todos, ...rest } = useQuery({
    queryKey: ['todos'],
    queryFn: () => api.getTodos(),
  })
  return { todos, ...rest }
}
```

```
/* eslint "@tanstack/query/no-rest-destructuring": "warn" */

const useTodos = () => {
  const { data: todos, ...rest } = useQuery({
    queryKey: ['todos'],
    queryFn: () => api.getTodos(),
  })
  return { todos, ...rest }
}
```

Examples of **correct** code for this rule:

```
const todosQuery = useQuery({
  queryKey: ['todos'],
  queryFn: () => api.getTodos(),
})

// normal object destructuring is fine
const { data: todos } = todosQuery
```

```
const todosQuery = useQuery({
  queryKey: ['todos'],
  queryFn: () => api.getTodos(),
})

// normal object destructuring is fine
const { data: todos } = todosQuery
```

When Not To Use It

If you set the `notifyOnChangeProps` options manually, you can disable this rule.
Since you are not using tracked queries, you are responsible for specifying which props should trigger a re-render.

Attributes

- ✅ Recommended
- 🔧 Fixable

Edit on GitHub

On this page

Disallow putting the result of query hooks directly in a React hook dependency array

Rule Details

Explanation

The object returned from the following query hooks is **not** referentially stable:

- `useQuery`
- `useSuspenseQuery`
- `useQueries`
- `useSuspenseQueries`
- `useInfiniteQuery`
- `useSuspenseInfiniteQuery`
- `useMutation`

The object returned from those hooks should **not** be put directly into the dependency array of a React hook (e.g., `useEffect`, `useMemo`, `useCallback`). Instead, destructure the return value of the query hook and pass the destructured values into the dependency array of the React hook.

Examples of *incorrect* code for this rule:

```
/* eslint "@tanstack/query/no-unstable-deps": "warn" */
import { useCallback } from 'React'
import { useMutation } from '@tanstack/react-query'

function Component() {
  const mutation = useMutation({ mutationFn: (value: string) => value })
  const callback = useCallback(() => {
    mutation.mutate('hello')
  }, [mutation])
  return null
}
```

```
/* eslint "@tanstack/query/no-unstable-deps": "warn" */
import { useCallback } from 'React'
import { useMutation } from '@tanstack/react-query'

function Component() {
  const mutation = useMutation({ mutationFn: (value: string) => value })
  const callback = useCallback(() => {
    mutation.mutate('hello')
  }, [mutation])
  return null
}
```

Examples of *correct* code for this rule:

```
/* eslint "@tanstack/query/no-unstable-deps": "warn" */
import { useCallback } from 'React'
import { useMutation } from '@tanstack/react-query'

function Component() {
  const { mutate } = useMutation({ mutationFn: (value: string) => value })
  const callback = useCallback(() => {
    mutate('hello')
  }, [mutate])
  return null
}
```

```
/* eslint "@tanstack/query/no-unstable-deps": "warn" */
import { useCallback } from 'React'
import { useMutation } from '@tanstack/react-query'

function Component() {
  const { mutate } = useMutation({ mutationFn: (value: string) => value })
  const callback = useCallback(() => {
    mutate('hello')
  }, [mutate])
  return null
}
```

Attributes

- ✅ Recommended
- 🔧 Fixable

Edit on GitHub

On this page

Ensure correct order of inference sensitive properties for infinite queries

On this page

Explanation

For the following functions, the property order of the passed-in object matters due to type inference:

- `useInfiniteQuery`
- `useSuspenseInfiniteQuery`
- `infiniteQueryOptions`

The **correct property order** is:

1. `queryFn`
2. `getPreviousPageParam`
3. `getNextPageParam`

All other properties are insensitive to the order as they do not depend on type inference.

Rule Details

Examples of **incorrect** code:

```
/* eslint "@tanstack/query/infinite-query-property-order": "warn" */
import { useInfiniteQuery } from '@tanstack/react-query'

const query = useInfiniteQuery({
  queryKey: ['projects'],
  getNextPageParam: (lastPage) => lastPage.nextId ?? undefined,
  queryFn: async ({ pageParam }) => {
    const response = await fetch(`/api/projects?cursor=${pageParam}`)
    return await response.json()
  },
  initialPageParam: 0,
  getPreviousPageParam: (firstPage) => firstPage.previousId ?? undefined,
  maxPages: 3,
})
```

```
/* eslint "@tanstack/query/infinite-query-property-order": "warn" */
import { useInfiniteQuery } from '@tanstack/react-query'

const query = useInfiniteQuery({
  queryKey: ['projects'],
  getNextPageParam: (lastPage) => lastPage.nextId ?? undefined,
```

```
  queryFn: async ({ pageParam }) => {
    const response = await fetch(`/api/projects?cursor=${pageParam}`)
    return await response.json()
  },
  initialPageParam: 0,
  getPreviousPageParam: (firstPage) => firstPage.previousId ?? undefined,
  maxPages: 3,
})
```

Examples of **correct** code:

```
/* eslint "@tanstack/query/infinite-query-property-order": "warn" */
import { useInfiniteQuery } from '@tanstack/react-query'

const query = useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: async ({ pageParam }) => {
    const response = await fetch(`/api/projects?cursor=${pageParam}`)
    return await response.json()
  },
  initialPageParam: 0,
  getPreviousPageParam: (firstPage) => firstPage.previousId ?? undefined,
  getNextPageParam: (lastPage) => lastPage.nextId ?? undefined,
  maxPages: 3,
})
```

```
/* eslint "@tanstack/query/infinite-query-property-order": "warn" */
import { useInfiniteQuery } from '@tanstack/react-query'

const query = useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: async ({ pageParam }) => {
    const response = await fetch(`/api/projects?cursor=${pageParam}`)
    return await response.json()
  },
  initialPageParam: 0,
  getPreviousPageParam: (firstPage) => firstPage.previousId ?? undefined,
  getNextPageParam: (lastPage) => lastPage.nextId ?? undefined,
  maxPages: 3,
})
```

Attributes

- ✅ Recommended
- 🔧 Fixable

Edit on GitHub

Edit this page on GitHub

Additional Resources

- Rule Details
- Attributes

TkDodo's Blog | TanStack Query React Docs

On this page

Blog Overview

TanStack Query maintainer TkDodo has a series of blog posts about using and working with the library. Some articles show general best practices, but most have an *opinionated* point of view.

Featured Articles

#1: Practical React Query

> An advanced introduction to React Query, showing practical tips that go beyond the docs. It covers explaining the defaults (*staleTime* vs. *gcTime*), concepts like keeping server and client state separate, handling dependencies, creating custom hooks, and outlining why the *enabled* option is very powerful. Read more...

#2: React Query Data Transformations

> Learn the possibilities to perform the quite common and important task of transforming your data with React Query. From transforming in the *queryFn* to using the *select* option, this article outlines the pros and cons of all the different approaches. Read more...

#3: React Query Render Optimizations

> Let's take a look at what you can do when your component re-renders too often when using React Query. The library is already pretty optimized, but there are still some opt-in features (like *tracked*

queries) that you can use to avoid the *isFetching* transition. We're also looking into what *structural sharing* refers to. Read more...

#4: Status Checks in React Query

We usually check for *isPending* first before checking for *isError*, but sometimes, checking if *data* is available should be the first thing to do. This article shows how the wrong status check order can negatively impact user experience. Read more...

#5: Testing React Query

The docs already cover pretty well what you need to do to get started when testing React Query. This article shows some additional tips (like turning off *retries* or silencing the *console*) you might want to follow when testing custom hooks or components using them. It also links to an example repository with tests for success and error states, powered by *mock-service-worker*. Read more...

#6: React Query and TypeScript

Since React Query is written in TypeScript, it has great support for it. This blog post explains the various Generics, how you can leverage type inference to avoid having to explicitly type *useQuery* and friends, what to do with *unknown* errors, how type narrowing works, and more! Read more...

#7: Using WebSockets with React Query

A step-by-step guide on how to make real-time notifications work with React Query, with either event-based subscriptions or pushing full data directly to the client. Applicable to anything from the browser native WebSocket API over Firebase and even GraphQL subscriptions. Read more...

And many more articles covering advanced topics, community resources, examples, plugins, and internal architecture details.

Additional Resources

- Infinite Query Property Order
- Community Projects
- Want to Skip the Docs?

> Query.gg - The Official React Query Course “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg” —Tanner Linsley

Partners & Tools

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!

Creative Commons Usage Notes

- Some code snippets and images are presented inline or as part of the content.
- The article emphasizes best practices, community workflows, and architectural insights into React Query.

Note: This is a simplified and formatted excerpt from a comprehensive documentation and blog page.

Community Projects

There are lots of community projects that build on top of React Query and use it to provide additional functionality or enhanced developer experience. Projects are listed in alphabetical order. If you have a project that you would like to add to this list, please open a PR!

Please note that these projects are entirely community maintained. If you have questions about these projects, please reach out to the project maintainers.

Atomic CRM

A full-featured CRM built with React, react-admin, and Supabase.
Link: https://marmelab.com/atomic-crm/

batshit

A batch manager that will deduplicate and batch requests for a certain data type made within a window.
Link: https://github.com/yornaath/batshit

Blitz

The Missing Fullstack Toolkit for Next.js.
Link: https://blitzjs.com/

Connect

A family of libraries for building browser and gRPC-compatible HTTP APIs.
Link: https://connectrpc.com/docs

DevTools Chrome Extension

A Chrome browser extension that provides devtools for TanStack Query, allowing you to inspect and debug queries, mutations, and cache state directly in Chrome DevTools.
Link: https://chromewebstore.google.com/detail/tanstack-query-devtools/annajfchloimdhceglpgglpeepfghfai

GraphQL Code Generator

Generate React Query hooks from your GraphQL schema.
Link: https://the-guild.dev/graphql/codegen

Http-wizard

End-to-end type-safe Fastify API with TypeScript magic ✨.
Link: https://http-wizard.com

Kubb

Generate SDKs for all your APIs.
Link: https://www.kubb.dev/

NgQuery

Query adapter for Angular.
Link: https://ngneat.github.io/query/

Normy

Automatic normalization and data updates for data fetching libraries.
Link: https://github.com/klis87/normy

OpenAPI codegen

A tool for generating code based on an OpenAPI schema.
Link: https://github.com/fabien0102/openapi-codegen

OpenAPI Qraft React

Generate type-safe API clients and Hooks for TanStack Query directly from OpenAPI Documents. Zero-runtime overhead, Proxy-based design, seamless SSR support, and full TanStack Query functionality.
Link: https://github.com/OpenAPI-Qraft/openapi-qraft

OpenAPI React Query codegen

Generate TanStack Query hooks based on an OpenAPI specification file.
Link: https://github.com/7nohe/openapi-react-query-codegen

OpenAPI zod client

Generate a zodios client from an OpenAPI specification.
Link: https://github.com/astahmer/openapi-zod-client

openapi-fetch

A 2KB min, typesafe fetch wrapper that uses static TypeScript type inference and no runtime checks.
Link: https://openapi-ts.dev/openapi-react-query/

Orval

Generate TypeScript client from OpenAPI specifications.
Link: https://orval.dev/

Query Key factory

A library for creating typesafe standardized query keys, useful for cache management in `@tanstack/query`.
Link: https://github.com/lukemorales/query-key-factory

Rapini

🥬 OpenAPI to React Query (or SWR) & Axios.
Link: https://github.com/rametta/rapini

React Query Kit

🕊️ A toolkit for ReactQuery that makes ReactQuery hooks reusable and typesafe.
Link: https://github.com/liaoliao666/react-query-kit

React Query Rewind

Time travel and visualize state during development.
Link: https://reactqueryrewind.com/

React Query Swagger

Generate React Query hooks based on Swagger API definitions.
Link: https://github.com/Shaddix/react-query-swagger

Suspensive React Query

Enhances React Query with Suspense support, allowing for simpler and more declarative data fetching.
Link: https://suspensive.org/docs/react-query/motivation

tRPC

End-to-end typesafe APIs made easy.
Link: https://trpc.io/

ts-rest

Incrementally adoptable type-safety for your new and existing APIs.
Link: https://ts-rest.com/

wagmi

React Hooks for Ethereum based on `@tanstack/react-query`.
Link: https://wagmi.sh/

zodios

End-to-end typesafe REST API toolbox.
Link: https://www.zodios.org/

Edit on GitHub

On this page

- Atomic CRM
- batshit
- Blitz
- Connect
- DevTools Chrome Extension
- GraphQL Code Generator
- Http-wizard
- Kubb
- NgQuery
- Normy
- OpenAPI codegen
- OpenAPI Qraft React
- OpenAPI React Query codegen
- OpenAPI zod client
- openapi-fetch
- Orval
- Query Key factory
- Rapini
- React Query Kit
- React Query Rewind
- React Query Swagger
- Suspensive React Query
- tRPC
- ts-rest
- wagmi
- zodios

React Example: Simple

Github
StackBlitz
CodeSandbox

Code Explorer / Interactive Sandbox

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  useQuery,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <ReactQueryDevtools />
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { isPending, error, data, isFetching } = useQuery({
    queryKey: ['repoData'],
    queryFn: async () => {
      const response = await fetch(
        'https://api.github.com/repos/TanStack/query'
      )
      return await response.json()
    },
  })

  if (isPending) return 'Loading...'

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <h1>{data.full_name}</h1>
      <p>{data.description}</p>
      <strong>👀 {data.subscribers_count}</strong>{' '}
      <strong>✨ {data.stargazers_count}</strong>{' '}
      <strong>🍴 {data.forks_count}</strong>
      <div>{isFetching ? 'Updating...' : ''}</div>
    </div>
```

```
  )
}

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

Note: The content includes repeated code blocks with identical content.

React TanStack Query Basic Example | TanStack Query Docs

TanStack Query v5

Framework

React

Version

Latest

Search...

- K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview react
- Installation react
- Quick Start react
- Devtools react
- Videos & Talks react
- Comparison react
- TypeScript react
- GraphQL react
- React Native react

Guides & Concepts

- Important Defaults react
- Queries react
- Query Keys react
- Query Functions react
- Query Options react
- Network Mode react
- Parallel Queries react
- Dependent Queries react

- Background Fetching Indicators react
- Window Focus Refetching react
- Disabling/Pausing Queries react
- Query Retries react
- Paginated Queries react
- Infinite Queries react
- Initial Query Data react
- Placeholder Query Data react
- Mutations react
- Query Invalidation react
- Invalidation from Mutations react
- Updates from Mutation Responses react
- Optimistic Updates react
- Query Cancellation react
- Scroll Restoration react
- Filters react
- Performance & Request Waterfalls react
- Prefetching & Router Integration react
- Server Rendering & Hydration react
- Advanced Server Rendering react
- Caching react
- Render Optimizations react
- Default Query Fn react
- Suspense react
- Testing react
- Does this replace [Redux, MobX, etc]? react
- Migrating to v3 react
- Migrating to v4 react
- Migrating to v5 react

API Reference

- QueryClient core
- QueryCache core
- MutationCache core
- QueryObserver core
- InfiniteQueryObserver core
- QueriesObserver core
- streamedQuery core
- focusManager core
- onlineManager core
- notifyManager core
- useQuery react
- useQueries react
- useInfiniteQuery react
- useMutation react
- useIsFetching react
- useIsMutating react
- useMutationState react
- useSuspenseQuery react
- useSuspenseInfiniteQuery react
- useSuspenseQueries react
- QueryClientProvider react
- useQueryClient react
- queryOptions react
- infiniteQueryOptions react
- usePreFetchQuery react

- usePreFetchInfiniteQuery react
- QueryErrorResetBoundary react
- useQueryErrorResetBoundary react
- hydration react

ESLint

- eslint-plugin-query core
- Exhaustive Deps core
- Stable Query Client core
- No Rest Destructuring core
- No Unstable Deps core
- Infinite Query Property Order core

Community Resources

- TkDodo's Blog react
- Community Projects react

Examples

- Simple react
- Basic react
- Basic w/ GraphQL-Request react
- Auto Refetching / Polling / Realtime react
- Optimistic Updates (UI) react
- Optimistic Updates (Cache) react
- Pagination react
- Load-More & Infinite Scroll react
- Infinite query with Max pages react
- Suspense react
- Default Query Function react
- Playground react
- Prefetching react
- Star Wars react
- Rick And Morty react
- Next.js Pages react
- Next.js app with prefetching react
- Next.js app with streaming react
- React Native react
- React Router react
- Offline Queries and Mutations react
- Algolia react
- Shadow DOM react
- Devtools Embedded Panel react
- Chat example (streaming) react

Plugins

- persistQueryClient react
- createSyncStoragePersister react
- createAsyncStoragePersister react
- broadcastQueryClient (Experimental) react
- createPersister (Experimental) react

React Example: Basic

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox

- **public**
  - **src**
    - index.tsx
  - .gitignore
  - README.md
  - eslint.config.js
  - index.html
  - package.json
  - tsconfig.json
  - vite.config.ts

Source Code (TypeScript)

