

Data Fetching with TanStack Query

*Caching, Synchronization, and Server State for Modern
Web Apps*

Tobias Renholt

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press



© 2026 by NOBTREX LLC. All rights reserved.

For permissions and other inquiries, write to:
P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language models have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

Contents

1	Introduction	1
2	Framing Server State in Modern Frontend Architecture	5
2.1	Why Server State Requires a Different Mental Model	6
2.2	From React Query to TanStack Query	15
2.3	Declarative Data Dependencies Instead of Effect Chains	23
2.4	The Operational Responsibilities TanStack Query Owns	34
3	Bootstrapping the Query Runtime	45
3.1	Designing the QueryClient as Shared Infrastructure	45
3.2	Composing QueryClientProvider Boundaries for Real Applications	55
3.3	Encoding Global Defaults as Runtime Policy	64
3.4	Operating the Runtime with Devtools and Imperative Client APIs	73

4	Queries, Keys, and Cache Identity	83
4.1	Anatomy of a Declarative Query	83
4.2	Modeling Dependent Reads with Enabled Queries . . .	94
4.3	Designing Query Keys for Correctness and Scale	105
4.4	Operating Cache Reuse and Refetch Boundaries Through Key Design	114
5	Freshness, Refetching, and the Cache Lifecycle	125
5.1	Fresh, Stale, Active, Inactive, and Collected States . . .	125
5.2	Using staleTime to Control Synchronization Pressure .	133
5.3	Understanding gcTime and Inactive Cache Retention .	144
5.4	Configuring Refetch on Mount, Focus, Reconnect, and Interval	154
5.5	Hydration, Activity, and Lifecycle Edge Cases	165
6	Mutations, Invalidation, and Consistency After Writes	175
6.1	Modeling Server Writes with useMutation	175
6.2	Using Targeted Invalidation as the Default Consistency Strategy	186
6.3	Repairing Cached Read Models with Direct Cache Updates	195
6.4	Implementing Optimistic Updates and Rollback Policies	204
6.5	Operating Mutations Under Concurrency, Failure, and User Feedback Constraints	214

7	Scaling Read Patterns: Pagination, Infinite Loading, and Prefetching	225
7.1	Paginated Queries as Parameterized Cache Entries . . .	226
7.2	Implementing Infinite Queries with Page-Param Contracts	236
7.3	Bounding Infinite List Retention with maxPages and Cache Lifecycle Policy	247
7.4	Prefetching for Latency Hiding and Navigation Smoothness	257
8	Server Rendering, Hydration, Offline Resilience, and Production Patterns	269
8.1	Implementing Server Prefetch, Dehydration, and Client Hydration	270
8.2	Choosing Freshness Windows and Isolation Boundaries for SSR	281
8.3	Persisting Cache State Across Sessions	288
8.4	Operating Queries and Mutations Under Offline and Flaky Network Conditions	298
8.5	Applying Production Heuristics to Real TanStack Query Architectures	307

Chapter 1

Introduction

Modern web applications depend on data that is requested asynchronously, shared across screens, revisited over time, and updated under changing network conditions. This data is not owned by the frontend. It is retrieved from external systems, can become stale without local interaction, may be refetched automatically, and often needs to remain coherent across multiple components at once. These characteristics make server state fundamentally different from local UI state. Treating both categories with the same tools and assumptions often leads to unnecessary complexity, duplicated logic, and fragile synchronization behavior.

TanStack Query addresses this problem by providing a focused system for managing server state in client applications. Rather than acting as a general-purpose state container, it defines a clear operational model for fetching, caching, synchronizing, and updating asynchronous data. Its value is not limited to reducing boilerplate around HTTP requests. It also formalizes concerns that become increasingly important as applications grow: cache identity, data freshness, background refetching,

retry behavior, mutation coordination, and consistency after writes.

This book presents TanStack Query as infrastructure for modern front-end architecture. It adopts the current v5-era API and terminology, and explains the library through the mental model that best supports long-term maintainability: declarative data dependencies. Instead of building imperative chains of effects that manually trigger requests, manage loading flags, and coordinate retries, readers will learn to describe resources in terms of stable query keys, query functions, execution conditions, and cache policy. This shift improves correctness because it aligns application behavior with the identity and lifecycle of the underlying data.

The chapters that follow move from foundations to advanced production patterns. The opening chapters define server state precisely, explain the evolution from React Query to TanStack Query, and establish the responsibilities the library assumes on behalf of the application. From there, the book builds the runtime layer around `QueryClient`, provider placement, and global defaults, showing how application-wide policy is encoded. It then turns to the core read path: queries, query keys, cache identity, and the rules that govern reuse and segmentation.

Once those fundamentals are in place, the book examines freshness and cache lifecycle in detail. Concepts such as `staleTime`, `gcTime`, active observers, and refetch triggers are essential to understanding why cached data behaves the way it does under mounting, focus changes, reconnects, and navigation. The write path follows, with careful treatment of mutations, invalidation, direct cache updates, optimistic workflows, and concurrency concerns. Later chapters extend the model to pagination, infinite queries, and prefetching, where performance and memory discipline become especially important. The final chapter addresses server rendering, hydration, persistence, offline behavior, and broader architectural heuristics for production systems.

Throughout the book, the emphasis is on practical correctness over incidental cleverness. TanStack Query is most effective when its abstractions are used with discipline: query keys must be designed deliberately, defaults must reflect real synchronization needs, and write handling should favor simple, robust consistency mechanisms whenever possible. The goal of this book is to help readers build that discipline, understand the tradeoffs behind the library’s design, and apply TanStack Query confidently in applications that need reliable server-state management at scale.

Chapter 2

Framing Server State in Modern Frontend Architecture

Modern frontend bugs around data are rarely caused by rendering alone; they come from stale reads, duplicated fetches, race conditions, inconsistent retries, and the hidden cost of pretending remote data behaves like local memory. This chapter gives readers the vocabulary and mental model to see TanStack Query not as a convenience hook library, but as an architectural response to the operational pressures of server state in real applications.

2.1. Why Server State Requires a Different Mental Model

A frontend often fails long before it looks obviously broken. The first version works: fetch some JSON, put it in component state, show a spinner, render the result. Then the screen gains navigation, filters, background updates, retries, optimistic actions, offline gaps, and multiple views of the same record. What looked like ordinary state turns into a moving boundary between your UI and a system you do not control. The code starts to accumulate flags such as `isLoading`, `hasLoaded`, `isRefreshing`, `didCancel`, and `needsReload`. That sprawl is usually not a sign that your team forgot a pattern. It is a sign that remote data obeys a different set of rules than local state.

A small effect-driven fetch makes the problem concrete:

```
import { useEffect, useState } from 'react'

export function UserList() {
  const [users, setUsers] = useState([])
  const [loading, setLoading] = useState(true)
  const [error, setError] = useState(null)

  useEffect(() => {
    let ignore = false

    async function load() {
      setLoading(true)
      setError(null)

      try {
        const res = await fetch('/api/users')
        if (!res.ok) {
          throw new Error('request failed')
        }
        const data = await res.json()
        if (!ignore) {
          setUsers(data)
        }
      } catch (err) {
        if (!ignore) {
          setError(err)
        }
      }
    }
  })
}
```

```
        }
    } finally {
        if (!ignore) {
            setLoading(false)
        }
    }
}

load()

return () => {
    ignore = true
}
}, [])

if (loading) return <p>Loading...</p>
if (error) return <p>Error</p>

return (
    <ul>
        {users.map(user => (
            <li key={user.id}>{user.name}</li>
        ))}
    </ul>
)
}
```

That code is plausible, common, and insufficient. It handles one mount-time request and one cleanup path. It does not answer whether data is still fresh after five seconds, whether another screen can reuse it, whether multiple components trigger duplicate requests, whether a reconnect should retry, whether tab focus should revalidate, whether a detail screen and a list screen should agree after an edit, or whether a slower request should overwrite newer data. Once those questions matter, you are no longer managing *loading state*. You are managing server state.

Ownership and authority

The most useful distinction is not where the data came from or whether the payload is JSON. The distinction is who owns the truth. Local UI state is usually owned by the running client process. If you store

whether a modal is open, which tab is selected, or the current value of an uncontrolled draft field, your application is authoritative for that data. Reads are synchronous, writes are immediate, and the value changes only because your code changed it.

Server state is observed, not owned. A user profile loaded from an API may be displayed in your application, cached in memory, and even edited through your interface, but the canonical value lives elsewhere. Another browser tab, another user, an administrator tool, a background job, or the server itself may change it without your application doing anything. Your UI only holds a copy acquired at some time under some network conditions. That copy can be useful, but it is not authoritative.

That single difference changes the entire programming model. When you own local state, mutation is enough. When you observe server state, you also need policies for acquisition, reuse, freshness, synchronization, and conflict handling. Treating both categories as interchangeable collapses important operational distinctions. A boolean in local memory does not become stale because a laptop went offline for thirty seconds. A server response can.

Freshness is a first-class property

Local state is generally either current or intentionally outdated because you have not updated it yet. Server state is different: even a successful read only tells you what the server returned at a specific moment. The practical question is not merely “do I have data,” but “how old is this data, and under what conditions is it safe to reuse?”

This is why server-state systems care about freshness windows rather than just values. A cached list of invoices from two seconds ago might be perfectly acceptable for navigation responsiveness. The same list could be unacceptable after a reconnect, after a mutation, or when the user opens a screen that depends on exact balances. Freshness is con-

textual and temporal.

HTTP itself reflects this reality. Correct caching is not defined by URL string matching alone; it depends on request and response metadata, validation rules, and header-sensitive semantics such as `Vary`. Frontend server-state tooling does not invent complexity here. It exposes complexity that already exists in remote data access. If your application ignores freshness entirely, it does not become simpler in production. It simply pushes stale-data bugs into places where they are harder to detect.

Synchronization pressure comes from outside the component tree

A React component rerenders because props or local state changed. Server state changes according to external events that may have nothing to do with your component lifecycle. A record can be updated by another actor. A network can fail transiently and recover. A tab can be backgrounded and later focused. A mobile device can disconnect and reconnect. Authentication can change. A mutation can make previously fetched data incomplete or invalid.

These are synchronization pressures. They force your UI to reconcile its local snapshot with a source of truth that evolves independently. If you model remote data as a plain state variable, your code has no native place to express this pressure. You start encoding it indirectly with effect dependencies, timers, focus listeners, route listeners, retry loops, and hand-rolled cache maps. The implementation becomes brittle because each concern is solved locally while the real problem is global.

A production UI usually needs at least some of the following behaviors:

- reuse data across screens without immediately discarding it,
- distinguish between “no data yet” and “data shown while refreshing”,

- avoid duplicate requests for the same resource,
- retry transient failures without retrying permanent ones forever,
- refetch when the application regains a better chance of success,
- prevent older responses from overwriting newer intent,
- keep overlapping views reasonably consistent after writes.

Those are not optional enhancements layered on top of fetching. They are part of the operational contract of presenting remote data in an interactive application.

Failure handling is structural, not exceptional

With local state, failure is usually a logic bug in your code. With server state, failure is a routine operating condition. DNS can fail. TCP connections can stall. A request can time out. A server can respond with an error because of temporary load. A browser can abort a fetch because the user navigated away. A response can succeed but arrive too late to match the current UI intent.

This matters because error handling cannot sit at the edge as a rare exception path. It must be part of the state model. A screen often needs to distinguish among at least four materially different situations:

- no request has run yet,
- a request is in flight and there is no cached data,
- a request failed and there is no usable data,
- a request failed but a previous result is still available.

Those states are operationally distinct. They produce different UI behavior, recovery options, and user expectations. A simple

loading/error/data triad often collapses them in ways that feel acceptable on a toy screen and confusing in a real application.

Cacheability and reuse change the economics of reading

When you update local state, the write directly changes what the application will read next. Remote reads are more expensive. They involve latency, network variability, server load, authentication context, and serialization overhead. That cost makes reuse valuable, but reuse is never free because it introduces the freshness question.

You therefore need a vocabulary that separates *having data in memory* from *knowing whether it should be reused*. Server-state management lives in that space. A list of products fetched on one route might be good enough to show immediately on a second route while a background refresh runs. Without cache reuse, navigation feels slower than it needs to. Without freshness policy, reuse quietly serves outdated information. Good server-state architecture balances both.

This also explains why duplicated fetches are not merely a performance smell. They can create inconsistency. If two components independently fetch the same resource, they may load at different times, fail differently, and present contradictory snapshots. Once you have multiple readers of the same remote entity, identity and sharing become architectural concerns.