```
import * as React from 'react'
import ReactDOM from 'react-dom/client'
import { QueryClient, useQuery, useQueryClient } from '@tanstack/react-query'
import { PersistQueryClientProvider } from '@tanstack/react-query-persist-client'
import { createAsyncStoragePersister } from '@tanstack/query-async-storage-persister'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      gcTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})

const persister = createAsyncStoragePersister({
  storage: window.localStorage,
})

type Post = {
  id: number
  title: string
  body: string
}

function usePosts() {
  return useQuery({
    queryKey: ['posts'],
    queryFn: async (): Promise<Array<Post>> => {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts')
      return await response.json()
    },
  })
```

```
}

function Posts({
  setPostId,
}: {
  setPostId: React.Dispatch<React.SetStateAction<number>>
}) {
  const queryClient = useQueryClient()
  const { status, data, error, isFetching } = usePosts()

  return (
    <div>
      <h1>Posts</h1>
      <div>
        {status === 'pending' ? (
          'Loading...'
        ) : status === 'error' ? (
          <span>Error: {error.message}</span>
        ) : (
          <>
            <div>
              {data.map((post) => (
                <p key={post.id}>
                  <a
                    onClick={() => setPostId(post.id)}
                    href="#"
                    style={
                      // We can access the query data here to show bold links for
                      // ones that are cached
                      queryClient.getQueryData(['post', post.id])
                        ? {
                            fontWeight: 'bold',
                            color: 'green',
                          }
                        : {}
                    }
                  >
                    {post.title}
                  </a>
                </p>
              ))}
            </div>
            <div>{isFetching ? 'Background Updating...' : ' '}</div>
          </>
        )}
      </div>
    </div>
  )
}

const getPostById = async (id: number): Promise<Post> => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/posts/${id}`,
  )
  return await response.json()
```

```
}

function usePost(postId: number) {
  return useQuery({
    queryKey: ['post', postId],
    queryFn: () => getPostById(postId),
    enabled: !!postId,
  })
}

function Post({
  postId,
  setPostId,
}: {
  postId: number
  setPostId: React.Dispatch<React.SetStateAction<number>>
}) {
  const { status, data, error, isFetching } = usePost(postId)

  return (
    <div>
      <div>
        <a onClick={() => setPostId(-1)} href="#">
          Back
        </a>
      </div>
      {!postId || status === 'pending' ? (
        'Loading...'
      ) : status === 'error' ? (
        <span>Error: {error.message}</span>
      ) : (
        <>
          <h1>{data.title}</h1>
          <div>
            <p>{data.body}</p>
          </div>
          <div>{isFetching ? 'Background Updating...' : ' '}</div>
        </>
      )}
    </div>
  )
}

function App() {
  const [postId, setPostId] = React.useState(-1)

  return (
    <PersistQueryClientProvider
      client={queryClient}
      persistOptions={{ persister }}
    >
      <p>
        As you visit the posts below, you will notice them in a loading state
        the first time you load them. However, after you return to this list and
        click on any posts you have already visited again, you will see them
```

```
        load instantly and background refresh right before your eyes!{' '}
        <strong>
          (You may need to throttle your network speed to simulate longer
          loading sequences)
        </strong>
      </p>
      {postId > -1 ? (
        <Post postId={postId} setPostId={setPostId} />
      ) : (
        <Posts setPostId={setPostId} />
      )}
      <ReactQueryDevtools initialIsOpen />
    </PersistQueryClientProvider>
  )
}

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

Note: The same code block is duplicated, ensure to include it only once in actual use.

React Example: Default Query Function

Github
StackBlitz
CodeSandbox

Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import type { QueryKey } from '@tanstack/react-query'

type Post = {
  id: number
  title: string
  body: string
}

// Define a default query function that will receive the query key
const defaultQueryFn = async ({ queryKey }: { queryKey: QueryKey }) => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com${queryKey[0]}`
  )
  return await response.json()
}

// provide the default query function to your app via the query client
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      queryFn: defaultQueryFn,
    },
  },
})

function App() {
  const [postId, setPostId] = React.useState(-1)

  return (
    <QueryClientProvider client={queryClient}>
      <p>
        As you visit the posts below, you will notice them in a loading state
```

```
        the first time you load them. However, after you return to this list and
        click on any posts you have already visited again, you will see them
        load instantly and background refresh right before your eyes!{' '}
        <strong>
          (You may need to throttle your network speed to simulate longer
          loading sequences)
        </strong>
      </p>
      {postId > -1 ? (
        <Post postId={postId} setPostId={setPostId} />
      ) : (
        <Posts setPostId={setPostId} />
      )}
      <ReactQueryDevtools initialIsOpen />
    </QueryClientProvider>
  )
}

function Posts({
  setPostId,
}: {
  setPostId: React.Dispatch<React.SetStateAction<number>>
}) {
  const queryClient = useQueryClient()

  // All you have to do now is pass a key!
  const { status, data, error, isFetching } = useQuery<Array<Post>>({
    queryKey: ['/posts'],
  })

  return (
    <div>
      <h1>Posts</h1>
      <div>
        {status === 'pending' ? (
          'Loading...'
        ) : status === 'error' ? (
          <span>Error: {error.message}</span>
        ) : (
          <>
            <div>
              {data.map((post) => (
                <p key={post.id}>
                  <a
                    onClick={() => setPostId(post.id)}
                    href="#"
                    style={
                      // We can use the queryCache here to show bold links for
                      // ones that are cached
                      queryClient.getQueryData([`/posts/${post.id}`])
                        ? {
                            fontWeight: 'bold',
                            color: 'green',
                          }
                        : {}
```

```
                  }
                >
                  {post.title}
                </a>
              </p>
            ))}
          </div>
          <div>{isFetching ? 'Background Updating...' : ' '}</div>
        </>
      )}
    </div>
  </div>
  )
}

function Post({
  postId,
  setPostId,
}: {
  postId: number
  setPostId: React.Dispatch<React.SetStateAction<number>>
}) {
  // You can even leave out the queryFn and just go straight into options
  const { status, data, error, isFetching } = useQuery<Post>({
    queryKey: [`/posts/${postId}`],
    enabled: !!postId,
  })

  return (
    <div>
      <div>
        <a onClick={() => setPostId(-1)} href="#">
          Back
        </a>
      </div>
      {!postId || status === 'pending' ? (
        'Loading...'
      ) : status === 'error' ? (
        <span>Error: {error.message}</span>
      ) : (
        <>
          <h1>{data.title}</h1>
          <div>
            <p>{data.body}</p>
          </div>
          <div>{isFetching ? 'Background Updating...' : ' '}</div>
        </>
      )}
    </div>
  )
}

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

React Example: Playground

Github
StackBlitz
CodeSandbox

Code Explorer and Sandbox

File Structure

- `public`
- `src`
  - `index.tsx`
  - `styles.css`
- `.eslintrc`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.ts`

React Example Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  useMutation,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import './styles.css'

let id = 0
let list = [
  'apple',
  'banana',
  'pineapple',
  'grapefruit',
  'dragonfruit',
  'grapes',
].map((d) => ({ id: id++, name: d, notes: 'These are some notes' }))

type Todos = typeof list
type Todo = Todos[0]

let errorRate = 0.05
```

```
let queryTimeMin = 1000
let queryTimeMax = 2000

const queryClient = new QueryClient()

function Root() {
  const [staleTime, setStaleTime] = React.useState(1000)
  const [gcTime, setGcTime] = React.useState(3000)
  const [localErrorRate, setErrorRate] = React.useState(errorRate)
  const [localFetchTimeMin, setLocalFetchTimeMin] = React.useState(queryTimeMin)
  const [localFetchTimeMax, setLocalFetchTimeMax] = React.useState(queryTimeMax)

  React.useEffect(() => {
    errorRate = localErrorRate
    queryTimeMin = localFetchTimeMin
    queryTimeMax = localFetchTimeMax
  }, [localErrorRate, localFetchTimeMax, localFetchTimeMin])

  React.useEffect(() => {
    queryClient.setDefaultOptions({
      queries: {
        staleTime,
        gcTime,
      },
    })
  }, [gcTime, staleTime])

  return (
    <QueryClientProvider client={queryClient}>
      <p>
        The "staleTime" and "gcTime" durations have been altered in this example
        to show how query stale-ness and query caching work on a granular level
      </p>
      <div>
        Stale Time: <input
          type="number"
          min="0"
          step="1000"
          value={staleTime}
          onChange={(e) => setStaleTime(parseFloat(e.target.value))}
          style={{ width: '100px' }}
        />
      </div>
      <div>
        Garbage collection Time: <input
          type="number"
          min="0"
          step="1000"
          value={gcTime}
          onChange={(e) => setGcTime(parseFloat(e.target.value))}
          style={{ width: '100px' }}
        />
      </div>
      <br />
      <div>
```

```
        Error Rate: <input
          type="number"
          min="0"
          max="1"
          step=".05"
          value={localErrorRate}
          onChange={(e) => setErrorRate(parseFloat(e.target.value))}
          style={{ width: '100px' }}
        />
      </div>
      <div>
        Fetch Time Min: <input
          type="number"
          min="1"
          step="500"
          value={localFetchTimeMin}
          onChange={(e) => setLocalFetchTimeMin(parseFloat(e.target.value))}
          style={{ width: '60px' }}
        />{' '}
      </div>
      <div>
        Fetch Time Max: <input
          type="number"
          min="1"
          step="500"
          value={localFetchTimeMax}
          onChange={(e) => setLocalFetchTimeMax(parseFloat(e.target.value))}
          style={{ width: '60px' }}
        />
      </div>
      <br />
      <App />
      <br />
      <ReactQueryDevtools initialIsOpen />
    </QueryClientProvider>
  )
}

function App() {
  const queryClient = useQueryClient()
  const [editingIndex, setEditingIndex] = React.useState<number | null>(null)
  const [views, setViews] = React.useState(['', 'fruit', 'grape'])

  return (
    <div className="App">
      <div>
        <button onClick={() => queryClient.invalidateQueries()}>
          Force Refetch All
        </button>
      </div>
      <br />
      <hr />
      {views.map((view, index) => (
        <div key={index}>
          <Todos
```

```
            initialFilter={view}
            setEditingIndex={setEditingIndex}
            onRemove={() => {
              setViews((old) => [...old, ''])
            }}
          />
          <br />
        </div>
      ))}
      <button
        onClick={() => {
          setViews((old) => [...old, ''])
        }}
      >
        Add Filter List
      </button>
      <hr />
      {editingIndex !== null ? (
        <>
          <EditTodo
            editingIndex={editingIndex}
            setEditingIndex={setEditingIndex}
          />
          <hr />
        </>
      ) : null}
      <AddTodo />
    </div>
  )
}

function Todos({ initialFilter = '', setEditingIndex }) {
  const [filter, setFilter] = React.useState(initialFilter)

  const { status, data, isFetching, error, failureCount, refetch } = useQuery({
    queryKey: ['todos', { filter }],
    queryFn: fetchTodos,
  })

  return (
    <div>
      <div>
        <label>
          Filter:{' '}
          <input value={filter} onChange={(e) => setFilter(e.target.value)} />
        </label>
      </div>
      {status === 'pending' ? (
        <span>Loading... (Attempt: {failureCount + 1})</span>
      ) : status === 'error' ? (
        <span>
          Error: {error.message}
          <br />
          <button onClick={() => refetch()}>Retry</button>
        </span>
```

```
      ) : (
        <>
          <ul>
            {data
              ? data.map((todo) => (
                  <li key={todo.id}>
                    {todo.name}{' '}
                    <button onClick={() => setEditingIndex(todo.id)}>Edit</button>
                  </li>
                ))
              : null}
          </ul>
          <div>
            {isFetching ? (
              <span>
                Background Refreshing... (Attempt: {failureCount + 1})
              </span>
            ) : (
              <span> </span>
            )}
          </div>
        </>
      )}
    </div>
  )
}

function EditTodo({ editingIndex, setEditingIndex }) {
  const queryClient = useQueryClient()

  // Don't attempt to query until editingIndex is truthy
  const { status, data, isFetching, error, failureCount, refetch } = useQuery({
    queryKey: ['todo', { id: editingIndex }],
    queryFn: () => fetchTodoById({ id: editingIndex }),
  })

  const [todo, setTodo] = React.useState(data || {})

  React.useEffect(() => {
    if (editingIndex !== null && data) {
      setTodo(data)
    } else {
      setTodo({})
    }
  }, [data, editingIndex])

  const saveMutation = useMutation({
    mutationFn: patchTodo,
    onSuccess: (data) => {
      // Update `todos` and the individual todo queries when this mutation succeeds
      queryClient.invalidateQueries({ queryKey: ['todos'] })
      queryClient.setQueryData(['todo', { id: editingIndex }], data)
    },
  })
```

```
const onSave = () => {
  saveMutation.mutate(todo)
}

const disableEditSave =
  status === 'pending' || saveMutation.status === 'pending'

return (
  <div>
    <div>
      {data ? (
        <>
          <button onClick={() => setEditingIndex(null)}>Back</button> Editing
          Todo "{data.name}" (#{editingIndex})
        </>
      ) : null}
    </div>
    {status === 'pending' ? (
      <span>Loading... (Attempt: {failureCount + 1})</span>
    ) : error ? (
      <span>
        Error! <button onClick={() => refetch()}>Retry</button>
      </span>
    ) : (
      <>
        <label>
          Name:{' '}
          <input
            value={todo.name}
            onChange={(e) =>
              e.persist() ||
              setTodo((old) => ({ ...old, name: e.target.value }))
            }
            disabled={disableEditSave}
          />
        </label>
        <label>
          Notes:{' '}
          <input
            value={todo.notes}
            onChange={(e) =>
              e.persist() ||
              setTodo((old) => ({ ...old, notes: e.target.value }))
            }
            disabled={disableEditSave}
          />
        </label>
        <div>
          <button onClick={onSave} disabled={disableEditSave}>
            Save
          </button>
        </div>
        <div>
          {saveMutation.status === 'pending'
            ? 'Saving...'
```

```
              : saveMutation.status === 'error'
              ? saveMutation.error.message
              : 'Saved!'}
          </div>
          <div>
            {isFetching ? (
              <span>
                Background Refreshing... (Attempt: {failureCount + 1})
              </span>
            ) : (
              <span> </span>
            )}
          </div>
        </>
      )}
    </div>
  )
}

function AddTodo() {
  const queryClient = useQueryClient()
  const [name, setName] = React.useState('')

  const addMutation = useMutation({
    mutationFn: postTodo,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['todos'] })
    },
  })

  return (
    <div>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        disabled={addMutation.status === 'pending'}
      />
      <button
        onClick={() => {
          addMutation.mutate({ name, notes: 'These are some notes' })
        }}
        disabled={addMutation.status === 'pending' || !name}
      >
        Add Todo
      </button>
      <div>
        {addMutation.status === 'pending'
          ? 'Saving...'
          : addMutation.status === 'error'
          ? addMutation.error.message
          : 'Saved!'}
      </div>
    </div>
  )
}
```

```
function fetchTodos({ signal, queryKey: [, { filter }] }) {
  console.info('fetchTodos', { filter })

  if (signal) {
    signal.addEventListener('abort', () => {
      console.info('cancelled', filter)
    })
  }

  return new Promise((resolve, reject) => {
    setTimeout(
      () => {
        if (Math.random() < errorRate) {
          return reject(
            new Error(JSON.stringify({ fetchTodos: { filter } }, null, 2))
          )
        }
        resolve(list.filter((d) => d.name.includes(filter)))
      },
      queryTimeMin + Math.random() * (queryTimeMax - queryTimeMin)
    )
  })
}

function fetchTodoById({ id }) {
  console.info('fetchTodoById', { id })
  return new Promise((resolve, reject) => {
    setTimeout(
      () => {
        if (Math.random() < errorRate) {
          return reject(
            new Error(JSON.stringify({ fetchTodoById: { id } }, null, 2))
          )
        }
        resolve(list.find((d) => d.id === id))
      },
      queryTimeMin + Math.random() * (queryTimeMax - queryTimeMin)
    )
  })
}

function postTodo({ name, notes }) {
  console.info('postTodo', { name, notes })
  return new Promise((resolve, reject) => {
    setTimeout(
      () => {
        if (Math.random() < errorRate) {
          return reject(
            new Error(JSON.stringify({ postTodo: { name, notes } }, null, 2))
          )
        }
        const todo = { name, notes, id: id++ }
        list = [...list, todo]
        resolve(todo)
```

```
      },
      queryTimeMin + Math.random() * (queryTimeMax - queryTimeMin)
    )
  })
}

function patchTodo(todo) {
  console.info('patchTodo', todo)
  return new Promise((resolve, reject) => {
    setTimeout(
      () => {
        if (Math.random() < errorRate) {
          return reject(new Error(JSON.stringify({ patchTodo: todo }, null, 2)))
        }
        list = list.map((d) => {
          if (d.id === todo.id) {
            return todo
          }
          return d
        })
        resolve(todo)
      },
      queryTimeMin + Math.random() * (queryTimeMax - queryTimeMin)
    )
  })
}

const rootElement = document.getElementById('root')
ReactDOM.createRoot(rootElement).render(<Root />);
```

React Example: Prefetching

Github | StackBlitz | CodeSandbox

Code Explorer | Code | Interactive Sandbox

Project Structure

- src
    - pages
        - [user]
        - _app.tsx
        - index.tsx
- .gitignore
- README.md
- next-env.d.ts
- next.config.js
- package.json
- tsconfig.json

```
import React from 'react'
import { useQuery, useQueryClient } from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const getCharacters = async (): Promise<{
  results: Array<{ id: number; name: string }>
}> => {
  await new Promise((r) => setTimeout(r, 500))
  const response = await fetch('https://rickandmortyapi.com/api/character/')
  return await response.json()
}

const getCharacter = async (selectedChar: number) => {
  await new Promise((r) => setTimeout(r, 500))
  const response = await fetch(
    `https://rickandmortyapi.com/api/character/${selectedChar}`,
  )
  return await response.json()
}

export default function Example() {
  const queryClient = useQueryClient()
  const rerender = React.useState(0)[1]
  const [selectedChar, setSelectedChar] = React.useState(1)

  const charactersQuery = useQuery({
    queryKey: ['characters'],
    queryFn: getCharacters,
  })

  const characterQuery = useQuery({
```

```
  queryKey: ['character', selectedChar],
  queryFn: () => getCharacter(selectedChar),
})

return (
  <div className="App">
    <p>
      Hovering over a character will prefetch it, and when it's been
      prefetched it will turn <strong>bold</strong>. Clicking on a prefetched
      character will show their stats below immediately.
    </p>
    <h2>Characters</h2>
    {charactersQuery.isPending ? (
      'Loading...'
    ) : (
      <>
        <ul>
          {charactersQuery.data?.results.map((char) => (
            <li
              key={char.id}
              onClick={() => {
                setSelectedChar(char.id)
              }}
              onMouseEnter={async () => {
                await queryClient.prefetchQuery({
                  queryKey: ['character', char.id],
                  queryFn: () => getCharacter(char.id),
                  staleTime: 10 * 1000, // only prefetch if older than 10 seconds
                })

                setTimeout(() => {
                  rerender({})
                }, 1)
              }}
            >
              <div
                style={
                  queryClient.getQueryData(['character', char.id])
                    ? {
                        fontWeight: 'bold',
                      }
                    : {}
                }
              >
                {char.id} - {char.name}
              </div>
            </li>
          ))}
        </ul>

        <h3>Selected Character</h3>
        {characterQuery.isPending ? (
          'Loading...'
        ) : (
          <>
```

```
              <pre>{JSON.stringify(characterQuery.data, null, 2)}</pre>
            </>
          )}
          <ReactQueryDevtools initialIsOpen />
        </>
      )}
    </div>
  )
}
```

```
import React from 'react'
import { useQuery, useQueryClient } from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const getCharacters = async (): Promise<{
  results: Array<{ id: number; name: string }>
}> => {
  await new Promise((r) => setTimeout(r, 500))
  const response = await fetch('https://rickandmortyapi.com/api/character/')
  return await response.json()
}

const getCharacter = async (selectedChar: number) => {
  await new Promise((r) => setTimeout(r, 500))
  const response = await fetch(
    `https://rickandmortyapi.com/api/character/${selectedChar}`,
  )
  return await response.json()
}

export default function Example() {
  const queryClient = useQueryClient()
  const rerender = React.useState(0)[1]
  const [selectedChar, setSelectedChar] = React.useState(1)

  const charactersQuery = useQuery({
    queryKey: ['characters'],
    queryFn: getCharacters,
  })

  const characterQuery = useQuery({
    queryKey: ['character', selectedChar],
    queryFn: () => getCharacter(selectedChar),
  })

  return (
    <div className="App">
      <p>
        Hovering over a character will prefetch it, and when it's been
        prefetched it will turn <strong>bold</strong>. Clicking on a prefetched
        character will show their stats below immediately.
      </p>
      <h2>Characters</h2>
      {charactersQuery.isPending ? (
        'Loading...'
```