Overlapping read models multiply consistency problems

The same server data rarely appears in only one shape. A user may exist in a paginated list, a detail page, an assignee picker, and a dashboard summary card. Each view has a different projection, loading cadence, and interaction pattern. Yet users expect them to agree closely enough that the application feels coherent.

This is where teams often reach for the wrong abstraction. They mirror server responses into a general-purpose client store and treat those

objects as if they were local canonical state. That can work for a while, but it pushes remote-data semantics into a state model designed for synchronous ownership. You then need custom code for expiration, refetch triggers, invalidation, de-duplication, mutation aftermath, and race handling anyway. The store becomes a cache without admitting that it is a cache.

The better mental model starts by naming the problem correctly. A list and a detail screen are not two local states that happen to contain similar objects. They are two observations of server-owned data, each with its own reuse and synchronization policy. Once you accept that, concepts like cache identity, invalidation, and background refresh stop looking like library-specific cleverness. They become direct responses to the shape of the problem.

Why effect-driven fetching scales poorly

React's `useEffect` is for synchronizing a component with an external system. Fetching data is one such synchronization, so using an effect is not wrong. The trouble begins when effects are treated as the architecture rather than the low-level mechanism. Manual fetching in effects requires you to wire every operational concern yourself: start conditions, cancellation cleanup, race avoidance, loading transitions, retries, reuse, and preloading. It also tends to produce network waterfalls when child components wait for parents to render before they start their own requests.

Even a more careful fetch with cancellation support shows the boundary of the pattern:

```
useEffect(() => {
  const controller = new AbortController()

  async function load() {
    const res = await fetch('/api/users', {
      signal: controller.signal,
    })
    const data = await res.json()
  }
```

```
    setUsers(data)
  }

  load().catch(err => {
    if (err.name !== 'AbortError') {
      setError(err)
    }
  })

  return () => {
    controller.abort()
  }
}, [])
```

This improves correctness, but the burden remains on you. You still need to answer what happens on remount, focus, reconnect, parameter changes, duplicated subscribers, stale cached values, and post-mutation consistency. Each answer is another local rule. Over time the codebase accumulates many such rules, often expressed differently in different features. The result is not just more code. It is divergent behavior that users experience as randomness.

A practical taxonomy

A precise vocabulary helps you decide which tool belongs where.

Local UI state is client-owned, synchronous, and authoritative within the running app. Examples include dialog visibility, selected row IDs, sort direction, unsaved form draft text, and temporary drag state.

Server state is server-owned, asynchronously retrieved, potentially stale, shareable across views, and subject to refetch or invalidation. Examples include account balances, project lists, notifications, pricing data, and audit records.

Derived view state sits on top of either category. A filtered array computed from fetched products is not itself server state; it is a local derivation of a server snapshot. That distinction matters because you

usually should not persist the derived value as if it were canonical.

Form draft state is a frequent source of confusion. The submitted entity may be server state, but the in-progress form values are local state until you commit them. Copying a server record into a form is reasonable. Treating the draft as the new source of truth for the rest of the application usually is not.

When simpler techniques are still enough

Not every remote read justifies dedicated server-state infrastructure. If you have a one-shot request on a screen with no reuse, no background synchronization needs, no mutation coupling, and no meaningful stale window, a direct fetch can be sufficient. A small marketing page that loads a single feature flag or a short-lived admin utility may not need a cache-aware runtime.

The decision signals are operational. Simpler techniques remain adequate when most of the following are true:

- the data is read once and discarded,
- no other screen needs the same result,
- staleness has little user impact,
- retries and reconnection behavior do not matter,
- you do not need to keep views consistent after writes.

Dedicated server-state management becomes a strong fit when the opposite conditions appear: repeated reads, cross-route reuse, user-visible latency, multi-view consistency, background refresh expectations, or a meaningful distinction between cached data and current truth. At that point the complexity already exists in the product. A library does not create it; the library centralizes it.

Common category errors

A few mistakes recur because they blur ownership boundaries.

Mirroring server data into local stores by default. This creates a second authority without solving freshness. You pay synchronization cost twice.

Conflating form state with canonical data. Draft edits are local until persisted. Treat them as such.

Assuming a successful fetch implies correctness. Success only means you obtained a response, not that the value is still current when rendered.

Treating every refetch as a bug. For server state, revalidation is often evidence that the system is honoring freshness constraints rather than ignoring them.

Reducing remote reads to booleans. Real applications need richer status than “loaded or not” because cached data, background activity, and failure recovery overlap.

Once you classify remote data by ownership, freshness, synchronization pressure, and failure modes, the architecture becomes easier to reason about. Server state stops looking like awkward local state and starts looking like a managed observation of external truth, with caching and revalidation as core behaviors rather than afterthoughts.

2.2. From React Query to TanStack Query

Once you start treating remote data as server state rather than as ad hoc component state, documentation quality becomes a practical dependency. The difficulty is that the surrounding literature spans multiple names, multiple major versions, and multiple assumptions about

how the library is meant to be used. You will routinely encounter articles titled “React Query,” code importing from a TanStack package, migration notes discussing renamed APIs, and examples whose underlying mental model no longer matches current guidance. If you do not stabilize the vocabulary first, you can copy correct-looking code that encodes the wrong era.

A short compatibility snapshot helps anchor the discussion:

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

// v5 terminology notes:
// - React integration package: '@tanstack/react-query'
// - React 18.0 or later is required
// - Hydrate was renamed to HydrationBoundary
// - loading status terminology moved to pending
// - QueryClientProvider contextSharing was removed
```

That snippet is intentionally small. It is not meant to teach hydration yet. It establishes a rule that matters everywhere else: when you read modern React examples for this library, the authoritative package name is '@tanstack/react-query', and the naming around some core concepts changed in v5. If you search casually and paste the first example you find, you may mix versions before you ever write your first query.

Why the name changed

The library is officially described as being formerly known as React Query. That phrase matters because it explains why older material is still common and often still conceptually valuable. The project began in the React ecosystem, and much of the community vocabulary was built under that name. The rename to TanStack Query reflects expansion into the broader TanStack family and a multi-framework orientation rather than a cosmetic rebrand.

You should read that change as a scope signal. “React Query” suggested a React-specific helper for fetching. “TanStack Query” places the li-

library in a larger family of infrastructure tools and emphasizes that the core problem is not unique to React. Server state has to be fetched, cached, synchronized, invalidated, and surfaced coherently in any serious client application. React remains a major integration point, but the conceptual center of gravity moved from a React-centric label to a broader runtime and architecture story.

That distinction helps prevent a subtle misunderstanding. The rename does not mean the library abandoned React or rewrote its core purpose. The central idea remains stable: manage server-state reads and writes with cache-aware, declarative coordination rather than bespoke effect choreography. What changed is that the naming now better reflects the library's place in a wider ecosystem and the fact that the problem space extends beyond one view library.

Why version context matters more than branding

Name confusion is annoying; version confusion is dangerous. A code sample can look almost identical across versions while embedding a different mental model. That matters because people often treat frontend libraries as if only syntax changes between releases. With server-state tooling, terminology often encodes behavior. If a status name changes, or a boundary component is renamed, or a provider option disappears, the library is usually clarifying how it wants you to think, not just how it wants you to type.

Three verified v5 boundaries are especially important for orienting your reading:

- React integrations use the package `'@tanstack/react-query'`.
- v5 requires React 18.0 or later.
- Several identifiers changed, including `Hydrate` to `HydrationBoundary`, a shift from loading terminology

to pending, and the removal of `QueryClientProvider` `contextSharing`.

Each of those has practical consequences. The import path tells you whether an example belongs to the current naming regime. The React 18 requirement tells you whether an old application baseline can adopt v5 directly. The renamed identifiers tell you whether a blog post, conference slide, or code review comment is speaking the same language as your codebase.

Stable concepts versus unstable call surfaces

One of the easiest ways to misread older material is to discard it entirely once the name changes. That is unnecessary. A large portion of the conceptual model survives version changes intact. Query identity still matters. Cache-backed reads still matter. Fresh versus stale data still matters. The distinction between server state and local state still matters. If an older article explains why query keys are central, or why remote data cannot be treated as ordinary client memory, it can still be useful.

The unstable part is the call surface: exact identifiers, status naming, migration-sensitive components, and provider behavior. That is where you need discipline. Treat conceptual writing as portable until proven otherwise. Treat API examples as version-bound unless verified.

A practical reading rule helps:

1. Trust older material for architecture if it discusses query identity, freshness, caching, or server-state boundaries in a way that does not depend on exact names.
2. Verify every import path, component name, and option name before you copy code.

3. If an article mixes old branding with current package names, slow down and check its publication date and target version.

That rule is especially important when a team inherits a codebase built across several years. The danger is not just broken compilation. The deeper risk is conceptual drift, where one part of the team speaks in v5 terms while another part keeps applying older assumptions.

Interpreting the v5 posture

The v5 release matters because it sharpens the modern posture of the library. Even without cataloging every migration detail, you should recognize two broad signals.

First, the library is presented through a more consistent modern API style. This supports clearer declarative usage and reduces ambiguity across hooks and integrations. Later material can therefore treat object-shaped usage and modern naming as the baseline rather than as a special case.

Second, several terminology updates better align the surface API with runtime behavior. Renaming `Hydrate` to `HydrationBoundary` is a good example. The new name frames the construct as a boundary in the render tree where hydrated cache state becomes available, which is a more precise mental model than a verb-like name that sounds like a one-off action. Likewise, moving status terminology from `loading` toward `pending` clarifies that asynchronous work and rendered data availability are not always the same thing. In server-state systems, you can have cached data while a request is still in progress. A term like “pending” can better express a request lifecycle without implying that the UI necessarily has nothing useful to show.

You do not need every detail of those changes yet. What matters is the direction: the modern API surface is designed to make server-state semantics more explicit. When the names change, treat that as guidance

about the model, not just the syntax.

A concise migration map

When you audit examples or legacy code, keep a compact translation list nearby:

Older identifier or assumption	v5 interpretation
React Query	Historical name; current React package is '@tanstack/react-query'
Hydrate	Renamed to HydrationBoundary
loading status wording	Use v5 pending terminology where applicable
contextSharing on QueryClientProvider	Removed; share a QueryClient explicitly when needed
Pre-React 18 assumptions	v5 requires React 18.0 or later

This table is deliberately narrow. It is not a migration checklist. It is a compatibility filter you can apply while reading code and documentation. If an example uses one of the left-hand entries as if it were current, you should assume the surrounding API details may also be dated.

Why contextSharing removal matters conceptually

One change deserves more than a simple rename note: the removal of QueryClientProvider contextSharing. This is easy to misread as a small provider option cleanup. It is more significant because it changes how you reason about shared cache behavior, especially in complex applications such as microfrontend deployments.

Older patterns could encourage the idea that shared client behavior might emerge from provider-level magic. The removal pushes you toward explicit ownership: if multiple parts of an application should observe the same query cache, pass or otherwise coordinate a shared QueryClient instance intentionally. That makes cache topology a design choice rather than an accidental side effect of context configuration.

You should treat that as an architectural improvement. Shared caches

affect consistency, isolation, memory use, invalidation reach, and failure domains. Those are too important to hide behind an implicit setting. In a microfrontend environment, for example, a shared `QueryClient` can unify cache state across fragments, but it also couples them operationally. An isolated client can protect boundaries, but it sacrifices reuse. The removal of `contextSharing` forces that trade-off into the open.

Because the full constructor and setup details are covered elsewhere, it is enough here to hold onto the governing rule: shared query behavior in modern TanStack Query should be explicit, not incidental.

Reading legacy code without overreacting

Many teams inherit applications that use a mix of naming generations. You might see comments referring to React Query, imports already updated to `'@tanstack/react-query'`, and internal abstractions named after concepts from an earlier release. Do not treat that mixture as immediate evidence of architectural failure. The key question is whether the code relies on obsolete semantics or merely stale vocabulary.

A practical audit sequence works well:

1. Check the package imports first. If the code imports from `'@tanstack/react-query'`, you are at least in the current naming family for React integrations.
2. Search for known renamed identifiers such as `Hydrate`. Those are reliable markers of older API posture.
3. Search for provider usage that assumes `contextSharing`. If present, plan for explicit client sharing instead.
4. Check the application's React version. If it is below 18.0, v5 is not a drop-in target.

5. Separate conceptual abstractions from compatibility issues. Names inside your own codebase can be changed gradually if the underlying runtime behavior is already correct.

This audit order saves time because it identifies hard boundaries early. There is little value debating status terminology if the application is not yet on a compatible React baseline.

Why older examples can still mislead even when they compile

Version mismatch is not only a compilation problem. Sometimes old examples still compile after minor edits, which makes them more dangerous. You can update an import path, fix one renamed component, and still carry forward an outdated style of reasoning. That often happens when teams cargo-cult snippets without reading the migration intent behind them.

Suppose you take an older article that treats hydration as a simple one-time restore step, rename `Hydrate` to `HydrationBoundary`, and stop there. The code may work, but the name change was signaling a more precise boundary-oriented model. If you miss that, you are more likely to place it incorrectly in the tree or misunderstand which observers see hydrated state. The same pattern appears with status naming: changing strings mechanically is easy; adjusting your thinking about request lifecycle versus UI availability is harder and more important.

This is why version-aware reading is part of architecture, not mere housekeeping. API names are compressed documentation.

A working rule for the rest of the material

Use TanStack Query v5 as the normative baseline for React work. Interpret “React Query” as a historical label unless a legacy codebase or older article clearly targets that era. When you see older examples,

extract the enduring idea and then translate the API details into the current vocabulary before adopting them.

That rule prevents three common errors:

- mixing documentation generations within the same feature,
- treating the rename as a total conceptual reset,
- assuming that any code mentioning queries is modern enough to copy directly.

The rename from React Query to TanStack Query tells you where the project sits in the ecosystem. The v5 boundaries tell you which examples are current. Together they give you a stable interpretive frame: modern React code uses '@tanstack/react-query', expects React 18.0 or later, and follows updated terminology such as HydrationBoundary and pending. Once those reference points are fixed, the surrounding ideas become much easier to read accurately, compare across sources, and apply without silently importing outdated assumptions.

2.3. Declarative Data Dependencies Instead of Effect Chains

The operational problems around server state become much easier to manage once you stop asking, “When should this component fire a fetch?” and start asking, “Which remote resource does this render depend on?” That shift sounds small, but it moves control away from component lifecycle choreography and toward resource identity. In practice, this is where many codebases either stabilize or begin to accumulate timing bugs.

A familiar failure mode appears when data loading grows outward from effects. A parent mounts, fetches a user, sets local state, then a child notices the user ID and fetches projects. Another child waits for permissions. Each fetch is technically correct, yet the application has encoded a workflow in render timing. The result is a network waterfall, duplicated state flags, and subtle races when inputs change quickly.

A minimal contrast makes the shift concrete:

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

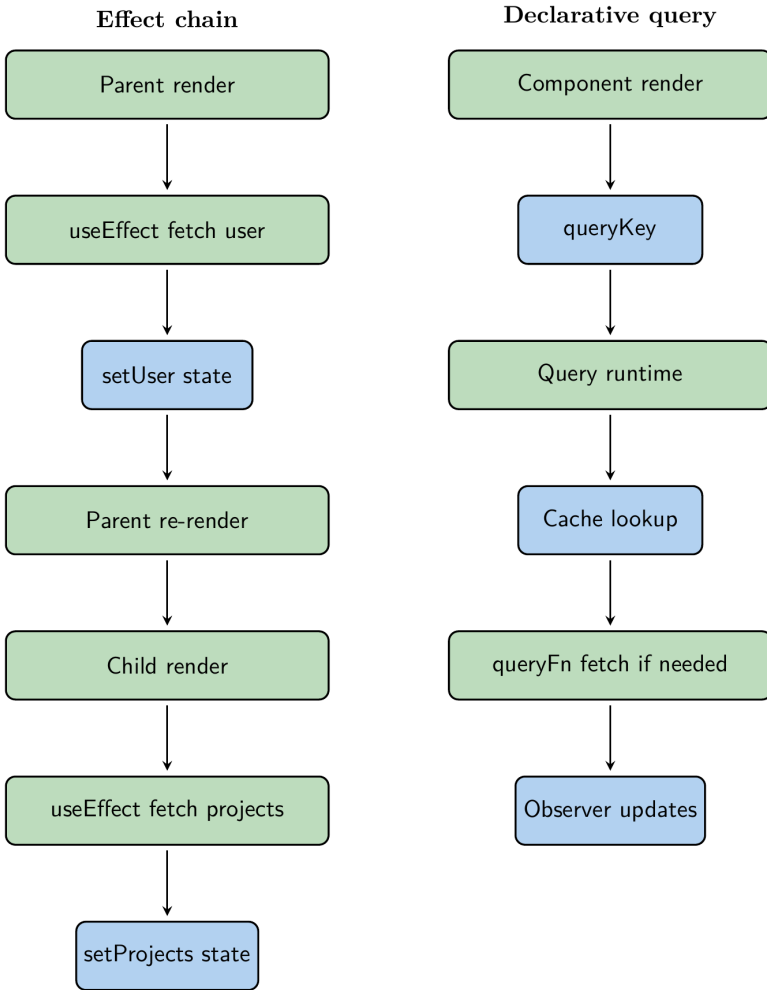
function UserPanel({ email }) {
  const userQuery = useQuery({
    queryKey: ['user', email],
    queryFn: async ({ signal }) => {
      const res = await fetch(
        '/api/users?email=' + encodeURIComponent(email),
        { signal }
      )
      if (!res.ok) {
        throw new Error('request failed')
      }
      return res.json()
    },
  })

  const userId = userQuery.data?.id

  const projectsQuery = useQuery({
    queryKey: ['projects', userId],
    queryFn: async ({ signal }) => {
      const res = await fetch(
        '/api/projects?userId=' + encodeURIComponent(userId),
        { signal }
      )
      if (!res.ok) {
        throw new Error('request failed')
      }
      return res.json()
    },
  })

  return null
}
```