```
      ) : (
        <>
          <ul>
            {charactersQuery.data?.results.map((char) => (
              <li
                key={char.id}
                onClick={() => {
                  setSelectedChar(char.id)
                }}
                onMouseEnter={async () => {
                  await queryClient.prefetchQuery({
                    queryKey: ['character', char.id],
                    queryFn: () => getCharacter(char.id),
                    staleTime: 10 * 1000, // only prefetch if older than 10 seconds
                  })

                  setTimeout(() => {
                    rerender({})
                  }, 1)
                }}
              >
                <div
                  style={
                    queryClient.getQueryData(['character', char.id])
                      ? {
                          fontWeight: 'bold',
                        }
                      : {}
                  }
                >
                  {char.id} - {char.name}
                </div>
              </li>
            ))}
          </ul>

          <h3>Selected Character</h3>
          {characterQuery.isPending ? (
            'Loading...'
          ) : (
            <>
              <pre>{JSON.stringify(characterQuery.data, null, 2)}</pre>
            </>
          )}
          <ReactQueryDevtools initialIsOpen />
        </>
      )}
    </div>
  )
}
```

Our Partners

https://www.speakeasy.com/product/react-query?utm_source=tanstack&utm_campaign=tanstack

Want to Skip the Docs?

Query.gg - The Official React Query Course
“If you’re serious about really understanding React Query, there’s no better way than with query.gg” — Tanner Linsley

---

Learn More

Headless forms

https://tandstack.com/form
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

Build & Publish utilities

https://tandstack.com/config
The build and publish utilities used by all of our projects. Use it if you dare!

React TanStack Query Star Wars Example | TanStack Query Docs

TanStack Query v5

Navigation

- **Auto**
- Framework
  - React
- Version
  - Latest

Search...

- K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Overview react
- Installation react
- Quick Start react
- Devtools react
- Videos & Talks react
- Comparison react
- TypeScript react
- GraphQL react
- React Native react

Guides & Concepts

- Important Defaults react
- Queries react
- Query Keys react
- Query Functions react
- Query Options react
- Network Mode react
- Parallel Queries react
- Dependent Queries react
- Background Fetching Indicators react
- Window Focus Refetching react
- Disabling/Pausing Queries react
- Query Retries react

- Paginated Queries react
- Infinite Queries react
- Initial Query Data react
- Placeholder Query Data react
- Mutations react
- Query Invalidation react
- Invalidation from Mutations react
- Updates from Mutation Responses react
- Optimistic Updates react
- Query Cancellation react
- Scroll Restoration react
- Filters react
- Performance & Request Waterfalls react
- Prefetching & Router Integration react
- Server Rendering & Hydration react
- Advanced Server Rendering react
- Caching react
- Render Optimizations react
- Default Query Fn react
- Suspense react
- Testing react
- Does this replace [Redux, MobX, etc]? react
- Migrating to v3 react
- Migrating to v4 react
- Migrating to v5 react

API Reference

- QueryClient core
- QueryCache core
- MutationCache core
- QueryObserver core
- InfiniteQueryObserver core
- QueriesObserver core
- streamedQuery core
- focusManager core
- onlineManager core
- notifyManager core
- useQuery react
- useQueries react
- useInfiniteQuery react
- useMutation react
- useIsFetching react
- useIsMutating react
- useMutationState react
- useSuspenseQuery react
- useSuspenseInfiniteQuery react
- useSuspenseQueries react
- QueryClientProvider react
- useQueryClient react
- queryOptions react
- infiniteQueryOptions react
- usePreFetchQuery react
- usePrefetchInfiniteQuery react
- QueryErrorResetBoundary react
- useQueryErrorResetBoundary react
- hydration react

ESLint

- ESLint Plugin Query core
- Exhaustive Deps core
- Stable Query Client core
- No Rest Destructuring core
- No Unstable Deps core
- Infinite Query Property Order core

Community Resources

- TkDodo's Blog react
- Community Projects react

Examples

- Simple react
- Basic react
- Basic w/ GraphQL-Request react
- Auto Refetching / Polling / Realtime react
- Optimistic Updates (UI) react
- Optimistic Updates (Cache) react
- Pagination react
- Load-More & Infinite Scroll react
- Infinite query with Max pages react
- Suspense react
- Default Query Function react
- Playground react
- Prefetching react
- Star Wars react
- Rick And Morty react
- Next.js Pages react
- Next.js app with prefetching react
- Next.js app with streaming react
- React Native react
- React Router react
- Offline Queries and Mutations react
- Algolia react
- Shadow DOM react
- Devtools Embedded Panel react
- Chat example (streaming) react

Plugins

- persistQueryClient react
- createSyncStoragePersister react
- createAsyncStoragePersister react
- broadcastQueryClient (Experimental) react
- createPersister (Experimental) react

React Example: Star Wars

Github
StackBlitz
CodeSandbox

Code Explorer & Sandbox

- **Code Explorer**
- **Interactive Sandbox**

Project Structure

- `public`
- `src`
  - App.tsx
  - Character.jsx
  - Characters.jsx
  - Film.jsx
  - Films.jsx
  - Home.jsx
  - Layout.tsx
  - fetch.jsx
  - index.tsx
  - styles.css
- `.eslintrc`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.ts`

Setup Code

```
import ReactDOM from 'react-dom/client'
import App from './App'

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

Additional Links

- Prefetching
- Rick And Morty

Our Partners

More Resources

- TanStack Table

Supercharge your tables or build a datagrid from scratch for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, and Lit while retaining 100% control over markup and styles.

- TanStack DB

  TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥

React Example: Rick Morty

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

Files:

- `public`
- `src`
  - `App.tsx`
  - `Character.jsx`
  - `Characters.jsx`
  - `Episode.jsx`
  - `Episodes.jsx`
  - `Home.jsx`
  - `Layout.tsx`
  - `fetch.jsx`
  - `index.tsx`
  - `styles.css`
- `.eslintrc`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.ts`

React Entry Point:

```
import ReactDOM from 'react-dom/client'
import App from './App'

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

Related Links

- Star Wars
- Next.js Pages

Our Partners

Skip the Docs?

> **Query.gg - The Official React Query Course**
> *"If you're serious about really understanding React Query, there's no better way than with query.gg"* — Tanner Linsley

---

More Resources

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.

The build and publish utilities used by all of our projects. Use it if you dare!

React Example: Nextjs

Github
StackBlitz
CodeSandbox

Code Explorer & Interactive Sandbox

Directory Structure

- `src/`
  - `components/`
  - `hooks/`
  - `pages/`
    - `_app.tsx`
    - `client-only.tsx`
    - `index.tsx`
- `.gitignore`
- `README.md`
- `next-env.d.ts`
- `next.config.js`
- `package.json`
- `tsconfig.json`

Example Code

```
import React from 'react'
import { QueryClient, dehydrate } from '@tanstack/react-query'
import { Header, InfoBox, Layout, PostList } from '../components'
import { fetchPosts } from '../hooks/usePosts'

const Home = () => {
  return (
    <Layout>
      <Header />
      <InfoBox>ℹ️ This page shows how to use SSG with React-Query.</InfoBox>
      <PostList />
    </Layout>
  )
}

export async function getStaticProps() {
  const queryClient = new QueryClient()

  await queryClient.prefetchQuery({
    queryKey: ['posts', 10],
    queryFn: () => fetchPosts(10),
  })

  return {
```

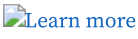
```
    props: {
      dehydratedState: dehydrate(queryClient),
    },
  }
}

export default Home
```

Additional Links

- Rick And Morty
- Next.js app with prefetching

Our Partners



> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg” — Tanner Linsley

---

Related Tools and Resources

- TanStack Table: Supercharge your tables or build a datagrid for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, and Lit while retaining 100% control over markup and styles.
- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations for a reactive, consistent, and blazing-fast UI.

Skip the Docs?

Learn more at query.gg

> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg” — Tanner Linsley

React Example: Nextjs App Prefetching

Github
StackBlitz
CodeSandbox

Code Explorer / Interactive Sandbox

App Structure

- favicon.ico
- get-query-client.ts
- layout.tsx
- page.tsx
- pokemon-info.tsx
- pokemon.ts
- providers.tsx
- .eslintrc.cjs
- .gitignore
- README.md
- next.config.js
- package.json
- tsconfig.json

Sample Code

```
import React from 'react'
import { HydrationBoundary, dehydrate } from '@tanstack/react-query'
import { pokemonOptions } from '@/app/pokemon'
import { getQueryClient } from '@/app/get-query-client'
import { PokemonInfo } from './pokemon-info'

export default function Home() {
  const queryClient = getQueryClient()

  void queryClient.prefetchQuery(pokemonOptions)

  return (
    <main>
      <h1>Pokemon Info</h1>
      <HydrationBoundary state={dehydrate(queryClient)}>
        <PokemonInfo />
      </HydrationBoundary>
    </main>
  )
}
```

Related Links

- Next.js Pages
- Next.js app with streaming

Our Partners

Additional Resources

- TanStack Table: Supercharge your tables or build a datagrid for TS/JS, React, Vue, Solid, Svelte, Qwik, Angular, Lit with full control over markup and styles.
- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations for reactive, consistent, and blazing-fast UI.

More About Query.gg

React Example: Nextjs Suspense Streaming

Github | StackBlitz | CodeSandbox

Code Explorer

Code

Interactive Sandbox

Directory Structure

- `src`
  - `app`
    - `api`
    - `layout.tsx`
    - `page.tsx`
    - `providers.tsx`
- `next-env.d.ts`
- `next.config.js`
- `package.json`
- `tsconfig.json`

Example Code (TypeScript for Next.js)

```
'use client'
import { isServer, useSuspenseQuery } from '@tanstack/react-query'
import { Suspense } from 'react'

export const runtime = 'edge' // 'nodejs' (default) | 'edge'

function getBaseURL() {
  if (!isServer) {
    return ''
  }
  if (process.env.VERCEL_URL) {
    return `https://${process.env.VERCEL_URL}`
  }
  return 'http://localhost:3000'
}
const baseUrl = getBaseURL()

function useWaitQuery(props: { wait: number }) {
  const query = useSuspenseQuery({
    queryKey: ['wait', props.wait],
```

```
    queryFn: async () => {
      const path = `/api/wait?wait=${props.wait}`
      const url = baseUrl + path

      const res: string = await (
        await fetch(url, {
          cache: 'no-store',
        })
      ).json()
      return res
    },
  })

  return [query.data as string, query] as const
}

function MyComponent(props: { wait: number }) {
  const [data] = useWaitQuery(props)

  return <div>result: {data}</div>
}

export default function MyPage() {
  return (
    <>
      <Suspense fallback={<div>waiting 100....</div>}>
        <MyComponent wait={100} />
      </Suspense>
      <Suspense fallback={<div>waiting 200....</div>}>
        <MyComponent wait={200} />
      </Suspense>
      <Suspense fallback={<div>waiting 300....</div>}>
        <MyComponent wait={300} />
      </Suspense>
      <Suspense fallback={<div>waiting 400....</div>}>
        <MyComponent wait={400} />
      </Suspense>
      <Suspense fallback={<div>waiting 500....</div>}>
        <MyComponent wait={500} />
      </Suspense>
      <Suspense fallback={<div>waiting 600....</div>}>
        <MyComponent wait={600} />
      </Suspense>
      <Suspense fallback={<div>waiting 700....</div>}>
        <MyComponent wait={700} />
      </Suspense>

      <fieldset>
        <legend>
          <code>Suspense</code>-container
        </legend>
        <Suspense
          fallback={
            <>
              <div>waiting 800....</div>
```

```
              <div>waiting 900....</div>
              <div>waiting 1000....</div>
            </>
          }
        >
          <MyComponent wait={800} />
          <MyComponent wait={900} />
          <MyComponent wait={1000} />
        </Suspense>
      </fieldset>
    </>
  )
}
```

Additional Information

- This example demonstrates React Suspense with streaming data fetching using TanStack Query in Next.js.
- Adjust the `wait` parameter to simulate different loading times.
- Use the provided links to view the example on GitHub, StackBlitz, or CodeSandbox.

React Example: React Native

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

- assets
- src
- .eslintrc
- .gitignore
- App.tsx
- README.md
- app.json
- babel.config.js
- package.json
- tsconfig.json