Even before discussing loading states or UI output, the important difference is visible. Each data requirement is declared through `useQuery( queryKey, queryFn )`. The `queryKey` states what resource you want. The `queryFn` states how to retrieve it. The runtime now owns the mechanics around observation, reuse, deduplication, and freshness policy.



What “declarative” means here

In UI programming, “declarative” is often used loosely. Here it has a precise meaning. You do not imperatively schedule a fetch because a

component mounted, because a dependency array changed, or because a child happened to appear after a parent resolved. You declare that the current render depends on a resource identified by a specific key, and you provide a function capable of producing that resource. The library then decides whether it can serve data from cache, whether it should refetch, whether multiple observers can share in-flight work, and how to expose status changes consistently.

That division of labor matters because component lifecycle is not resource identity. A component might mount twice in development, unmount and remount due to navigation, or disappear because of conditional rendering. None of those events mean the underlying server resource changed. Effect-driven fetching ties remote reads to those accidents of rendering. Declarative querying ties them to the semantics of the data instead.

You can think of the contract like this:

- The component declares a dependency.
- The dependency is named by `queryKey`.
- The retrieval mechanism lives in `queryFn`.
- The runtime resolves that dependency against cache state, network state, and freshness rules.
- Observers receive a coherent view of status and data.

Once you view the flow in those terms, many “advanced” features stop looking exotic. Deduplication is just a consequence of two observers declaring the same dependency. Background refetching is a consequence of continuing to observe a resource whose freshness policy says it should be revalidated. Cache reuse is a consequence of the runtime recognizing a previously resolved identity.

Why keys become the identity layer

The `queryKey` is not a parameter bag attached to a fetch helper. It is the identity of the server-state dependency. That identity has to be stable enough that the runtime can answer a hard question: “Have I already seen this resource, and if so, what should happen now?”

Effect chains answer the wrong question. They ask whether some combination of local variables changed according to JavaScript equality rules inside a dependency array. That may trigger the right fetch sometimes, but it does not create an explicit, reusable identity for the resource itself. Two components can independently fetch the same endpoint with the same arguments and still have no shared concept of sameness.

A key solves that by pulling identity out of lifecycle code and making it explicit. If two parts of the application declare `['user', email]`, they are not merely running similar logic. They are observing the same named resource. That gives the runtime a basis for cache lookup, in-flight request sharing, staleness tracking, and invalidation later on.

Key design deserves a full treatment elsewhere, but a few properties matter immediately:

- Include the variables that materially define the resource.
- Keep the structure stable for equivalent requests.
- Separate identity from presentation concerns such as which component requested the data.

When teams skip that discipline, they often recreate it accidentally with ad hoc maps, memoization wrappers, and manually coordinated store slices. Query keys turn that accidental infrastructure into a first-class mechanism.

How effect chains create waterfalls

Effect-based fetching breaks down most obviously when one remote read enables another. Consider a parent that fetches a user record by email and then passes the resulting `userId` to a child, which fetches projects. The second request cannot begin until the first render completes, the effect fires, the response returns, state updates, the child renders, and its effect runs. That is not a logical dependency between the resources alone. It is a dependency mediated by component timing.

A simplified imperative version often looks like this:

```
function Parent({ email }) {
  const [user, setUser] = useState(null)

  useEffect(() => {
    let ignore = false

    async function loadUser() {
      const res = await fetch(
        '/api/users?email=' + encodeURIComponent(email)
      )
      const data = await res.json()
      if (!ignore) {
        setUser(data)
      }
    }

    loadUser()
    return () => {
      ignore = true
    }
  }, [email])

  return <Child userId={user?.id} />
}

function Child({ userId }) {
  const [projects, setProjects] = useState([])

  useEffect(() => {
    if (!userId) return

    let ignore = false
```

```
    async function loadProjects() {
      const res = await fetch(
        '/api/projects?userId=' + encodeURIComponent(userId)
      )
      const data = await res.json()
      if (!ignore) {
        setProjects(data)
      }
    }

    loadProjects()
    return () => {
      ignore = true
    }
  }, [userId])

  return null
}
```

Nothing about that code is absurd. The problem is that the component tree has become the orchestrator for the data graph. As the graph grows, your fetch behavior becomes sensitive to render order, mount timing, and cleanup correctness. That is exactly the kind of manual synchronization React warns about when direct fetching in effects becomes hard to optimize and difficult to compose for caching or server rendering.

With declarative queries, the component tree can still express dependency relationships, but the runtime owns the data lifecycle. If two reads are independent, declare them independently. If one read depends on data from another, express that dependency as input availability rather than as a chain of imperative effects. The meaning shifts from “run this effect after that state update” to “this resource exists once this identity can be formed.”

Queries are read contracts, not fetch calls

A query is best understood as a read-side contract around an asynchronous resource. The hook call does initiate work when needed, but reducing it to “a fetch hook” misses the point. A query captures

identity, retrieval, observation, and cache participation together. That is why the canonical shape is `useQuery( queryKey, queryFn )`: one field says what the resource is, the other says how to retrieve it.

This model explains several runtime behaviors that feel awkward in an effect-first design:

- multiple observers can subscribe to the same resource,
- an already cached result can be shown before a refetch finishes,
- stale resources can be revalidated without throwing away useful data,
- status reporting can distinguish request progress from initial absence of data.

Those behaviors are natural once the runtime sees a query as a persistent resource contract rather than as one component's private fetch.

Mutation emerges as the write-side counterpart

The same architectural move that clarifies queries also clarifies mutations. Reads and writes are not symmetric. A read describes an observable resource that can be cached, shared, and revalidated. A write describes an action against server state that has execution status and downstream consequences. The conceptual pairing matters even before you get into mutation APIs in detail.

If your mental model is effect-driven, writes often become a grab bag of event handlers that call `fetch`, set some local booleans, then manually force whatever screens seem affected to reload. If your mental model is declarative, a mutation is understood in relation to the read model it disturbs. The write itself reports its own status; the cache and query identities determine which observed resources need to be updated or

invalidated. That does not make mutation logic trivial, but it gives it a stable place in the architecture.

Cancellation becomes part of the query contract

One place where declarative querying integrates cleanly with the platform is cancellation. TanStack Query passes an `AbortSignal` to the query function. If you consume that signal in `fetch`, the runtime can coordinate cancellation-aware behavior without the ad hoc cleanup flags typical of effects.

A cancellation-aware query function looks like this:

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

function ProjectList({ userId }) {
  const projectsQuery = useQuery({
    queryKey: ['projects', userId],
    queryFn: async ({ signal }) => {
      const res = await fetch(
        '/api/projects?userId=' + encodeURIComponent(userId),
        { signal }
      )

      if (!res.ok) {
        throw new Error('request failed')
      }

      return res.json()
    },
  })

  return null
}
```

This matters for more than cleanup aesthetics. In effect code, you often suppress stale updates with a local `ignore` flag after unmount or parameter change. That prevents a state write, but it does not necessarily stop the network work or integrate the result coherently with cache state. By consuming the provided `signal`, you align your request with the runtime's lifecycle decisions. The cancellation semantics become part of the resource model rather than a patch around component teardown.

There is an important nuance in the underlying behavior: the runtime provides the signal, but cancellation only takes effect at the transport layer if your query function actually uses it. That is a good example of TanStack Query's design style. The library gives you the control boundary and coordination mechanism; you still need to wire the platform primitive correctly.

Why declarative does not mean passive

A common misunderstanding is that declarative data dependencies somehow make fetching static or opaque. The opposite is true. A declarative query is an observed resource with policy. Defaults such as stale refetch triggers mean the runtime is not merely caching a one-time promise result. It is actively managing when a resource should be rechecked under conditions like remount, window focus, or reconnect when the data is stale.

That behavior is hard to reproduce cleanly with effect chains because the logic is cross-cutting. A focus listener should not live inside every component that cares about fresh data. Reconnect behavior should not be hand-rolled on each screen. Once dependencies are declared by identity, the runtime can apply shared policy across all observers of those identities.

This is also why the declarative model scales better under concurrency. As more components read overlapping resources, the number of possible timing interactions grows quickly. With effects, each component owns its own request orchestration, so the interaction surface grows combinatorially. With queries, coordination collapses onto shared identities and a central cache-aware runtime.

When effects are still the right tool

Replacing effect chains with declarative queries does not mean effects become obsolete. Effects remain appropriate when you need to syn-

chronize with external systems that are not best modeled as reusable server resources. Examples include wiring browser APIs, starting subscriptions, integrating imperative widgets, or triggering side effects that do not represent a cacheable read model.

The useful test is operational reuse. If the work represents a remote resource that multiple parts of the application may observe, that needs identity, freshness handling, and cache participation, an effect is usually too low-level. If the work is a one-off imperative bridge with no cache semantics, an effect is often the right abstraction.

This distinction prevents a different category error: pushing every asynchronous action into query machinery whether or not it behaves like server state.

A better control boundary

Once you declare dependencies with `queryKey` and `queryFn`, you stop scattering orchestration logic across components. The component expresses what it needs. The runtime resolves how that need should be satisfied given cache contents, in-flight requests, cancellation, and freshness policy. That boundary is the real architectural move. It turns data loading from a set of lifecycle tricks into a resource model with explicit identity, shared observation, and predictable coordination.

2.4. The Operational Responsibilities TanStack Query Owns

Once you adopt declarative queries, the most important architectural question is no longer how to fetch data from a component. It is where the operational boundary sits. Teams often under-assign responsibility and treat TanStack Query as a thin wrapper around `fetch`. Just as often, they over-assign responsibility and expect it to behave like a

universal application store, a normalized entity graph, or a workflow engine. Both mistakes create friction because they hide the real value of the runtime: it specializes in managing the difficult lifecycle of server state after you have declared what resource you need.

A small example makes that boundary visible. The code below is intentionally modest, but it already delegates several responsibilities that are awkward to implement consistently by hand:

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

function Orders({ customerId }) {
  const ordersQuery = useQuery({
    queryKey: ['orders', customerId],
    queryFn: async ({ signal }) => {
      const res = await fetch(
        '/api/orders?customerId=' +
          encodeURIComponent(customerId),
        { signal }
      )

      if (!res.ok) {
        throw new Error('request failed')
      }

      return res.json()
    },
  })

  if (ordersQuery.isPending) {
    return <p>Loading...</p>
  }

  if (ordersQuery.error) {
    return <p>Request failed</p>
  }

  return (
    <ul>
      {ordersQuery.data.map(order => (
        <li key={order.id}>{order.number}</li>
      ))}
    </ul>
  )
}
```


That single query declaration says nothing about request deduplication, stale-versus-fresh transitions, background refetch on focus, reconnect behavior, inactive cache retention, or cancellation wiring. Yet those concerns do not disappear. The runtime takes them on. That is the core control boundary: you declare server-state dependencies and consume their observable state; TanStack Query manages the lifecycle mechanics attached to those dependencies.

What the runtime owns

TanStack Query owns the operational layer around reads and, in a complementary way, around writes. On the read side, it tracks in-flight work, indexes cached results by identity, coordinates multiple observers of the same resource, and exposes a coherent status surface to the UI. It also applies freshness and retention policy over time rather than treating a response as an eternal value.

Those responsibilities are not cosmetic conveniences. They are necessary because remote data changes outside your process, can fail transiently, can be requested by multiple views simultaneously, and often remains useful even when a newer request is underway. If you try to implement those behaviors screen by screen, you do not merely write more code. You create inconsistent semantics across the application.

A useful way to think about the runtime is as the server-state control plane inside the client. It does not own the domain truth, but it does own the local policy and coordination needed to observe that truth effectively.

Freshness is policy, not a cache accident

One of the clearest responsibilities TanStack Query owns is freshness. A response placed in memory is not automatically valid forever. By default, cached data becomes stale immediately unless you configure `staleTime`. That default often surprises teams because they assume

caching means “do not fetch again.” TanStack Query uses a different model: caching answers whether data can be reused locally; freshness answers whether that data should be treated as current enough to avoid revalidation.

That split is essential. A query can have cached data and still be stale. In that state, the runtime can show the cached result immediately while also deciding that the resource should be checked again under the right trigger. This allows the UI to avoid unnecessary loading gaps without pretending that an old snapshot is perpetually authoritative.

You should treat `staleTime` as a statement about tolerated staleness, not as a performance-only knob. A short `staleTime` increases synchronization pressure and network activity but keeps data closer to the source of truth. A longer `staleTime` reduces refetch frequency and improves perceived responsiveness, but it expands the window during which users may see an outdated snapshot. Different resources deserve different values because different business operations tolerate different freshness windows.

There is also an important edge case: `staleTime` set to `'static'` is stricter than `Infinity` for some invalidation and refetch behaviors. You should not treat those values as interchangeable shortcuts for “never stale.” They encode different operational intent.

Retention is separate from freshness

Another common misunderstanding is to collapse freshness and memory retention into one idea. TanStack Query keeps them separate. Freshness answers whether cached data is current enough to trust without revalidation. Retention answers how long an inactive query remains in memory after nothing is observing it.

Inactive queries stay cached until `gcTime` expires, after which they are garbage-collected. This separation matters because a query can be

stale but still worth keeping in memory, especially if the user is likely to revisit the same screen soon. Conversely, a query could have been fresh when last observed but still be removed later because it no longer justifies occupying memory.

That lifecycle gives you a more precise model than simple memoization:

1. fetch or reuse a result,
2. mark it fresh for some period if configured,
3. allow it to become stale,
4. keep it cached while inactive,
5. remove it after `gcTime` if no observer returns.

This is one reason TanStack Query should not be confused with a plain global store. A store typically retains whatever you put into it until you explicitly replace or remove it. TanStack Query applies time-based lifecycle policy because server-state snapshots have operational value that decays.

Background synchronization is part of the job

Freshness policy only matters if the runtime can act on it. By default, stale queries may refetch on mount, window focus, and reconnect. These behaviors often look surprising in development because developers mentally translate “cached” into “done.” In a server-state model, they are better understood as revalidation triggers. The application is not discarding cache value; it is recognizing that certain moments offer strong opportunities to resynchronize with the server.

Window focus is a good example. A user might leave a tab open for minutes or hours, return, and expect the data to reflect current reality.

If the query is stale, refetching on focus is usually more correct than continuing to show an old snapshot indefinitely. Reconnect serves the same purpose after network interruption. Mount-based refetching handles the common case where a screen is revisited and its data should be checked again if its freshness window has expired.

These defaults are not universally ideal, but they are grounded in how remote data behaves. The point is not that every application should keep the defaults forever. The point is that someone has to own synchronization triggers, and a central runtime is the right place to do it.

Retries and failure policy belong in the runtime

Transient failure is normal in networked systems. A central server-state runtime can therefore make failure handling more consistent by owning retry behavior instead of leaving each screen to invent its own rules. If one component retries aggressively, another never retries, and a third retries in a blocking loop, the application presents a chaotic reliability model to users.

TanStack Query's value here is not merely that it can retry. The deeper value is that retries become a policy surface attached to a named resource lifecycle rather than an improvised pattern in each effect or event handler. Centralized retry behavior also improves operability because you can reason about failure modes in one place instead of diffusing them across the tree.

This same logic applies to status exposure. A query with cached data and a background refetch in flight should not necessarily collapse to the same UI state as a query that has never loaded at all. By standardizing request lifecycle handling, the runtime gives you a consistent vocabulary for rendering failure and recovery states.

Cancellation and race coordination

Cancellation is another operational responsibility that belongs with the

runtime. Query functions receive an `AbortSignal`. If you pass that signal to `fetch`, the request becomes cancellation-aware in a way that aligns with query observation and lifecycle changes.

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

function Customer({ id }) {
  const customerQuery = useQuery({
    queryKey: ['customer', id],
    queryFn: async ({ signal }) => {
      const res = await fetch(
        '/api/customers/' + encodeURIComponent(id),
        { signal }
      )

      if (!res.ok) {
        throw new Error('request failed')
      }

      return res.json()
    },
  })

  return null
}
```

You can also cancel queries manually through the client:

```
queryClient.cancelQueries({
  queryKey: ['customer', id],
})
```

This matters because race conditions in server-state code rarely come from one `fetch` alone. They come from changing inputs, multiple observers, route transitions, and overlapping requests. The runtime cannot eliminate those conditions by magic, but it can provide coordinated behavior around them. If you consume the provided `signal`, cancellation becomes part of the request contract rather than an afterthought implemented with local flags.

Why these responsibilities are not incidental library features

The responsibilities above can look like framework-specific extras un-

til you view them through HTTP and distributed-system semantics. Remote data systems already require cache keying, freshness validation, reuse decisions, and conditions for serving stale responses. HTTP caching rules make this explicit: correct reuse depends on more than a URL string, and staleness is a real property with validation consequences. TanStack Query does not create those concerns; it gives frontend code a structured way to manage them at the application layer.

That perspective matters because it protects you from two bad architectural instincts. The first is thinking that you can ignore freshness and synchronization because “it is just JSON over HTTP.” The second is thinking that TanStack Query is somehow over-engineered because it handles stale data, refetch triggers, and cache lifecycle. Those behaviors are not ornamental. They reflect the actual operating conditions of remote state.

What TanStack Query deliberately does not own

The runtime is specialized, not universal. It does not replace domain modeling, authorization logic, transport concerns, or backend correctness. It does not design your API. It does not decide which resources exist or what invariants they must satisfy. It does not replace router state, local UI state, or form draft state.

It also does not automatically normalize your data into a relational client graph. This is a deliberate scope choice. Some state-management systems encourage a fully normalized entity cache with automatic merging and cross-view propagation. TanStack Query instead leans heavily on query identity, invalidation, and refetch as the primary consistency tools. That choice avoids a large amount of complexity for many applications, but it means you should not expect the runtime to infer every relationship among overlapping projections automatically.

That boundary is healthy. A normalized graph can be powerful in

specialized domains, but it requires strong assumptions about entity identity, field-level merging, and relationship maintenance. TanStack Query does not impose that model as the default shape of server-state management.

Why invalidation and refetch often beat manual cache surgery

Because TanStack Query is not a universal normalized graph, teams sometimes assume it is less capable. The more accurate view is that it optimizes for a different trade-off. Invalidating and refetching a named resource is often simpler and safer than manually editing every cached projection that could include a changed entity. The runtime makes this practical by giving you durable identities and shared observation.

Manual cache updates still have a place, especially when latency or UX requires immediate local reflection of a write. But those updates should be deliberate exceptions, not the default consistency strategy for every mutation. If you try to maintain a perfect client-side graph for all server data, you may spend more effort on synchronization logic than the application warrants.

The runtime boundary supports that trade-off: own the mechanics of cache-backed reads, stale/fresh transitions, retries, refetching, retention, and coordinated observation; leave domain-specific consistency policies and API semantics to application code.

A quick bug taxonomy

A useful diagnostic habit is to map recurring frontend bugs to the responsibility boundary they reveal:

- duplicated fetches usually indicate missing or inconsistent query identity,
- stale reads usually indicate a missing freshness policy,

- race conditions usually indicate missing cancellation or observer coordination,
- inconsistent retry behavior usually indicates failure policy is scattered through components,
- vanished data after navigation usually indicates cache retention is being misunderstood.

This framing changes how you debug. Instead of patching a specific component with another effect or boolean flag, you ask which operational responsibility has no clear owner.

Version boundary and runtime assumptions

One compatibility fact matters directly to this operational boundary: v5 requires React 18 because it relies on `useSyncExternalStore`. That matters less as a migration footnote than as a clue about architecture. TanStack Query is not just wrapping `fetch`; it is integrating an external, cache-backed, observable data source into React with modern subscription semantics. The runtime is substantial enough that the React integration needs the correct external store contract.

That further reinforces the scope distinction. A tool built on top of external store synchronization is operating at the level of shared runtime state management, not at the level of convenience helpers attached to individual components.

Using the boundary correctly

You get the most value from TanStack Query when you let it own the responsibilities it is designed for and resist pushing unrelated concerns into it. Keep long-lived UI preferences, modal state, and in-progress form drafts elsewhere. Keep domain invariants and permission rules in application and backend logic. Model server reads and writes through query and mutation abstractions so the runtime can apply cache iden-

tity, freshness policy, revalidation triggers, retries, cancellation, and retention coherently.

Once you hold that line, the library becomes easier to reason about. It is neither a global junk drawer for every piece of state nor a trivial fetch wrapper. It is the part of your frontend architecture that manages the operational reality of observing server-owned data under latency, failure, reuse, and change.

Chapter 3

Bootstrapping the Query Runtime

Most teams first meet TanStack Query through ‘useQuery’, but production behavior is determined much earlier—at client construction, provider placement, and default policy selection. This chapter treats bootstrapping as an architectural act: the moment where you define isolation boundaries, synchronization posture, and operational observability for every query and mutation that will follow in the rest of the book.

3.1. Designing the QueryClient as Shared Infrastructure

A surprising amount of TanStack Query instability comes from treating `QueryClient` like a disposable helper instead of runtime infrastructure. The failure usually looks innocuous: a component creates a new

client during render, navigation remounts that component, and the application suddenly loses cache continuity. Queries refetch as if the user opened a fresh session, in-flight observers detach, devtools appear to “forget” prior entries, and teams often blame the library rather than the lifetime mistake. The root cause is architectural: the client owns the caches, coordinates query observers, applies defaults, and tracks mutations. If you recreate the client, you recreate the runtime.

A minimal browser bootstrap makes the lifetime expectation concrete. Create one client for the application session, outside transient render paths, and pass that stable instance into the provider:

```
import React from 'react'
import { createRoot } from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'

import { App } from './App'

const queryClient = new QueryClient()

createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  </React.StrictMode>
)
```

That short snippet carries an important design decision. The `queryClient` is not owned by `App`; it is owned by the application bootstrap. Route changes, component remounts, error boundaries, and local state transitions do not get to decide whether the cache still exists. Once you internalize that ownership rule, many apparently unrelated issues become easier to reason about.

What the client actually owns

You need precise vocabulary because later policy and provider deci-

sions depend on it. `QueryClient` is the top-level runtime object. It coordinates a `QueryCache`, a `MutationCache`, and application-wide defaults. The `QueryCache` stores query data, timestamps, metadata, and state. The `MutationCache` tracks mutation execution and lifecycle state. The client also exposes the imperative control plane used by infrastructure code: `fetchQuery`, `prefetchQuery`, `getQueryData`, `setQueryData`, `invalidateQueries`, `cancelQueries`, `removeQueries`, `resumePausedMutations`, and related methods.

That means a client instance is an isolation boundary, not just a container. Two subtrees connected to different clients do not share query entries, mutation state, invalidation effects, or activity counts. If one client invalidates a query, the other client does nothing. If one client has warm cached data, the other client still sees a cold start. When you choose where to instantiate a client, you are choosing which parts of the system are allowed to coordinate.

Why render-time construction is incorrect

The most common anti-pattern is creating the client in a component body without stabilization:

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

export function AppShell() {
  const queryClient = new QueryClient()

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

This code may appear to work during a quick smoke test, but every render creates a fresh runtime. In development, Strict Mode can expose

the problem early because components may mount more than once, but production is not safe either. Any remount caused by routing, feature flags, error recovery, layout replacement, or host integration can reset the client. The resulting behavior is expensive and confusing:

- Cached server state disappears unexpectedly.
- Identical queries refetch after transitions that should have reused data.
- Inactive entries never survive long enough to benefit from the default `gcTime` window.
- Mutation coordination can appear to reset mid-session.
- Observability tools show fragmented runtime history.

If you need to create the client inside a React-owned entry point, make the instance stable for that owner. A common browser pattern is lazy state initialization:

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

export function Root() {
  const [queryClient] = React.useState(
    () => new QueryClient()
  )

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

The key is not `useState` itself. The key is lifecycle ownership: the root component owns one client for as long as that root remains mounted.

Browser SPA lifetime: one client per application session

In a browser-only SPA, you usually want one long-lived `QueryClient` for the lifetime of the application root. That gives you the behavior users expect from a persistent client-side runtime: route transitions reuse cached server state, active observers can detach and reattach without discarding data, and background refetching operates against one coherent cache instead of a series of short-lived ones.

This design has direct memory and correctness implications. TanStack Query keeps inactive queries in cache for a default `gcTime` of five minutes. That retention window only means anything if the owning client survives longer than the component that first requested the data. In a stable SPA client, navigating away from a screen and back a minute later can reuse still-cached data while deciding whether it should refetch according to freshness policy. In a recreated client, the query entry is gone immediately, so every navigation behaves like a cold boot.

You should also treat the SPA client as the natural home for runtime policy. Defaults for retries, stale behavior, network posture, and focus refetching belong on the long-lived client because those policies define how the application synchronizes with the server over time. A client that disappears whenever the view tree churns cannot enforce a coherent policy surface.

Request-scoped ownership on the server

Server rendering changes the lifetime rule because the isolation problem changes. In a browser, sharing one client across the session is usually desirable. On the server, sharing one client across requests is often a data leak.

Each incoming request should be treated as its own lifecycle owner. Create a fresh `QueryClient` for that request, use it for prefetching

or request-local rendering work, and then let it become eligible for cleanup when the response is complete. Do not keep a process-wide singleton client for all users. If you do, cached data from one request can influence another request's output, contaminate freshness assumptions, and blur user boundaries in ways that are hard to detect under low load but dangerous in production.

The pattern is simple even when the surrounding server framework varies:

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

export function createRequestQueryClient() {
  return new QueryClient()
}

export async function handleRequest(req) {
  const queryClient = createRequestQueryClient()

  // request-local prefetching and rendering work
  // queryClient is not reused for another request
}
```

The implementation details of SSR hydration differ by framework and belong with hydration mechanics, but the ownership principle is stable: the server-side client is request-scoped, not application-global. That boundary protects privacy and preserves correctness. It also keeps memory usage more predictable because query entries do not accumulate across unrelated users in a long-lived server singleton.

Why server singletons are especially risky

A server singleton can fail in subtle ways long before it fails catastrophically. Suppose user A triggers a request that prefetches account-specific data under a valid query key. If user B later renders a page that matches the same key structure, the shared server cache may serve stale or unauthorized data into rendering logic unless every prefetch path and key design is perfectly partitioned. Even if you avoid direct data leakage, timestamps and stale-state decisions from one request

can affect another request's behavior.

Memory behavior also degrades. A long-lived server process handling many requests will keep accumulating query entries until garbage collection removes inactive ones, but those entries reflect cross-request traffic, not one user session. The result is avoidable cache churn in a process that should generally prioritize isolation over reuse. Warm-cache efficiency on the server is not worth violating request boundaries.

Embedded application islands and microfrontend seams

Not every browser surface should share one client. Embedded widgets, microfrontends, tenant-isolated consoles, and independently deployed application islands often benefit from separate clients. The question is not whether multiple clients are allowed; it is whether the surfaces should coordinate as one runtime.

Use distinct clients when the surfaces differ in one or more of these ways:

- They are deployed independently and cannot rely on shared bootstrap order.
- They should not share cache data for privacy or tenancy reasons.
- They have intentionally different defaults for retries, focus behavior, or network mode.
- They may be mounted and unmounted independently on the same page.
- A host page should not gain visibility into the guest's cached server state.

In that environment, isolation is a feature. A widget embedded into a marketing site should not silently share server-state cache with

the host application's admin shell just because both happen to use TanStack Query. Separate clients prevent accidental coupling. The trade-off is that identical queries across those boundaries will not deduplicate or share warm results. That is acceptable when independence matters more than reuse.

Tests, sandboxes, and preview environments

Tests should almost always create a fresh client per test case or per isolated render. Shared test clients cause state bleed: one test warms data, another test reads it unexpectedly, and failures become order-dependent. The same rule applies to component sandboxes such as design previews or Storybook-like environments. Each story or render fixture should decide whether it needs a clean runtime or an explicitly seeded one.

A simple test helper makes the pattern explicit:

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

export function createTestWrapper() {
  const queryClient = new QueryClient()

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

You can tune defaults for tests later if needed, but isolation comes first. A deterministic test suite depends on being able to say exactly which cache entries exist at the start of each case. Reusing a client across unrelated tests undermines that guarantee.

Custom caches and global hooks belong at construction time

Treating the client as infrastructure also changes where you attach cross-cutting concerns. If you need application-wide query or mutation callbacks, create explicit `QueryCache` and `MutationCache` instances when you construct the client:

```
import {
  MutationCache,
  QueryCache,
  QueryClient,
} from '@tanstack/react-query'

const queryClient = new QueryClient({
  queryCache: new QueryCache({
    onError: (error, query) => {
      console.error('query failed', query.queryKey, error)
    },
  }),
  mutationCache: new MutationCache({
    onError: (error, variables, context, mutation) => {
      console.error(
        'mutation failed',
        mutation.options.mutationKey,
        error
      )
    },
  }),
})
```

This is another reason not to scatter client creation across components. Infrastructure hooks such as telemetry, audit logging, or global failure reporting should be registered once at the runtime boundary, not repeated opportunistically in view code. The mutation side is especially important because global `MutationCache` callbacks always run, which makes them appropriate for must-run operational concerns.

A practical ownership checklist

When you decide where a client should live, work through the execution environment rather than the component tree.

1. Identify the environment boundary: browser session, server request, test run, embedded widget, or isolated application shell.
2. Assign one lifecycle owner for that boundary.
3. Create exactly one `QueryClient` per owner.
4. Verify that ordinary remounts inside the boundary do not replace the client.
5. Verify that crossing out of the boundary does replace the client.

That last distinction matters. In a browser SPA, route transitions should not replace the client, but a full page reload should. On the server, one request should never reuse another request's client. In tests, one case should not inherit another case's cache. In a microfrontend host, mounting a second isolated island should not accidentally attach to the first island's runtime.

Decision signals for choosing lifetime scope

Several concrete signals help you choose between sharing and isolation.

If the cost of duplicated network requests is your main concern and the surfaces already trust each other, a broader client scope is usually appropriate. If privacy boundaries, tenant separation, or independently managed deployments dominate, isolate clients even if you lose cache reuse. If you need stable, session-length background synchronization, favor one browser client. If you need strict cleanup and zero cross-user bleed, favor per-request or per-test clients.

Do not couple client creation to transient UI state such as the selected tab, modal visibility, or route segment unless that state truly defines an isolation domain. Most of the time it does not. If changing a view conceptually stays within the same application session, replacing the client is almost always the wrong move.

Validating the design in practice

After you choose an ownership model, validate it with behavior rather than assumptions. In a SPA, navigate between screens that use the same query keys and confirm that cached data survives ordinary re-mounts. In a server-rendered environment, inspect request handling so each request gets a fresh client instance. In tests, run cases in different orders and confirm they do not influence one another. In embedded contexts, verify that invalidation in one island does not unexpectedly affect another.

If these checks fail, resist the urge to tune stale times or disable refetch triggers first. Lifetime mistakes often masquerade as freshness or invalidation problems. A stable, correctly scoped `QueryClient` gives the rest of the runtime a trustworthy foundation: one place where cache identity, mutation coordination, memory retention, and policy ownership are aligned with the real execution boundaries of the system.

3.2. Composing QueryClientProvider Boundaries for Real Applications

Two components can issue the same query key, run the same query function, and still fail to share cached data if they live under different `QueryClientProvider` boundaries. That failure is easy to misread. The hooks look identical, network requests look duplicated, and the bug appears to be about query identity. Often it is really about provider topology. `QueryClientProvider` is the visibility boundary for the runtime: hooks read the client from React context, and that context determines which cache, mutation state, defaults, and imperative operations a subtree can see.