```
import * as React from 'react'
import { AppStateStatus, Platform } from 'react-native'
import { NavigationContainer } from '@react-navigation/native'
import {
  QueryClient,
  QueryClientProvider,
  focusManager,
} from '@tanstack/react-query'

import { useAppState } from './src/hooks/useAppState'
import { MoviesStack } from './src/navigation/MoviesStack'
import { useOnlineManager } from './src/hooks/useOnlineManager'

function onAppStateChange(status: AppStateStatus) {
  // React Query already supports in web browser refetch on window focus by default
  if (Platform.OS !== 'web') {
    focusManager.setFocused(status === 'active')
  }
}

const queryClient = new QueryClient({
  defaultOptions: { queries: { retry: 2 } },
})

export default function App() {
  useOnlineManager()

  useAppState(onAppStateChange)

  return (
    <QueryClientProvider client={queryClient}>
```

```
      <NavigationContainer>
        <MoviesStack />
      </NavigationContainer>
    </QueryClientProvider>
  )
}
```

Our Partners

Want to Skip the Docs?
Query.gg - The Official React Query Course

> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg”
> — Tanner Linsley

Additional Resources

Learn More

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.
Learn More

Community & More

Next.js app with streaming
React Router

React Example: React Router

Github
StackBlitz
CodeSandbox

Code Explorer

Code	Sandbox
`<Your code here>`	Interactive Sandbox

```
export default function Index() {
  return (
    <p id="zero-state">
      This is a demo for integrating React Router with React Query.
      <br />
      Check out{' '}
      <a href="https://reactrouter.com/">
        the docs at reactrouter.com
      </a> and{' '}
      <a href="https://tanstack.com/query/v5/docs/">
        the docs at tanstack.com
      </a>.
    </p>
  );
}
```

Our Partners

Additional Resources

- TanStack Form
  Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

- TanStack Config
  The build and publish utilities used by all of our projects. Use it if you dare!

React Example: Basic GraphQL Request

Github
StackBlitz
CodeSandbox

Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import { gql, request } from 'graphql-request'

const endpoint = 'https://graphqlzero.almansi.me/api'

const queryClient = new QueryClient()

type Post = {
  id: number
  title: string
  body: string
}

function App() {
  const [postId, setPostId] = React.useState(-1)

  return (
    <QueryClientProvider client={queryClient}>
      <p>
        As you visit the posts below, you will notice them in a loading state
        the first time you load them. However, after you return to this list and
        click on any posts you have already visited again, you will see them
        load instantly and background refresh right before your eyes!{' '}
        <strong>
          (You may need to throttle your network speed to simulate longer
          loading sequences)
        </strong>
      </p>
      {postId > -1 ? (
        <Post postId={postId} setPostId={setPostId} />
      ) : (
        <Posts setPostId={setPostId} />
      )}
      <ReactQueryDevtools initialIsOpen />
```

```
    </QueryClientProvider>
  )
}

function usePosts() {
  return useQuery({
    queryKey: ['posts'],
    queryFn: async () => {
      const {
        posts: { data },
      } = await request<{ posts: { data: Array<Post> } }>(
        endpoint,
        gql`
          query {
            posts {
              data {
                id
                title
              }
            }
          }
        `
      )
      return data
    },
  })
}

function Posts({
  setPostId,
}: {
  setPostId: React.Dispatch<React.SetStateAction<number>>
}) {
  const queryClient = useQueryClient()
  const { status, data, error, isFetching } = usePosts()

  return (
    <div>
      <h1>Posts</h1>
      <div>
        {status === 'pending' ? (
          'Loading...'
        ) : status === 'error' ? (
          <span>Error: {error.message}</span>
        ) : (
          <>
            <div>
              {data.map((post) => (
                <p key={post.id}>
                  <a
                    onClick={() => setPostId(post.id)}
                    href="#"
                    style={
                      // Show bold links for cached posts
                      queryClient.getQueryData(['post', post.id])
```

```
                        ? {
                            fontWeight: 'bold',
                            color: 'green',
                          }
                        : {}
                    }
                  >
                    {post.title}
                  </a>
                </p>
              ))}
            </div>
            <div>{isFetching ? 'Background Updating...' : ' '}</div>
          </>
        )}
      </div>
    </div>
  )
}

function usePost(postId: number) {
  return useQuery({
    queryKey: ['post', postId],
    queryFn: async () => {
      const { post } = await request<{ post: Post }>(
        endpoint,
        gql`
        query {
          post(id: ${postId}) {
            id
            title
            body
          }
        }
        `
      )

      return post
    },
    enabled: !!postId,
  })
}

function Post({
  postId,
  setPostId,
}: {
  postId: number
  setPostId: React.Dispatch<React.SetStateAction<number>>
}) {
  const { status, data, error, isFetching } = usePost(postId)

  return (
    <div>
      <div>
```

```
        <a onClick={() => setPostId(-1)} href="#">
          Back
        </a>
      </div>
      {!postId || status === 'pending' ? (
        'Loading...'
      ) : status === 'error' ? (
        <span>Error: {error.message}</span>
      ) : (
        <>
          <h1>{data.title}</h1>
          <div>
            <p>{data.body}</p>
          </div>
          <div>{isFetching ? 'Background Updating...' : ' '}</div>
        </>
      )}
    </div>
  )
}

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

React Example: Offline

Github
StackBlitz
CodeSandbox

Code Explorer

Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import { worker } from './api'

worker.start()

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(
  <div style={{ padding: '16px' }}>
    <App />
  </div>,
)
```

Interactive Sandbox

Navigation

- React Router
- Algolia

Our Partners

- React Query Course (query.gg) (Opens in new tab)

Skip the Docs

Query.gg - The Official React Query Course
"If you're serious about really understanding React Query, there's no better way than with query.gg" — Tanner Linsley

Learn More

Additional Resources

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

Learn More

Build and publish utilities used by all of our projects.

Learn More

React Example: Algolia

Github
StackBlitz
CodeSandbox

Code Explorer

Code | Sandbox

Files:

- `src/App.tsx`
- `src/Search.tsx`
- `src/SearchResults.tsx`
- `src/algolia.ts`
- `src/index.tsx`
- `src/styles.css`
- `src/useAlgolia.ts`
- `.gitignore`
- `README.md`
- `eslint.config.js`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.ts`

React Setup

```
import ReactDOM from 'react-dom/client'

import App from './App'

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

Additional Links

- Offline Queries and Mutations
- Shadow DOM

Our Partners

> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg”
> —Tanner Linsley

Other Projects

TanStack Form

Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

TanStack Config

The build and publish utilities used by all of our projects. Use it if you dare!

React Example: Shadow Dom

Github | StackBlitz | CodeSandbox

Code Explorer

Code | **Sandbox**

Src Files

- DogList.tsx
- index.css
- main.tsx
- vite-env.d.ts
- .gitignore
- README.md
- eslint.config.js
- index.html
- package.json
- tsconfig.json
- vite.config.ts

TypeScript Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import { DogList } from './DogList'

const appRoot = document.getElementById('root')

if (appRoot) {
  const queryClient = new QueryClient()
  const shadowRoot = appRoot.attachShadow({ mode: 'open' })
  const root = ReactDOM.createRoot(shadowRoot)

  root.render(
    <React.StrictMode>
      <QueryClientProvider client={queryClient}>
        <div
          style={{
            width: '100vw',
            padding: '30px',
          }}
        >
          <h2>Dog Breeds</h2>
          <DogList />
        </div>
```

```
        <ReactQueryDevtools
          initialIsOpen={false}
          buttonPosition="bottom-left"
          shadowDOMTarget={appRoot.shadowRoot!}
        />
      </QueryClientProvider>
    </React.StrictMode>,
  )
}
```

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import { DogList } from './DogList'

const appRoot = document.getElementById('root')

if (appRoot) {
  const queryClient = new QueryClient()
  const shadowRoot = appRoot.attachShadow({ mode: 'open' })
  const root = ReactDOM.createRoot(shadowRoot)

  root.render(
    <React.StrictMode>
      <QueryClientProvider client={queryClient}>
        <div
          style={{
            width: '100vw',
            padding: '30px',
          }}
        >
          <h2>Dog Breeds</h2>
          <DogList />
        </div>
        <ReactQueryDevtools
          initialIsOpen={false}
          buttonPosition="bottom-left"
          shadowDOMTarget={appRoot.shadowRoot!}
        />
      </QueryClientProvider>
    </React.StrictMode>,
  )
}
```

Additional Resources

Algolia | Devtools Embedded Panel

Our Partners

Query.gg - The Official React Query Course

> “If you’re serious about *really* understanding React Query, there’s no better way than with query.gg”
> — Tanner Linsley

Additional Links

TanStack Form
Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit and Svelte.

TanStack Config
The build and publish utilities used by all of our projects.

React Example: Devtools Panel

Github | StackBlitz | CodeSandbox

Code Explorer / Interactive Sandbox

Files:

- `public/`
- `src/`
  - `index.tsx`
- `.eslintrc`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.ts`

React Code Example:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  useQuery,
} from '@tanstack/react-query'
import { ReactQueryDevtoolsPanel } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient()

export default function App() {
  const [isOpen, setIsOpen] = React.useState(false)

  return (
    <QueryClientProvider client={queryClient}>
      <Example />
      <button onClick={() => setIsOpen(!isOpen)}>
        {`${isOpen ? 'Close' : 'Open'} the devtools panel`}
      </button>
      {isOpen && (
        <ReactQueryDevtoolsPanel onClose={() => setIsOpen(false)} />
      )}
    </QueryClientProvider>
  )
}

function Example() {
  const { isPending, error, data, isFetching } = useQuery({
```

```
    queryKey: ['repoData'],
    queryFn: async () => {
      const response = await fetch(
        'https://api.github.com/repos/TanStack/query'
      )
      return await response.json()
    },
  })

  if (isPending) return 'Loading...'

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div style={{ paddingBottom: 20 }}>
      <h1>{data.full_name}</h1>
      <p>{data.description}</p>
      <strong>👀 {data.subscribers_count}</strong>{' '}
      <strong>✨ {data.stargazers_count}</strong>{' '}
      <strong>🍴 {data.forks_count}</strong>
      <div>{isFetching ? 'Updating...' : ''}</div>
    </div>
  )
}

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

React Example: Chat

Github | StackBlitz | CodeSandbox

Code Explorer

Code

```
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  useQuery,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

import './style.css'
import { useState } from 'react'
import { chatQueryOptions } from './chat'
import { Message } from './message'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <ReactQueryDevtools />
      <Example />
    </QueryClientProvider>
  )
}

function ChatMessage({ question }: { question: string }) {
  const { error, data = [], isFetching } = useQuery(chatQueryOptions(question))

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <Message message={{ content: question, isQuestion: true }} />
      <Message
        inProgress={isFetching}
        message={{ content: data.join(' '), isQuestion: false }}
      />
    </div>
  )
}

function Example() {
  const [questions, setQuestions] = useState<Array<string>>([])
  const [currentQuestion, setCurrentQuestion] = useState('')
```

```
  const submitMessage = () => {
    setQuestions([...questions, currentQuestion])
    setCurrentQuestion('')
  }

  return (
    <div className="flex flex-col h-screen max-w-3xl mx-auto p-4">
      <h1 className="text-3xl font-bold text-gray-800">
        TanStack Chat Example
      </h1>
      <div className="overflow-y-auto mb-4 space-y-4">
        {questions.map((question) => (
          <ChatMessage key={question} question={question} />
        ))}
      </div>

      <div className="flex items-center space-x-2">
        <input
          className="flex-1 p-2 border rounded-lg focus:outline-none focus:ring-2 focu
          value={currentQuestion}
          onChange={(e) => setCurrentQuestion(e.target.value)}
          onKeyDown={(e) => {
            if (e.key === 'Enter') {
              submitMessage()
            }
          }}
          placeholder="Type your message..."
        />
        <button
          onClick={submitMessage}
          disabled={!currentQuestion.trim()}
          className="flex items-center gap-2 bg-blue-600 hover:bg-blue-700 text-white
        >
          <span>Send</span>
          <svg
            xmlns="http://www.w3.org/2000/svg"
            width="24"
            height="24"
            viewBox="0 0 24 24"
            fill="none"
            stroke="currentColor"
            strokeWidth="2"
            strokeLinecap="round"
            strokeLinejoin="round"
          >
            <path d="M14.536 21.686a.5.5 0 0 0 .937-.024l6.5-19a.496.496 0 0 0-.635-.6
            <path d="m21.854 2.147-10.94 10.939" />
          </svg>
        </button>
      </div>
    </div>
  )
}
```

```
const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />)
```

[Same code repeated for reference]

React Example: Auto Refetching

GitHub | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

- `src`
  - `pages`
    - `api`
      - `index.tsx`
  - `.gitignore`
  - `README.md`
  - `next-env.d.ts`
  - `next.config.js`
  - `package.json`
  - `tsconfig.json`

Example Code

```
import React from 'react'

import {
  QueryClient,
  QueryClientProvider,
  useMutation,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const queryClient = useQueryClient()
  const [intervalMs, setIntervalMs] = React.useState(1000)
  const [value, setValue] = React.useState('')

  const { status, data, error, isFetching } = useQuery({
    queryKey: ['todos'],
    queryFn: async (): Promise<Array<string>> => {
      const response = await fetch('/api/data')
      return await response.json()
    },
    // Refetch the data every second
```

```
  refetchInterval: intervalMs,
})

const addMutation = useMutation({
  mutationFn: (add: string) => fetch(`/api/data?add=${add}`),
  onSuccess: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
})

const clearMutation = useMutation({
  mutationFn: () => fetch(`/api/data?clear=1`),
  onSuccess: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
})

if (status === 'pending') return <h1>Loading...</h1>
if (status === 'error') return <span>Error: {error.message}</span>

return (
  <div>
    <h1>Auto Refetch with stale-time set to {intervalMs}ms</h1>
    <p>
      This example is best experienced on your own machine, where you can open
      multiple tabs to the same localhost server and see your changes
      propagate between the two.
    </p>
    <label>
      Query Interval speed (ms):{' '}
      <input
        value={intervalMs}
        onChange={(ev) => setIntervalMs(Number(ev.target.value))}
        type="number"
        step="100"
      />{' '}
      <span
        style={{
          display: 'inline-block',
          marginLeft: '.5rem',
          width: 10,
          height: 10,
          background: isFetching ? 'green' : 'transparent',
          transition: !isFetching ? 'all .3s ease' : 'none',
          borderRadius: '100%',
          transform: 'scale(2)',
        }}
      />
    </label>
    <h2>Todo List</h2>
    <form
      onSubmit={(event) => {
        event.preventDefault()
        addMutation.mutate(value, {
          onSuccess: () => {
            setValue('')
          },
        })
      }}
```

```
      >
        <input
          placeholder="enter something"
          value={value}
          onChange={(ev) => setValue(ev.target.value)}
        />
      </form>
      <ul>
        {data.map((item) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
      <div>
        <button
          onClick={() => {
            clearMutation.mutate()
          }}
        >
          Clear All
        </button>
      </div>
      <ReactQueryDevtools initialIsOpen />
    </div>
  )
}
```

React Example: Optimistic Updates UI

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

File Structure

- `src`
  - `pages`
    - `api`
    - `index.tsx`
- `.gitignore`
- `README.md`
- `next-env.d.ts`
- `next.config.js`
- `package.json`
- `tsconfig.json`

Example Code

```
import * as React from 'react'
import {
  QueryClient,
  QueryClientProvider,
  useMutation,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const client = new QueryClient()

type Todos = {
  items: ReadonlyArray<{
    id: string
    text: string
  }>
  ts: number
}

async function fetchTodos(): Promise<Todos> {
  const response = await fetch('/api/data')
  return await response.json()
}

function useTodos() {
```

```
  return useQuery({ queryKey: ['todos'], queryFn: fetchTodos })
}

function Example() {
  const queryClient = useQueryClient()
  const [text, setText] = React.useState('')
  const todoQuery = useTodos()

  const addTodoMutation = useMutation({
    mutationFn: async (newTodo: string) => {
      const response = await fetch('/api/data', {
        method: 'POST',
        body: JSON.stringify({ text: newTodo }),
        headers: { 'Content-Type': 'application/json' },
      })
      if (!response.ok) {
        throw new Error('Something went wrong.')
      }
      return await response.json()
    },
    onSettled: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
  })

  return (
    <div>
      <p>
        In this example, new items can be created using a mutation. The new item
        will be optimistically added to the list in hopes that the server
        accepts the item. If it does, the list is refetched with the true items
        from the list. Every now and then, the mutation may fail though. When
        that happens, the previous list of items is restored and the list is
        again refetched from the server.
      </p>
      <form
        onSubmit={(e) => {
          e.preventDefault()
          setText('')
          addTodoMutation.mutate(text)
        }}
      >
        <input
          type="text"
          onChange={(event) => setText(event.target.value)}
          value={text}
        />
        <button disabled={addTodoMutation.isPending}>Create</button>
      </form>
      <br />
      {todoQuery.isSuccess && (
        <>
          <div>
            {/* The type of queryInfo.data will be narrowed because we check for isSuc
            Updated At: {new Date(todoQuery.data.ts).toLocaleTimeString()}
          </div>
          <ul>
```

```
            {todoQuery.data.items.map((todo) => (
              <li key={todo.id}>{todo.text}</li>
            ))}
            {addTodoMutation.isPending && (
              <li style={{ opacity: 0.5 }}>{addTodoMutation.variables}</li>
            )}
            {addTodoMutation.isError && (
              <li style={{ color: 'red' }}>
                {addTodoMutation.variables}
                <button
                  onClick={() =>
                    addTodoMutation.mutate(addTodoMutation.variables)
                  }
                >
                  Retry
                </button>
              </li>
            )}
          </ul>
          {todoQuery.isFetching && <div>Updating in background...</div>}
        </>
      )}
      {todoQuery.isPending && 'Loading'}
      {todoQuery.error instanceof Error && todoQuery.error.message}
    </div>
  )
}

export default function App() {
  return (
    <QueryClientProvider client={client}>
      <Example />
      <ReactQueryDevtools initialIsOpen />
    </QueryClientProvider>
  )
}
```

React Example: Optimistic Updates Cache

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

Project Structure

- `src`
  - `pages`
    - `api`
    - `index.tsx`
- `.gitignore`
- `README.md`
- `next-env.d.ts`
- `next.config.js`
- `package.json`
- `tsconfig.json`

Example Code

```
import * as React from 'react'

import {
  QueryClient,
  QueryClientProvider,
  queryOptions,
  useMutation,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const client = new QueryClient()

type Todos = {
  items: ReadonlyArray<{
    id: string
    text: string
  }>
  ts: number
}

async function fetchTodos(): Promise<Todos> {
  const response = await fetch('/api/data')
  return await response.json()
}
```

```
const todoListOptions = queryOptions({
  queryKey: ['todos'],
  queryFn: fetchTodos,
})

function Example() {
  const queryClient = useQueryClient()
  const [text, setText] = React.useState('')
  const { isFetching, ...queryInfo } = useQuery(todoListOptions)

  const addTodoMutation = useMutation({
    mutationFn: async (newTodo: string) => {
      const response = await fetch('/api/data', {
        method: 'POST',
        body: JSON.stringify({ text: newTodo }),
        headers: { 'Content-Type': 'application/json' },
      })
      return await response.json()
    },
    // When mutate is called:
    onMutate: async (newTodo: string) => {
      setText('')
      // Cancel any outgoing refetch
      // (so they don't overwrite our optimistic update)
      await queryClient.cancelQueries(todoListOptions)

      // Snapshot the previous value
      const previousTodos = queryClient.getQueryData(todoListOptions.queryKey)

      // Optimistically update to the new value
      if (previousTodos) {
        queryClient.setQueryData(todoListOptions.queryKey, {
          ...previousTodos,
          items: [
            ...previousTodos.items,
            { id: Math.random().toString(), text: newTodo },
          ],
        })
      }

      return { previousTodos }
    },
    // If the mutation fails,
    // use the context returned from onMutate to roll back
    onError: (err, variables, context) => {
      if (context?.previousTodos) {
        queryClient.setQueryData<Todos>(['todos'], context.previousTodos)
      }
    },
    // Always refetch after error or success:
    onSettled: () => queryClient.invalidateQueries({ queryKey: ['todos'] }),
  })

  return (
```

```
    <div>
      <p>
        In this example, new items can be created using a mutation. The new item
        will be optimistically added to the list in hopes that the server
        accepts the item. If it does, the list is refetched with the true items
        from the list. Every now and then, the mutation may fail though. When
        that happens, the previous list of items is restored and the list is
        again refetched from the server.
      </p>
      <form
        onSubmit={(e) => {
          e.preventDefault()
          addTodoMutation.mutate(text)
        }}
      >
        <input
          type="text"
          onChange={(event) => setText(event.target.value)}
          value={text}
        />
        <button disabled={addTodoMutation.isPending}>Create</button>
      </form>
      <br />
      {queryInfo.isSuccess && (
        <>
          <div>
            {/* The type of queryInfo.data will be narrowed because we check for isSuc
            Updated At: {new Date(queryInfo.data.ts).toLocaleTimeString()}
          </div>
          <ul>
            {queryInfo.data.items.map((todo) => (
              <li key={todo.id}>{todo.text}</li>
            ))}
          </ul>
          {isFetching && <div>Updating in background...</div>}
        </>
      )}
      {queryInfo.isLoading && 'Loading'}
      {queryInfo.error instanceof Error && queryInfo.error.message}
    </div>
  )
}

export default function App() {
  return (
    <QueryClientProvider client={client}>
      <Example />
      <ReactQueryDevtools initialIsOpen />
    </QueryClientProvider>
  )
}
```

React Example: Pagination

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

File Structure

- `src/`
  - `pages/`
    - `api/`
      - `index.tsx`
  - `.gitignore`
  - `README.md`
  - `next-env.d.ts`
  - `next.config.js`
  - `package.json`
  - `tsconfig.json`

Example Code

```
import React from 'react'
import {
  QueryClient,
  QueryClientProvider,
  keepPreviousData,
  useQuery,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

const fetchProjects = async (
  page = 0,
): Promise<{
  projects: Array<{ name: string; id: number }>
  hasMore: boolean
}> => {
  const response = await fetch(`/api/projects?page=${page}`)
  return await response.json()
}

function Example() {
```

```
const queryClient = useQueryClient()
const [page, setPage] = React.useState(0)

const { status, data, error, isFetching, isPlaceholderData } = useQuery({
  queryKey: ['projects', page],
  queryFn: () => fetchProjects(page),
  placeholderData: keepPreviousData,
  staleTime: 5000,
})

// Prefetch the next page!
React.useEffect(() => {
  if (!isPlaceholderData && data?.hasMore) {
    queryClient.prefetchQuery({
      queryKey: ['projects', page + 1],
      queryFn: () => fetchProjects(page + 1),
    })
  }
}, [data, isPlaceholderData, page, queryClient])

return (
  <div>
    <p>
      In this example, each page of data remains visible as the next page is
      fetched. The buttons and capability to proceed to the next page are also
      suppressed until the next page cursor is known. Each page is cached as a
      normal query too, so when going to previous pages, you'll see them
      instantaneously while they are also refetched invisibly in the background.
    </p>
    {status === 'pending' ? (
      <div>Loading...</div>
    ) : status === 'error' ? (
      <div>Error: {error.message}</div>
    ) : (
      // `data` will either resolve to the latest page's data
      // or if fetching a new page, the last successful page's data
      <div>
        {data.projects.map((project) => (
          <p key={project.id}>{project.name}</p>
        ))}
      </div>
    )}
    <div>Current Page: {page + 1}</div>
    <button
      onClick={() => setPage((old) => Math.max(old - 1, 0))}
      disabled={page === 0}
    >
      Previous Page
    </button>{' '}
    <button
      onClick={() => {
        setPage((old) => (data?.hasMore ? old + 1 : old))
      }}
      disabled={isPlaceholderData || !data?.hasMore}
    >
```

```
        Next Page
      </button>
      {
        // Since the last page's data potentially sticks around between page requests,
        // we can use `isFetching` to show a background loading indicator
        // since our `status === 'pending'` state won't be triggered
        isFetching ? <span> Loading...</span> : null
      }{' '}
      <ReactQueryDevtools initialIsOpen />
    </div>
  )
}
```

React Example: Load More Infinite Scroll

Github | StackBlitz | CodeSandbox

Code Explorer & Interactive Sandbox

Source Files

- src
  - pages
    - api
    - about.tsx
    - index.tsx
  - .gitignore
  - README.md
  - next-env.d.ts
  - next.config.js
  - package.json
  - tsconfig.json

Example Code (TypeScript)