A standard root-level topology makes the intended sharing model explicit. For a single-page application with one coherent runtime, place

one provider near the root and attach the long-lived client there:

```
import React from 'react'
import { createRoot } from 'react-dom/client'
import {
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query'

import { App } from './App'

const queryClient = new QueryClient()

createRoot(document.getElementById('root')).render(
  <QueryClientProvider client={queryClient}>
    <App />
  </QueryClientProvider>
)
```

That arrangement does more than make hooks work. It declares that every component under `App` participates in the same query runtime. Cache entries can be reused across routes, invalidation can coordinate across features, and global activity counts describe one shared system instead of several isolated ones.

Provider boundaries define runtime visibility

`QueryClientProvider` requires a `client` prop and scopes that client to a React subtree. Any hook that resolves the current client, including `useQueryClient()`, reads from the nearest provider unless you explicitly pass a client argument. That nearest-match rule is what makes provider placement an architectural decision rather than boilerplate. If a subtree sits under a nested provider, the nested client becomes the active runtime for that subtree even if an outer provider exists.

Once you frame the provider as a visibility boundary, several practical questions become crisp:

- Which parts of the UI should reuse the same cached server state?
- Which surfaces should coordinate invalidation and mutations?

- Which remounts are allowed to reset runtime access?
- Which embedded regions should remain isolated even on the same page?

Those questions are orthogonal to query-key design. Good keys cannot bridge two different clients. If two components need to share data, they must have both matching keys and access to the same provider-scoped client.

The default choice: one root provider for one shared app

For most SPAs, a single root provider is the right starting point because the user experiences the application as one interactive system. If a search panel, detail view, and navigation badge all consume related server state, you usually want them to deduplicate requests, observe the same freshness state, and react to the same invalidations. A root provider maximizes those benefits while keeping the integration model simple.

This choice also reduces accidental fragmentation. If you place the provider above the router, route transitions do not change which client query hooks resolve. The cache survives screen changes, and components mounted on different routes can still reuse entries created elsewhere in the session. In practice, that often matters more than any single query optimization because real applications move users through many route boundaries while expecting continuity of server state.

You should treat this as the baseline until you have a strong reason to split runtimes. Additional providers are easy to add later, but unintentional fragmentation becomes hard to diagnose once features begin depending on isolated caches.

Nested providers shadow outer clients

Nested providers do not merge. The inner provider shadows the outer

client for its subtree. That shadowing is powerful when intentional and dangerous when accidental.

Consider a shell that mounts an isolated admin area inside a public app:

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

const publicClient = new QueryClient()
const adminClient = new QueryClient()

export function Root() {
  return (
    <QueryClientProvider client={publicClient}>
      <PublicLayout />
      <QueryClientProvider client={adminClient}>
        <AdminShell />
      </QueryClientProvider>
    </QueryClientProvider>
  )
}
```

Here, `PublicLayout` and `AdminShell` are intentionally separate run-times. They may render on the same page, but they do not share cache entries or mutation coordination. That is often exactly what you want for an embedded console, privileged management surface, or tenant-isolated module.

What you should not expect is partial inheritance. The admin subtree does not fall back to the public client for missing queries. It reads and writes only its nearest client unless code explicitly passes a different client into APIs such as `useQueryClient(queryClient)`. That override exists, but you should use it sparingly. Once code frequently jumps across provider boundaries, the topology is no longer clear.

Route-scoped providers are valid but high-friction

Placing providers at route or layout boundaries can be correct, but only

when the route boundary is also a runtime boundary. Many teams place a provider inside a route component because it feels local and tidy. The cost arrives later: every route change remounts the provider, the client becomes route-scoped by accident, and cached state that should have survived navigation disappears.

Route-scoped providers make sense when routes are deliberately isolated. Examples include a public storefront and a separately governed internal tool delivered from the same host shell, or a multitenant interface where changing the active tenant should replace the entire server-state runtime. In those cases, route changes genuinely cross an isolation domain, so replacing the provider is appropriate.

The key test is operational, not stylistic: if a user navigates between two screens and you expect cached data, query status, and invalidation effects to carry across that transition, do not put the provider inside the branch that remounts. If you expect a hard reset, route scoping may be a good fit.

Provider placement and remount behavior

Provider topology is inseparable from remount topology. React context is stable only while the provider instance remains mounted. If you move a provider into a layout branch that frequently unmounts, everything below that branch loses access to the old client and binds to a new one when the branch remounts. The resulting symptoms often look like cache bugs:

- queries reissue after tab switches or layout changes;
- mutation side effects appear local to one branch;
- devtools show separate caches for what looks like one app;
- loading indicators reset on navigation.

When you audit provider placement, ignore the visual structure of the UI for a moment and ask a lower-level question: which React nodes are stable across the user flows that should preserve query runtime state? That stable region is where the provider belongs.

Using `useQueryClient` to confirm boundary intent

`useQueryClient()` returns the nearest provider client unless you override it. That makes it a precise diagnostic tool for boundary mistakes. If two components you expect to coordinate are resolving different clients, the problem is not key design or invalidation logic. It is topology.

A minimal diagnostic pattern is straightforward:

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

export function RuntimeProbe({ name }) {
  const queryClient = useQueryClient()

  console.log(name, queryClient)

  return null
}
```

Render probes in suspicious subtrees and compare the logged client references. If components that should share a runtime resolve different objects, you have either duplicated providers or placed one provider too low. If components that should be isolated resolve the same object, your boundary is too high.

The hook also confirms a subtle point about legality. Query hooks and client access are not globally available just because a client exists somewhere in the application. They are available only inside the provider-scoped subtree, unless you deliberately thread a client override into low-level infrastructure code.

Multi-root pages and embedded widgets

Modern web pages often contain multiple React roots. A marketing page might host a public search widget, an authenticated account panel, and a support chat mount point, each bootstrapped independently. In that world, provider topology is not just about nested trees; it is also about separate roots that cannot share context at all.

For independently mounted roots, you normally create a distinct `QueryClientProvider` per root. That isolates runtime ownership cleanly and avoids hidden coupling between code loaded at different times. If two roots truly need to share cache, you need a higher-level integration strategy than provider nesting because React context does not cross root boundaries. In many applications, it is better to accept separate caches than to build brittle cross-root coordination.

Partial migrations create a similar constraint. Suppose you are moving an older application incrementally into a React surface that uses TanStack Query. During the migration, a local provider around the new island is often the safest choice. It contains the runtime to the migrated region without forcing the legacy shell to adopt the same life-cycle or policy assumptions immediately.

SSR and request-scoped provider composition

Server rendering adds one more dimension: the provider boundary in the component tree must align with request-scoped client ownership. You still render `QueryClientProvider client={queryClient}`, but the `queryClient` itself should be created per request rather than reused across users.

A minimal shape looks like this:

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

export async function renderRequest(App) {
```

```
const queryClient = new QueryClient()

const dehydratedState = dehydrate(queryClient)

return {
  element: (
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  ),
  dehydratedState,
}
```

The exact SSR and hydration wiring varies by framework, so keep the main decision separate from the adapter details. The provider should wrap the subtree that needs access to the request-local client, and that boundary should remain stable when hydration attaches on the client. In v5, hydration behavior changed for existing queries, so the placement of the provider around hydrated subtrees can affect when updates appear to land if the client already contains data. That is one more reason to avoid casual provider duplication in hybrid trees.

Async Server Components are a documented exception to the general stable client rule. Their execution model differs from ordinary client-side rerenders, so creating a new `QueryClient` there does not carry the same meaning as recreating one in a browser component body. Treat that as a framework-specific execution boundary, not as a license to construct clients freely in interactive client components.

Testing boundary design under real navigation

You can validate provider topology with a short operational checklist.

1. Inventory your application roots and major layout boundaries.
2. Decide which surfaces should share one runtime.
3. Place providers at the highest stable node that matches that shar-

ing domain.

4. Navigate across routes and remount layouts intentionally.
5. Verify whether data reuse matches the intended boundary.

A useful sanity test is to load data in one screen, navigate away, then return through a path that remounts part of the UI. If the cache should have been shared but the application behaves like a cold start, inspect the provider boundary before changing query options. If two surfaces should have been isolated but one invalidation refreshes the other, your provider is too broad.

Common anti-patterns

Several mistakes recur in production codebases.

Provider nesting without intent. A nested provider created for local convenience silently splits the runtime. The inner subtree stops sharing with the outer application.

Route-local providers in frequently remounting branches. Developers place a provider where it is easiest to import, not where it is semantically stable. Navigation then resets the client.

Using extra providers to hide poor key design. Isolation can mask collisions temporarily, but it also destroys reuse and makes later integration harder. Fix key identity separately.

Assuming identical hooks imply shared cache. Hook code can be textually identical while targeting different clients. Context decides sharing.

Cross-boundary overrides as normal practice. `useQueryClient(queryClient)` can be useful in infrastructure code, but regular feature code should not routinely bypass the nearest provider. Frequent overrides are a sign that the boundary map is

wrong.

Choosing the topology deliberately

A single root provider improves reuse, simplifies invalidation reasoning, and produces one coherent observability surface. Multiple providers improve isolation, deployment independence, and security separation, but they fragment cache state and duplicate some network work. Neither model is universally better. The right choice follows from execution boundaries: one user-facing runtime gets one provider domain; distinct runtime domains get distinct providers.

When that mapping is explicit, TanStack Query behaves predictably. Components that should coordinate can see the same cache. Components that should remain separate cannot accidentally bleed state across boundaries. Provider placement stops being wrapper ceremony and becomes part of the application's runtime architecture.

3.3. Encoding Global Defaults as Runtime Policy

A stable `QueryClient` and well-chosen provider boundaries give you one coherent runtime. The next design decision is what that runtime believes about the network, the server, and your users. If every team sets `staleTime`, retry counts, and refetch triggers ad hoc at the call site, the application stops behaving like one system. One page silently tolerates stale data for minutes, another refetches on every focus change, a third retries failed requests aggressively enough to amplify an outage, and debugging becomes a comparison of local habits instead of an analysis of deliberate policy.

TanStack Query gives you a policy surface for this: `defaultOptions` on `QueryClient`. Use it to encode global assumptions first, then override locally only when a specific use case genuinely differs. A bootstrap with

opinionated defaults makes the pattern concrete:

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

export const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 30 * 1000,
      gcTime: 10 * 60 * 1000,
      retry: 2,
      refetchOnWindowFocus: false,
    },
    mutations: {
      retry: 0,
    },
  },
})
```

This is not mere convenience. Each value expresses a system-level claim. A `staleTime` of 30 seconds says most reads remain trustworthy for a short window. A `gcTime` of 10 minutes says inactive data is worth keeping around for navigation reuse. Turning off `refetchOnWindowFocus` says you prefer predictable network quiet over aggressive resynchronization on tab return. Setting mutation retries to zero says writes deserve domain-specific handling instead of a blanket retry posture.

Defaults are policy, not decoration

TanStack Query ships with strong defaults. Queries are stale by default. Stale queries may refetch automatically on mount, window refocus, reconnect, or polling intervals when configured. Inactive queries are garbage-collected after the default `gcTime` of five minutes. Failed queries retry three times with exponential backoff by default. Those defaults are reasonable general-purpose behavior, but they are not neutral. They favor eventual synchronization and resilience against transient network faults.

If your team never revisits them, you still have a policy; it is simply the library's default policy. That may be entirely appropriate for some

applications. The important step is to decide deliberately. A monitoring dashboard, a back-office CRUD application, and a medical intake workflow do not share the same tolerance for staleness, extra requests, or optimistic retries. Global defaults give you one place to encode that difference.

Freshness policy starts with `staleTime`

Most global behavior becomes intuitive once you understand `staleTime` as a freshness contract. The default behavior is `staleTime: 0`, which means data becomes stale immediately after it is fetched. That does not force an immediate refetch, but it makes the query eligible for background refetching on mount, focus, reconnect, and other synchronization triggers.

Increasing `staleTime` reduces how often those triggers result in network activity. If you set `staleTime` to 30 seconds, returning to a screen within that window uses the cached result as still fresh. This improves navigation speed and lowers request volume, but it also accepts a bounded amount of staleness. For many internal line-of-business systems, that trade-off is excellent. Users value responsive navigation, and the cost of being 20 seconds behind on a low-volatility list is small.