```
import React from 'react'
import Link from 'next/link'
import { useInView } from 'react-intersection-observer'
import {
  useInfiniteQuery,
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { ref, inView } = useInView()

  const {
    status,
    data,
    error,
```

```
  isFetching,
  isFetchingNextPage,
  isFetchingPreviousPage,
  fetchNextPage,
  fetchPreviousPage,
  hasNextPage,
  hasPreviousPage,
} = useInfiniteQuery({
  queryKey: ['projects'],
  queryFn: async ({ pageParam }): Promise<{
    data: Array<{ name: string; id: number }>
    previousId: number
    nextId: number
  }> => {
    const response = await fetch(`/api/projects?cursor=${pageParam}`)
    return await response.json()
  },
  initialPageParam: 0,
  getPreviousPageParam: (firstPage) => firstPage.previousId,
  getNextPageParam: (lastPage) => lastPage.nextId,
})

React.useEffect(() => {
  if (inView) {
    fetchNextPage()
  }
}, [fetchNextPage, inView])

return (
  <div>
    <h1>Infinite Loading</h1>
    {status === 'pending' ? (
      <p>Loading...</p>
    ) : status === 'error' ? (
      <span>Error: {error.message}</span>
    ) : (
      <>
        <div>
          <button
            onClick={() => fetchPreviousPage()}
            disabled={!hasPreviousPage || isFetchingPreviousPage}
          >
            {isFetchingPreviousPage
              ? 'Loading more...'
              : hasPreviousPage
              ? 'Load Older'
              : 'Nothing more to load'}
          </button>
        </div>
        {data.pages.map((page) => (
          <React.Fragment key={page.nextId}>
            {page.data.map((project) => (
              <p
                style={{
                  border: '1px solid gray',
```

```
                  borderRadius: '5px',
                  padding: '10rem 1rem',
                  background: `hsla(${project.id * 30}, 60%, 80%, 1)`,
                }}
                key={project.id}
              >
                {project.name}
              </p>
            ))}
          </React.Fragment>
        ))}
        <div>
          <button
            ref={ref}
            onClick={() => fetchNextPage()}
            disabled={!hasNextPage || isFetchingNextPage}
          >
            {isFetchingNextPage
              ? 'Loading more...'
              : hasNextPage
              ? 'Load Newer'
              : 'Nothing more to load'}
          </button>
        </div>
        <div>
          {isFetching && !isFetchingNextPage
            ? 'Background Updating...'
            : null}
        </div>
      </>
    )}
    <hr />
    <Link href="/about">Go to another page</Link>
    <ReactQueryDevtools initialIsOpen />
  </div>
 )
}
```

(Note: The same code appears twice in the provided content; only one version is included here for clarity.)

React Example: Infinite Query With Max Pages

Github | StackBlitz | CodeSandbox

Code

```
import React from 'react'
import {
  QueryClient,
  QueryClientProvider,
  useInfiniteQuery,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const {
    status,
    data,
    error,
    isFetching,
    isFetchingNextPage,
    isFetchingPreviousPage,
    fetchNextPage,
    fetchPreviousPage,
    hasNextPage,
    hasPreviousPage,
  } = useInfiniteQuery({
    queryKey: ['projects'],
    queryFn: async ({ pageParam }) => {
      const response = await fetch(`/api/projects?cursor=${pageParam}`)
      return await response.json()
    },
    initialPageParam: 0,
    getPreviousPageParam: (firstPage) => firstPage.previousId ?? undefined,
    getNextPageParam: (lastPage) => lastPage.nextId ?? undefined,
    maxPages: 3,
  })

  return (
```

```
<div>
  <h1>Infinite Query with max pages</h1>
  <h3>4 projects per page</h3>
  <h3>3 pages max</h3>
  {status === 'pending' ? (
    <p>Loading...</p>
  ) : status === 'error' ? (
    <span>Error: {error.message}</span>
  ) : (
    <>
      <div>
        <button
          onClick={() => fetchPreviousPage()}
          disabled={!hasPreviousPage || isFetchingPreviousPage}
        >
          {isFetchingPreviousPage
            ? 'Loading more...'
            : hasPreviousPage
            ? 'Load Older'
            : 'Nothing more to load'}
        </button>
      </div>
      {data.pages.map((page) => (
        <React.Fragment key={page.nextId}>
          {page.data.map((project) => (
            <p
              key={project.id}
              style={{
                border: '1px solid gray',
                borderRadius: '5px',
                padding: '8px',
                fontSize: '14px',
                background: `hsla(${project.id * 30}, 60%, 80%, 1)`,
              }}
            >
              {project.name}
            </p>
          ))}
        </React.Fragment>
      ))}
      <div>
        <button
          onClick={() => fetchNextPage()}
          disabled={!hasNextPage || isFetchingNextPage}
        >
          {isFetchingNextPage
            ? 'Loading more...'
            : hasNextPage
            ? 'Load Newer'
            : 'Nothing more to load'}
        </button>
      </div>
      <div>
        {isFetching && !isFetchingNextPage ? 'Background Updating...' : null}
      </div>
```

```
        </>
      )}
      <hr />
      <ReactQueryDevtools initialIsOpen={true} />
    </div>
  )
}
```

React Example: Suspense

Github
StackBlitz
CodeSandbox

Code Explorer & Interactive Sandbox

Public **src**

components index.tsx queries.ts

.eslintrc

.gitignore

README.md

index.html

package.json

tsconfig.json

vite.config.ts

```
import 'font-awesome/css/font-awesome.min.css'
import React, { lazy } from 'react'
import ReactDOM from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
  QueryErrorResetBoundary,
  useQueryClient,
} from '@tanstack/react-query'
import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
import { ErrorBoundary } from 'react-error-boundary'

import { fetchProjects } from './queries'

import Button from './components/Button'

const Projects = lazy(() => import('./components/Projects'))
const Project = lazy(() => import('./components/Project'))

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retry: 0,
    },
  },
})

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
```

```
}

function Example() {
  const queryClient = useQueryClient()
  const [showProjects, setShowProjects] = React.useState(false)
  const [activeProject, setActiveProject] = React.useState<string | null>(null)

  return (
    <>
      <Button
        onClick={() => {
          setShowProjects((old) => {
            if (!old) {
              queryClient.prefetchQuery({
                queryKey: ['projects'],
                queryFn: fetchProjects,
              })
            }
            return !old
          })
        }}
      >
        {showProjects ? 'Hide Projects' : 'Show Projects'}
      </Button>

      <hr />

      <QueryErrorResetBoundary>
        {({ reset }) => (
          <ErrorBoundary
            fallbackRender={({ error, resetErrorBoundary }) => (
              <div>
                There was an error!{' '}
                <Button onClick={() => resetErrorBoundary()}>Try again</Button>
                <pre style={{ whiteSpace: 'normal' }}>{error.message}</pre>
              </div>
            )}
            onReset={reset}
          >
            {showProjects ? (
              <React.Suspense fallback={<h1>Loading projects...</h1>}>
                {activeProject ? (
                  <Project
                    activeProject={activeProject}
                    setActiveProject={setActiveProject}
                  />
                ) : (
                  <Projects setActiveProject={setActiveProject} />
                )}
              </React.Suspense>
            ) : null}
          </ErrorBoundary>
        )}
      </QueryErrorResetBoundary>
      <ReactQueryDevtools initialIsOpen />
```

```
    </>;
  )
}

const rootElement = document.getElementById('root') as HTMLElement
ReactDOM.createRoot(rootElement).render(<App />);
```

[.... Additional repeated code block omitted for brevity ....]

Additional Resources

- Infinite query with Max pages
- Default Query Function

Our Partners

Additional Links

- TanStack Form: Headless, performant, and type-safe form state management for TS/JS, React, Vue, Angular, Solid, Lit, and Svelte.
- TanStack Config: The build and publish utilities used by all of our projects. Use it if you dare!

persistQueryClient

This is a set of utilities for interacting with **persisters**, which save your queryClient for later use. Different **persisters** can store your client and cache across many storage layers.

Build Persisters

- createSyncStoragePersister
- createAsyncStoragePersister
- Create a custom persister

How It Works

IMPORTANT - for persist to work properly, you probably want to pass `QueryClient` a `gcTime` value to override the default during hydration (as shown above).

If `gcTime` is not set when creating the `QueryClient` instance, it defaults to `300000` (5 minutes) for hydration, and the cache will be discarded after 5 minutes of inactivity. This is default garbage collection behavior.

It should be set the same or higher than `maxAge` in `persistQueryClient` (e.g., if `maxAge` is 24 hours (default), then `gcTime` should be 24 hours or more). If lower, garbage collection discards cache earlier than expected.

You can pass `Infinity` to disable garbage collection entirely.

Due to JavaScript limitations, the maximum `gcTime` is about 24 days (more).

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      gcTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
});
```

Cache Busting

Sometimes you make changes that should immediately invalidate cached data. You can pass a `buster` string option. If found cache does not include this `buster`, it will be discarded.

Functions accepting this option:

```
persistQueryClient({ queryClient, persister, buster: buildHash });
persistQueryClientSave({ queryClient, persister, buster: buildHash });
persistQueryClientRestore({ queryClient, persister, buster: buildHash });
```

Removal

If data is:

- expired (`maxAge`)

- busted (`buster`)
- error (e.g., throws)
- empty (e.g., undefined)

Then, `removeClient()` is called, discarding the cache immediately.

API

persistQueryClientSave

- Stores dehydrated query/mutation data via the persister.
- Throttled actions to limit write frequency.

```
persistQueryClientSave({
  queryClient,
  persister,
  buster = '',
  dehydrateOptions = undefined,
});
```

persistQueryClientSubscribe

- Runs `persistQueryClientSave` whenever cache changes.
- Returns an `unsubscribe` function.
- To erase cache, pass a new `buster` to `persistQueryClientRestore`.

```
persistQueryClientSubscribe({
  queryClient,
  persister,
  buster = '',
  dehydrateOptions = undefined,
});
```

persistQueryClientRestore

- Attempts to hydrate a persisted dehydrated cache back into the query cache.
- Discards cache if older than `maxAge`.

```
persistQueryClientRestore({
  queryClient,
  persister,
  maxAge = 1000 * 60 * 60 * 24, // 24 hours
  buster = '',
  hydrateOptions = undefined,
});
```

persistQueryClient

- Restores cache immediately.
- Subscribes to cache changes.

```
persistQueryClient({
  queryClient,
  persister,
  maxAge = 1000 * 60 * 60 * 24,
```

```
  buster = '',
  hydrateOptions = undefined,
  dehydrateOptions = undefined,
});
```

Options

PersistQueryClientOptions

```
interface PersistQueryClientOptions {
  queryClient: QueryClient;
  persister: Persister;
  maxAge?: number;
  buster?: string;
  hydrateOptions?: HydrateOptions;
  dehydrateOptions?: DehydrateOptions;
}
```

Three interfaces are available:

- `PersistedQueryClientSaveOptions` (used for save and subscribe, no hydrateOptions)
- `PersistedQueryClientRestoreOptions` (used for restore, no dehydrateOptions)
- `PersistQueryClientOptions` (used for full persist/query actions)

Usage with React

`persistQueryClient` attempts to restore cache and auto-subscribes to sync with storage. Restoring is asynchronous, so rendering during restore may cause race conditions.

Example

```
import { PersistQueryClientProvider } from '@tanstack/react-query-persist-client';
import { createAsyncStoragePersister } from '@tanstack/query-async-storage-persister';

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      gcTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
});

const persister = createAsyncStoragePersister({
  storage: window.localStorage,
});

ReactDOM.createRoot(rootElement).render(
  <PersistQueryClientProvider
    client={queryClient}
    persistOptions={{ persister }}
  >
    <App />
  </PersistQueryClientProvider>
);
```

Props

`PersistQueryClientProvider` props:

- `persistOptions` : options for persist functions.
- `onSuccess` , `onError` : callbacks after restore.

Hooks

- `useIsRestoring` : indicates if restore is in progress.

Persisters Interface

```
export interface Persister {
  persistClient(persistClient: PersistedClient): Promisable<void>;
  restoreClient(): Promisable<PersistedClient | undefined>;
  removeClient(): Promisable<void>;
}
```

PersistedClient

```
export interface PersistedClient {
  timestamp: number;
  buster: string;
  clientState: DehydratedState;
}
```

Building A Persister

Example: IndexedDB persister using idb-keyval

```
import { get, set, del } from 'idb-keyval';
import { PersistedClient, Persister } from '@tanstack/react-query-persist-client';

export function createIDBPersister(idbValidKey: IDBValidKey = 'reactQuery') {
  return {
    persistClient: async (client: PersistedClient) => {
      await set(idbValidKey, client);
    },
    restoreClient: async () => {
      return await get<PersistedClient>(idbValidKey);
    },
    removeClient: async () => {
      await del(idbValidKey);
    },
  } satisfies Persister;
}
```

Edit on GitHub

createSyncStoragePersister | TanStack Query React Docs

On this page

Deprecated

This plugin is deprecated and will be removed in the next major version.
You can simply use `@tanstack/query-async-storage-persister` instead.

Installation

This utility comes as a separate package and is available under the `'@tanstack/query-sync-storage-persister'` import.

```
npm install @tanstack/query-sync-storage-persister @tanstack/react-query-persist-clien
```

or

```
pnpm add @tanstack/query-sync-storage-persister @tanstack/react-query-persist-client
```

or

```
yarn add @tanstack/query-sync-storage-persister @tanstack/react-query-persist-client
```

or

```
bun add @tanstack/query-sync-storage-persister @tanstack/react-query-persist-client
```

Usage

- Import the `createSyncStoragePersister` function
- Create a new `syncStoragePersister`
- Pass it to the `persistQueryClient` function

```
import { persistQueryClient } from '@tanstack/react-query-persist-client'
import { createSyncStoragePersister } from '@tanstack/query-sync-storage-persister'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
```

```
      gcTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})

const localStoragePersister = createSyncStoragePersister({
  storage: window.localStorage,
})
// const sessionStoragePersister = createSyncStoragePersister({ storage: window.sessio

persistQueryClient({
  queryClient,
  persister: localStoragePersister,
})
```

Retries

Persistence can fail, e.g. if the size exceeds available storage space.
Errors can be handled gracefully by providing a `retry` function to the persister.

The retry function receives:

- `persistedClient`: the client it tried to save
- `error`: the error that occurred
- `errorCount`: number of retry attempts

It should return a new `PersistedClient` to retry, or `undefined` to stop retrying.

```
export type PersistRetryer = (props: {
  persistedClient: PersistedClient
  error: Error
  errorCount: number
}) => PersistedClient | undefined
```

Predefined strategies

By default, no retry occurs.
You can import predefined strategies from `'@tanstack/react-query-persist-client'`:

- `removeOldestQuery`: returns a `PersistedClient` with the oldest query removed.

```
const localStoragePersister = createSyncStoragePersister({
  storage: window.localStorage,
  retry: removeOldestQuery,
})
```

API

createSyncStoragePersister

Creates a syncStoragePersister to be used with `persistQueryClient`.

```
createSyncStoragePersister(options: CreateSyncStoragePersisterOptions)
```

Options

```
interface CreateSyncStoragePersisterOptions {
  /** The storage client used for setting and retrieving items (window.localStorage or
  storage: Storage | undefined | null
  /** The key to use when storing the cache */
  key?: string
  /** Throttle saving to disk to avoid spamming (in ms) */
  throttleTime?: number
  /** Serialization method for data to storage */
  serialize?: (client: PersistedClient) => string
  /** Deserialization method for data from storage */
  deserialize?: (cachedString: string) => PersistedClient
  /** Retry persistence on error */
  retry?: PersistRetryer
}
```

Default options:

```
{
  key = `REACT_QUERY_OFFLINE_CACHE`,
  throttleTime = 1000,
  serialize = JSON.stringify,
  deserialize = JSON.parse,
}
```

serialize and deserialize options

To handle large data, override `serialize` and `deserialize` using compression libraries like lz-string:

```
import { QueryClient } from '@tanstack/react-query'
import { persistQueryClient } from '@tanstack/react-query-persist-client'
import { createSyncStoragePersister } from '@tanstack/query-sync-storage-persister'

import { compress, decompress } from 'lz-string'

const queryClient = new QueryClient({
  defaultOptions: { queries: { staleTime: Infinity } },
})

persistQueryClient({
  queryClient,
  persister: createSyncStoragePersister({
    storage: window.localStorage,
    serialize: (data) => compress(JSON.stringify(data)),
    deserialize: (data) => JSON.parse(decompress(data)),
  }),
  maxAge: Infinity,
})
```

Editable Link

Edit on GitHub

createAsyncStoragePersister | TanStack Query React Docs

On this page

createAsyncStoragePersister

Installation

This utility comes as a separate package and is available under the `'@tanstack/query-async-storage-persister'` import.

```
npm install @tanstack/query-async-storage-persister @tanstack/react-query-persist-clie
```

```
pnpm add @tanstack/query-async-storage-persister @tanstack/react-query-persist-client
```

```
yarn add @tanstack/query-async-storage-persister @tanstack/react-query-persist-client
```

```
bun add @tanstack/query-async-storage-persister @tanstack/react-query-persist-client
```

Usage

- Import the `createAsyncStoragePersister` function
- Create a new asyncStoragePersister
  - You can pass any `storage` to it that adheres to the `AsyncStorage` interface – the example below uses the async-storage from React Native.
  - Storages that read and write synchronously, like `window.localStorage`, also adhere to the `AsyncStorage` interface and can be used with `createAsyncStoragePersister`.
- Wrap your app using the PersistQueryClientProvider

```
import AsyncStorage from '@react-native-async-storage/async-storage'
import { QueryClient } from '@tanstack/react-query'
import { PersistQueryClientProvider } from '@tanstack/react-query-persist-client'
import { createAsyncStoragePersister } from '@tanstack/query-async-storage-persister'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      gcTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})
```

```
const asyncStoragePersister = createAsyncStoragePersister({
  storage: AsyncStorage,
})

const Root = () => (
  <PersistQueryClientProvider
    client={queryClient}
    persistOptions={{ persister: asyncStoragePersister }}
  >
    <App />
  </PersistQueryClientProvider>
)

export default Root
```

Retries

Retries work the same as for a SyncStoragePersister, except they can also be asynchronous. All predefined retry handlers can be used.

API

createAsyncStoragePersister

Call this function to create an asyncStoragePersister for later use with `persistQueryClient`.

```
createAsyncStoragePersister(options: CreateAsyncStoragePersisterOptions)
```

Options

```
interface CreateAsyncStoragePersisterOptions {
  /** The storage client used for setting and retrieving items from cache */
  storage: AsyncStorage | undefined | null
  /** The key to use when storing the cache to localStorage */
  key?: string
  /** To avoid localStorage spamming, pass a time in ms to throttle saving the cache t
  throttleTime?: number
  /** How to serialize the data to storage */
  serialize?: (client: PersistedClient) => string
  /** How to deserialize the data from storage */
  deserialize?: (cachedString: string) => PersistedClient
  /** How to retry persistence on error */
  retry?: AsyncPersistRetryer
}

interface AsyncStorage<TStorageValue = string> {
  getItem: (key: string) => MaybePromise<TStorageValue | undefined | null>
  setItem: (key: string, value: TStorageValue) => MaybePromise<unknown>
  removeItem: (key: string) => MaybePromise<void>
  entries?: () => MaybePromise<Array<[key: string, value: TStorageValue]>>
}
```

Default Options

```
{
  key = `REACT_QUERY_OFFLINE_CACHE`,
  throttleTime = 1000,
  serialize = JSON.stringify,
  deserialize = JSON.parse,
}
```

Additional Resources

- Edit on GitHub

Related

- createSyncStoragePersister
- broadcastQueryClient (Experimental)

broadcastQueryClient (Experimental) | TanStack Query React Docs

---

On this page

---

broadcastQueryClient (Experimental)

> **VERY IMPORTANT:** This utility is currently in an experimental stage. This means that breaking changes will happen in minor **and** patch releases. Use at your own risk. If you choose to rely on this in production in an experimental stage, please lock your version to a patch-level version to avoid unexpected breakages.

broadcastQueryClient is a utility for broadcasting and syncing the state of your `queryClient` between browser tabs/windows with the same origin.

Installation

This utility comes as a separate package and is available under the `'@tanstack/query-broadcast-client-experimental'` import.

Usage

Import the `broadcastQueryClient` function, and pass it your `QueryClient` instance, and optionally, set a `broadcastChannel`.

```
import { broadcastQueryClient } from '@tanstack/query-broadcast-client-experimental'

const queryClient = new QueryClient()

broadcastQueryClient({
  queryClient,
  broadcastChannel: 'my-app',
})
```

API

broadcastQueryClient

Pass this function a `QueryClient` instance and optionally, a `broadcastChannel`.

```
broadcastQueryClient({ queryClient, broadcastChannel })
```

Options

An object of options:

```
interface BroadcastQueryClientOptions {
  /** The QueryClient to sync */
  queryClient: QueryClient
  /** This is the unique channel name that will be used
   * to communicate between tabs and windows */
  broadcastChannel?: string
  /** Options for the BroadcastChannel API */
  options?: BroadcastChannelOptions
}
```

The default options are:

```
{
  broadcastChannel = 'tanstack-query',
}
```

Further Resources

Edit on GitHub

On this page — Quick Links

experimental_createQueryPersister

Installation

This utility comes as a separate package and is available under the `'@tanstack/query-persist-client-core'` import.

```
npm install @tanstack/query-persist-client-core
```

or

```
pnpm add @tanstack/query-persist-client-core
```

or

```
yarn add @tanstack/query-persist-client-core
```

or

```
bun add @tanstack/query-persist-client-core
```

> **Note:** This util is also included in the `@tanstack/react-query-persist-client` package, so you do not need to install it separately if using that package.

Usage

- Import the `experimental_createQueryPersister` function
- Create a new `experimental_createQueryPersister`:
  - You can pass any `storage` that adheres to the `AsyncStorage` interface — e.g., the async-storage from React Native.

```
import AsyncStorage from '@react-native-async-storage/async-storage'
import { QueryClient } from '@tanstack/react-query'
import { experimental_createQueryPersister } from '@tanstack/query-persist-client-core

const persister = experimental_createQueryPersister({
  storage: AsyncStorage,
  maxAge: 1000 * 60 * 60 * 12, // 12 hours
})

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      gcTime: 1000 * 30, // 30 seconds
      persister: persister.persisterFn,
    },
  },
})
```

- Alternatively, the same setup can be duplicated as shown above.

  Note:
  `queryClient.setQueryData()` is not persisted automatically. For optimistic updates, you need to persist manually, e.g., using `persistQueryByKey`.
  Each query is lazily restored and persisted after each run of `queryFn`, respecting `staleTime`.

Adapted defaults

The `createPersister` plugin wraps the `queryFn`, acting as a caching layer.
`networkMode` defaults to `'offlineFirst'` when a persister is used, enabling restoration even without network.

Additional utilities

Invoking `experimental_createQueryPersister` provides additional helpers:

persistQueryByKey(queryKey: QueryKey, queryClient: QueryClient): Promise

Persists a specific query to storage, useful for optimistic updates.

```
const persister = experimental_createQueryPersister({
  storage: AsyncStorage,
  maxAge: 1000 * 60 * 60 * 12,
})

const queryClient = useQueryClient()

useMutation({
  mutationFn: updateTodo,
  onMutate: async (newTodo) => {
    // Optimistically update
    queryClient.setQueryData(['todos'], (old) => [...old, newTodo])
    // Persist to storage
    await persister.persistQueryByKey(['todos'], queryClient)
  },
})
```

retrieveQuery(queryHash: string): Promise<T | undefined>

Attempt to retrieve a persisted query by its hash.
Returns `undefined` if expired, invalid, or malformed (and removes it).

```
const data = await persister.retrieveQuery('some-query-hash')
```

persisterGc(): Promise

Cleans up storage from expired, invalid, or malformed entries.

```
await persister.persisterGc()
```

> **Requirement:** Storage must expose `entries()` method returning a key-value tuple array, e.g., `Object.entries(localStorage)`.

restoreQueries(queryClient: QueryClient, filters): Promise

Restores specific queries based on filters, useful on startup or offline mode.

```
await persister.restoreQueries(queryClient, {
  queryKey: ['todos'],
```

```
  exact: true,
})
```

API

experimental_createQueryPersister(options: StoragePersisterOptions)

Creates a persister with specified options.

Options

```
export interface StoragePersisterOptions {
  storage: AsyncStorage | Storage | undefined | null
  serialize?: (persistedQuery: PersistedQuery) => string
  deserialize?: (cachedString: string) => PersistedQuery
  buster?: string
  maxAge?: number
  prefix?: string
  filters?: QueryFilters
}
```

AsyncStorage interface

```
interface AsyncStorage<TStorageValue = string> {
  getItem: (key: string) => MaybePromise<TStorageValue | undefined | null>
  setItem: (key: string, value: TStorageValue) => MaybePromise<unknown>
  removeItem: (key: string) => MaybePromise<void>
  entries?: () => MaybePromise<Array<[key: string, value: TStorageValue]>>
}
```

Default options

```
{
  prefix = 'tanstack-query',
  maxAge = 1000 * 60 * 60 * 24,
  serialize = JSON.stringify,
  deserialize = JSON.parse,
}
```

Link: Edit on GitHub

On this page