Two special values deserve careful use. `Infinity` means the query will not become stale automatically; it stays fresh until you invalidate it or replace its data. This works well for reference data that changes rarely and for workflows where explicit invalidation is more trustworthy than background heuristics. `'static'` is stronger. It prevents refetching even when you invalidate manually. That is a deliberate opt-out from normal synchronization and should be reserved for data that is effectively immutable within the runtime, such as fixed build metadata or hardcoded configuration mirrored from a server.

Choosing a global `staleTime` is therefore not about picking a number that “feels fast.” You are declaring how long the organization is willing

to trust cached reads before synchronization pressure should resume.

Retention policy starts with `gcTime`

Freshness and retention are related but distinct. `staleTime` controls when data is considered eligible for background refetch. `gcTime` controls how long inactive query data remains in memory before TanStack Query garbage-collects it. Confusing the two leads to poor policy choices.

Suppose you want fast back-and-forth navigation across a workflow where screens reuse the same entities. A longer `gcTime` can help even if `staleTime` remains short. The data may be stale and due for a background refetch, but it still exists locally, so the screen can render from cache immediately while synchronization happens in the background. That often produces a better user experience than forcing a cold start every time the component remounts.

The opposite mistake is to inflate `gcTime` aggressively without examining memory behavior. If your application touches many large query results during a session, retaining inactive entries for an hour may consume memory far beyond what the navigation benefit justifies. You need to match `gcTime` to the revisit pattern of your users. Frequent tabbing between a handful of screens supports longer retention. Huge fan-out across thousands of distinct entities argues for a shorter window.

Retry policy expresses failure philosophy

Retries are another place where organizational values show up in code. Out of the box, failed queries retry three times with exponential backoff. That default assumes many failures are transient and worth absorbing automatically. It is often a good fit for reads because temporary packet loss or a short-lived backend hiccup should not force a user-visible error immediately.

Global query retry policy should answer two questions. First, how noisy are transient failures in your environment? Second, how expensive are repeat requests to your APIs and downstream systems? If you run across unreliable mobile networks, two or three retries may smooth over many false failures. If your backend enforces strict rate limits during incidents, aggressive retries can worsen the problem. A lower retry count may be more responsible.

Mutations deserve even more caution. A blanket mutation retry policy can be dangerous when writes have side effects that are not naturally idempotent. Creating a record, sending a payment instruction, or triggering an external workflow may not tolerate naive automatic re-execution. That is why many teams set a conservative global mutation retry policy, often zero, and enable retries only for specific mutation classes that are known to be safe.

Refetch triggers define synchronization posture

Staleness alone does not generate network traffic; triggers do. TanStack Query can refetch stale queries automatically when components mount, when the window regains focus, when connectivity returns, and on intervals if you configure them. Those triggers make the runtime feel alive, but they also shape network cost and perceived predictability.

`refetchOnWindowFocus` deserves explicit treatment because it is highly visible to users and developers. In many applications, the default focus-based synchronization is beneficial: users switch back to the tab and see current data without manual refresh. In others, especially heavy enterprise interfaces, focus changes happen constantly as people move between tools, and the extra requests feel noisy. Disable it globally if that noise outweighs the freshness benefit:

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

export const queryClient = new QueryClient({
```

```
    defaultOptions: {
      queries: {
        refetchOnWindowFocus: false,
      },
    },
  },
})
```

You can still override per query where focus-based synchronization is valuable. That asymmetry is often useful: keep the global posture calm, then opt in on high-value views such as incident dashboards or rapidly changing queues.

Custom focus detection for nonstandard hosts

Browser window focus is not the only execution environment. If your application runs in a nonstandard host or a React Native-like container, you may need to define what “focus” means for the runtime. TanStack Query exposes `focusManager.setEventListener` for that purpose:

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

focusManager.setEventListener((handleFocus) => {
  window.addEventListener('visibilitychange', handleFocus)
  window.addEventListener('focus', handleFocus)

  return () => {
    window.removeEventListener(
      'visibilitychange',
      handleFocus
    )
    window.removeEventListener('focus', handleFocus)
  }
})
```

The key design point is not the DOM API itself. It is that focus semantics belong to runtime policy. If your host emits a different notion of foreground activity, wire it in once at bootstrap instead of forcing individual queries to compensate for an environment-wide mismatch.

Connectivity policy and `onlineManager`

Connectivity is another runtime-wide assumption that should not

be left to ad hoc local code. In v5, `onlineManager` starts from `online: true` and updates based on window online/offline events; it no longer relies on `navigator.onLine`. That change matters because `navigator.onLine` can report false negatives in some Chromium-based environments, leading to a runtime that thinks it is offline when the network is usable.

That default improves robustness, but you still need to decide how your application should behave around interrupted connectivity. If your environment has its own reliable connectivity signal, you can customize online detection globally rather than embedding custom checks into query functions. Treat connectivity as infrastructure state, not feature logic.

Network mode changes offline pause and resume behavior

`networkMode` pushes the policy discussion further. TanStack Query supports three network modes, and you can set them globally or per query and mutation. The default is `'online'`. Under that mode, requests and retries respect online state. If the runtime goes offline during retries, retry behavior pauses and resumes later when connectivity returns.

This is especially important for mobile, offline-capable, or service-worker-heavy applications. Some async operations are true network calls and should pause when the network is unavailable. Others are local computations wrapped in promise-based APIs and should not be blocked by offline state. A global `networkMode` establishes the common case, while local overrides handle exceptional operations. Without that split, teams often cargo-cult local retry logic that fights the runtime.

Layering defaults with `setQueryDefaults`

Global policy does not mean one undifferentiated rule for every query. You can layer defaults by query key pattern with `setQueryDefaults`. In

v5, `getQueryDefaults` merges all matching registrations, so ordering now matters. Register defaults from least specific to most specific:

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

const queryClient = new QueryClient()

queryClient.setQueryDefaults(['issues'], {
  staleTime: 30 * 1000,
  gcTime: 10 * 60 * 1000,
})

queryClient.setQueryDefaults(['issues', 'detail'], {
  staleTime: 5 * 60 * 1000,
})

queryClient.setQueryDefaults(['issues', 'detail', 'audit'], {
  staleTime: Infinity,
})
```

With broad-to-specific ordering, the merged result reads naturally. All `issues` queries inherit the base policy. Detail queries refine freshness further. Audit detail queries then override again for a narrow, stable domain. If you register in the opposite order, the merge still happens, but the effective policy can become hard to reason about because later broad defaults may overwrite the assumptions you intended to make more specific.

The same caution applies to mutation defaults. Registration order is part of policy design, not a cosmetic implementation detail.

Keep query shape co-located even with strong globals

A mature global policy does not eliminate the need to keep each query's identity and fetch logic close together. `queryOptions` is the official helper for co-locating `queryKey` and `queryFn`, and tooling reinforces that pattern. This matters because defaults can standardize freshness and retries, but they cannot repair a mismatch between keys and fetch functions. If one team scatters key fragments and another duplicates query functions with slightly different options, the global policy surface

loses much of its value.

The practical rule is straightforward: set broad behavioral policy in the client, keep query identity and fetch definition together near the feature, and reserve local overrides for genuine exceptions. That split gives you both consistency and local clarity.

Choosing defaults by application class

Different systems push you toward different global postures.

A high-synchronization dashboard usually wants a shorter `staleTime`, may keep focus and reconnect refetching enabled, and often accepts the default query retry behavior because freshness matters more than minimizing background traffic. An internal CRUD application usually benefits from a moderate `staleTime`, longer `gcTime`, and a calmer focus policy because users value fast navigation and predictable screens over constant background refresh. A write-heavy workflow often uses conservative mutation retries and explicit invalidation after successful writes because duplicate side effects are more costly than a transient user-visible failure.

Do not search for universally correct numbers. Look for the operational signals: API rate limits, data volatility, tolerance for stale reads, cost of incorrect writes, and the team's ability to observe production behavior. Defaults are durable only when they reflect those realities.

A practical policy workflow

A useful rollout sequence is simple.

- Classify your main query domains by volatility and business risk.
- Choose one global baseline for reads and one for writes.
- Layer narrower defaults only where patterns are stable.
- Keep per-query exceptions rare and justified.

- Inspect actual cache and network behavior before freezing the policy.

That last step matters. Teams often respond to noisy refetching by disabling helpful triggers globally, or they increase retention windows without checking memory growth. Validate the runtime under real navigation, focus changes, and transient network loss. If a policy feels wrong in production, adjust the global assumption before scattering local fixes.

A well-chosen `defaultOptions` block gives your runtime a clear operational stance. Freshness, retries, retention, focus behavior, and connectivity handling stop being incidental library settings and become part of the application’s contract with its users and its backend systems.

3.4. Operating the Runtime with Devtools and Imperative Client APIs

Declarative queries and mutations should handle most application data flow, but you still need a control plane for the moments when you are debugging, preloading, resetting state, or coordinating work outside the normal render cycle. Those moments are where teams either gain a precise mental model of the runtime or start fighting it. If a query appears to refetch “randomly,” or a mutation succeeds yet the UI remains stale, you need tools that expose what the runtime actually believes: whether a query exists, whether it is active, whether it is stale, whether it was invalidated, and whether another client boundary is involved.

A route preload is a good example of imperative control used correctly. Suppose a navigation action should warm the cache before the desti-

nation screen mounts. You can do that with the shared `QueryClient` instead of inventing a separate preload cache:

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

const queryClient = new QueryClient()

async function openIssue(issueId) {
  await queryClient.prefetchQuery({
    queryKey: ['issues', issueId],
    queryFn: () => fetchIssue(issueId),
  })

  navigate('/issues/' + issueId)
}
```

This code is imperative, but it still respects the declarative model. The destination screen can use a normal query hook with the same `queryKey`. You are not bypassing the runtime; you are preparing it. That distinction matters throughout runtime operations: imperative APIs are most effective when they coordinate with the cache and observer model rather than replacing it.

Treat Devtools as the first inspection surface

When behavior looks suspicious, start with Devtools rather than guesses. Devtools are the officially supported inspection surface for query and mutation runtime state. They let you inspect whether a query is present, which key it uses, whether it is active or inactive, how stale it is, when it last updated, and whether invalidation or refetching is occurring. That information is more reliable than reasoning from UI symptoms alone.

The habit to cultivate is simple: before you change retry settings, disable focus refetching, or patch around perceived inconsistency, check whether the runtime state matches your mental model. Many apparent “random refetch” bugs turn out to be one of four concrete causes:

- the query is stale and a documented trigger fired;

- the component remounted under a different provider boundary;
- the key is not what the code author thought it was;
- an invalidation matched more broadly than intended.

Devtools are especially useful because they expose these causes without requiring you to instrument every feature manually. Production telemetry still needs its own pipelines, but for development and local diagnosis, Devtools should be your default starting point.

The imperative surface of `QueryClient`

`QueryClient` exposes a broad imperative API for prefetching, cache reads and writes, invalidation, refetching, cancellation, reset, removal, activity checks, and paused mutation resumption. You should think of these methods as runtime operations, not as an alternative programming style. They are most appropriate in routers, event handlers, session transitions, tests, and infrastructure code.

Several methods form the core operational vocabulary:

- `prefetchQuery` warms the cache without forcing an immediate consumer.
- `fetchQuery` actively fetches and returns data through the query pipeline.
- `ensureQueryData` is useful when you need data present and want the runtime to decide whether cache reuse is sufficient.
- `getQueryData` and `getQueriesData` inspect cached data synchronously.
- `setQueryData` and `setQueriesData` update cached entries directly.

- `invalidateQueries` marks matches invalid and typically causes active queries to refetch in the background.
- `refetchQueries` performs targeted refetching.
- `cancelQueries`, `resetQueries`, `removeQueries`, and `clear support cleanup` and reset workflows.
- `isFetching` and `isMutating` expose current activity counts.
- `resumePausedMutations` resumes mutations paused by offline conditions.

You do not need all of these in everyday feature code. You do need to know they exist so you can solve operational problems with the runtime's own mechanisms instead of layering parallel state machines around it.

Prefetching and ensuring data without abandoning hooks

Imperative preloading is most effective when it aligns with future declarative consumption. `prefetchQuery` is ideal when you want to warm likely-needed data ahead of time, such as on hover, route intent, or optimistic navigation. `fetchQuery` is stronger: it retrieves data through the client and gives you the result immediately. Use that when a non-React caller needs the actual value before proceeding. `ensureQueryData` is useful when some callers want a guarantee that data exists but should still benefit from cache reuse.

The practical difference is about control flow. If navigation can happen whether or not the preload finishes, use `prefetchQuery`. If the next operation depends on the resolved data, use `fetchQuery` or `ensureQueryData`. In all three cases, the query should still be modeled declaratively in the screen that consumes it. The imperative call is setup, not the long-term source of truth.

Invalidation is the primary write-side control lever

After a successful write, most applications need some form of re-synchronization. `invalidateQueries` is the runtime's main control for that. It marks matching queries invalid and, by default, refetches active queries in the background. That default is valuable because it preserves cache identity while telling the runtime that its current data is no longer trustworthy.

A targeted invalidation might look like this in infrastructure or event handling code:

```
await queryClient.invalidateQueries({
  queryKey: ['issues', issueId],
})
```

The power and the danger both depend on query keys. If your keys are well structured, you can invalidate a precise slice of server state. If your keys are sloppy or overly broad, one invalidation can trigger surprisingly large portions of the application. That is why imperative APIs and key design are tightly coupled. Precision in one depends on precision in the other.

You also need to understand the `refetchType` adjustment. By default, invalidation refetches active matches in the background. If you need different behavior, you can change that. For example, you may want to mark queries invalid without immediate refetch, or you may want to include inactive queries in a broader maintenance operation. Those are operational choices, not everyday UI habits.

Prefer invalidation over broad manual cache edits

Direct cache writes with `setQueryData` or `setQueriesData` are sometimes appropriate, especially for applying a known server response or maintaining temporary consistency in a tightly controlled flow. They are not a substitute for server truth. When teams overuse direct writes, they often end up papering over missing invalidation or weak query-key design.

A sound rule is to prefer invalidation when the server is the authority and you simply need the runtime to re-synchronize. Reach for direct cache writes when you already possess the authoritative next value or when the cost of a round trip is unnecessary and the consistency risk is well understood. The farther your local edit drifts from a concrete server response, the more likely it is to create hidden divergence.

Activity counts support global diagnostics

Global loading indicators should not guess from local component state. `queryClient.isFetching()` and `queryClient.isMutating()` return numeric activity counts, which makes them suitable for top-level diagnostics and shell-wide feedback. Even if your UI later uses hooks for reactive rendering, understanding these counts is useful because they tell you how much work the runtime is doing at a given moment.

A minimal operational probe is straightforward:

```
function logRuntimeActivity(queryClient) {
  const fetchCount = queryClient.isFetching()
  const mutationCount = queryClient.isMutating()

  console.log({
    fetchCount,
    mutationCount,
  })
}
```

Counts are more informative than booleans in busy applications. A loading bar that appears while `isFetching() > 0` tells you the runtime is active somewhere. A diagnostics panel that records the actual counts can reveal whether one invalidation triggered one background refresh or twenty. That difference is often the clue you need when tracking accidental fan-out.

Reset, removal, and logout workflows

Session transitions are another place where imperative control matters. A logout flow should not merely navigate away; it should usually stop

in-flight work and remove or reset shared server state that should not bleed into the next authenticated session.

A simple routine can cancel active queries and then clear the client:

```
async function logout(queryClient) {  
  await queryClient.cancelQueries()  
  queryClient.clear()  
  
  navigate('/login')  
}
```

This is intentionally blunt. In some applications, `resetQueries()` or `removeQueries()` targeted to specific keys is safer than `clear()`, especially if a runtime contains public and private data that should be treated differently. The design question is always the same: after the session boundary changes, which cached server state is still valid for the next user context? Whatever the answer is, express it explicitly. Do not rely on incidental remounts to clean up sensitive data.

Cache subscriptions for low-level observability

Devtools are excellent for inspection, but they are not your production observability layer. For that, TanStack Query exposes lower-level hooks through `QueryCache.subscribe` and `MutationCache.subscribe`. These subscriptions let infrastructure code observe runtime events and translate them into telemetry, metrics, or debugging traces.

A sketch of the pattern looks like this:

```
const unsubscribe = queryClient  
  .getQueryCache()  
  .subscribe((event) => {  
    console.log('query-cache-event', event)  
  })
```

Use this layer carefully. It is powerful, but it is lower level than most feature teams need. The best use cases are central logging, runtime analytics, and incident instrumentation where you need to observe

cache transitions across the whole application. Keep the processing lightweight and avoid introducing heavy side effects into subscription callbacks.

The same principle applies to `MutationCache.subscribe`. It is appropriate for must-run operational telemetry, not for ordinary view logic. If a feature requires cache subscriptions just to keep the UI correct, the declarative model is probably being bypassed in the wrong place.

Paused mutations and offline recovery

If your application operates in unreliable network conditions, paused mutations become part of runtime operations. `TanStack Query` can pause retries and resume them later when connectivity returns, depending on network mode and online state. `resumePausedMutations()` gives you an explicit control point for recovery workflows.

This is especially useful in applications that restore a session, recover from offline periods, or reinitialize runtime state after a connectivity transition. Instead of inventing a custom retry queue, you can let the mutation runtime hold paused work and then resume it through the client when the environment is ready.

Persistence and multi-tab coordination

Runtime operations do not stop at one page lifetime. Persist-client tooling exists for cache persistence, which allows query state to survive reloads or process restarts depending on the chosen persister. The official tooling notes that persister writes are throttled to reduce storage cost. That throttling matters operationally because it reminds you that persistence is not a write-through mirror of every transient change; it is a managed persistence layer with its own trade-offs.

Multi-tab synchronization is possible with `broadcastQueryClient`, but it is explicitly experimental and carries breaking-change warnings.

Treat it as an advanced optimization, not a foundation. If you adopt it, do so with the same caution you would apply to any cross-context runtime coordination feature: validate behavior under tab races, stale storage, and partial connectivity.

A disciplined operational flow

When something looks wrong, use a repeatable sequence.

1. Reproduce the behavior under a known provider boundary.
2. Inspect the query or mutation in Devtools.
3. Verify key shape, activity state, and freshness timestamps.
4. Check whether a documented trigger or invalidation fired.
5. Use a targeted client API to test a hypothesis.

That ordering keeps you from reaching for imperative methods too early. For example, if a query stays stale after a mutation, inspect whether the mutation invalidated the correct key before calling `refetchQueries()`. If duplicate requests appear, check whether the screen remounted under a different client before changing retry policy. If a loading bar never disappears, compare `isFetching()` and `isMutating()` counts before blaming one feature's local state.

Common anti-patterns

Several mistakes are worth avoiding.

- *Using imperative APIs as the primary data model.* If screens rely on manual fetch, manual write, and manual refresh calls scattered across event handlers, you lose the benefits of declarative caching and observer-driven updates.

- *Broad invalidation without disciplined keys.* A vague `invalidateQueries` call can create unnecessary network storms.
- *Direct cache edits to hide server inconsistency.* `setQueryData` can make the UI look correct temporarily while the actual server state remains unresolved.
- *Treating Devtools as production telemetry.* Devtools are for inspection, not for long-term operational observability.
- *Ignoring client boundaries during debugging.* Imperative calls operate on one `QueryClient`. If you inspect one runtime and the bug lives in another provider subtree, your interventions will appear to do nothing.

Keep the control plane in service of the declarative model

The runtime is easiest to operate when imperative code remains narrow and purposeful. Use Devtools to inspect, use prefetching to prepare, use invalidation to re-synchronize, use activity counts to observe, and use reset or clear operations at explicit lifecycle boundaries such as logout or test teardown. Let declarative queries and mutations continue to own steady-state behavior.

That division of responsibility keeps your cache coherent and your debugging grounded in actual runtime state. Imperative APIs become tools for precision rather than escape hatches from the model that makes TanStack Query valuable in the first place.

Chapter 4

Queries, Keys, and Cache Identity

Most TanStack Query bugs are not caused by fetching itself, but by subtle mistakes in identity: a query runs too early, two resources accidentally share cache state, or invalidation misses the data the UI actually depends on. This chapter makes those failure modes legible by showing how declarative queries, dependency-controlled execution, and disciplined key design combine into the cache contract that the rest of the library builds on.

4.1. Anatomy of a Declarative Query

A large share of confusion around TanStack Query starts with a simple misread of the hook: you call `useQuery`, data eventually appears, and it is tempting to think of the hook as a fetch button embedded in render. That model breaks as soon as two components “fetch the same thing”

and behave differently. One mounts with cached data and no spinner. Another shows stale data while refetching in the background. A third mounts while offline or disabled and remains idle. If you treat every one of those situations as “the fetch is running” or “the fetch is not running,” your UI logic collapses distinct runtime states into the same branch and subtle bugs follow.

A declarative query is more precise than that. You describe a resource identity and a retrieval function, and TanStack Query manages an observable cache entry around that description. The hook does not mean “go fetch now.” It means “this component observes the state of the resource identified by this key, using this function when data must be acquired or refreshed.”

A minimal query in TanStack Query v5 looks like this:

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

async function fetchTodo(todoId) {
  const response = await fetch(`/api/todos/${todoId}`)

  if (!response.ok) {
    throw new Error('Failed to load todo')
  }

  return response.json()
}

export function TodoDetails({ todoId }) {
  const query = useQuery({
    queryKey: ['todo', todoId],
    queryFn: () => fetchTodo(todoId),
  })

  if (query.status === 'pending') {
    return <p>Loading...</p>
  }

  if (query.status === 'error') {
    return <p>{query.error.message}</p>
  }

  return <h1>{query.data.title}</h1>
```

```
}

```

That example is intentionally small because the contract itself is small. The query is defined by two required pieces:

- `queryKey`: the identity of the resource
- `queryFn`: how to retrieve that resource

Everything TanStack Query does with caching, reuse, background refetching, and invalidation hangs off that contract.

The object form is the v5 baseline

TanStack Query v5 uses the object-based `useQuery({ . . . })` API. If you are returning from older React Query material, this matters because older tuple-style examples and older terminology can blur the current model. In v5, the object form makes the hook read like a declaration of behavior rather than a positional call signature. That improves more than readability. It keeps identity, fetching behavior, retry controls, and state options grouped in one place, which is useful once queries become parameterized and reused.

The minimal valid declaration still stays compact: a unique `queryKey` and a promise-returning `queryFn`. The promise must either resolve data or throw an error. TanStack Query does not require you to adopt a custom transport layer. `fetch`, `Axios`, and `SDK` clients are all acceptable as long as the function returns a promise. The hook does not care how bytes move across the wire; it cares that a stable resource identity maps to an async retrieval operation.

Identity and transport are separate concerns

You get better results from TanStack Query when you separate two questions cleanly:

1. What resource am I talking about?
2. How do I retrieve it?

`queryKey` answers the first. `queryFn` answers the second.

That distinction sounds obvious until code starts drifting. Teams often smuggle parameters into the `fetch` function without reflecting them in the key. For example, a token, locale, page number, or filter value influences the returned data, but the key remains too broad. The result is incorrect cache reuse: TanStack Query sees one identity while your `fetch` function is actually producing multiple result variants.

A common anti-pattern looks like this:

```
function SearchResults({ term }) {
  const query = useQuery({
    queryKey: ['search'],
    queryFn: () => fetch(
      `/api/search?q=${encodeURIComponent(term)}`
    ).then((r) => r.json()),
  })

  // ...
}
```

The network request changes when `term` changes, but the identity does not. You have encoded transport variance without encoding resource variance. From the cache’s point of view, every search term is “the same query.” That is not a refetch bug. It is an identity bug.

The corrected declaration makes the key describe the actual resource variant:

```
function SearchResults({ term }) {
  const query = useQuery({
    queryKey: ['search', { term }],
    queryFn: () => fetch(
      `/api/search?q=${encodeURIComponent(term)}`
    ).then((r) => r.json()),
  })
}
```

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

The retrieval logic may still use `term` directly, but now the identity and the transport inputs agree. That agreement is one of the core correctness rules in TanStack Query: every value that semantically changes the fetched result belongs in the query key.

A query is a cache entry observed by components

It helps to stop thinking of each `useQuery` call as an isolated unit. The hook is better understood as an observer attached to a cache-backed query entry. Multiple components can observe the same entry. If they declare the same key, they are not creating unrelated fetch jobs. They are subscribing to shared server-state.

This is why different components can render different UI states against the same underlying query. One observer may mount when no data exists yet and show a hard loading state. Another observer may mount later, find cached data immediately, and render content while a background refetch occurs. A third observer may mount after an error and render the error branch. The query entry is shared; the observer's moment in time and rendering policy are not.

That observer model also explains why mounting a component does not guarantee a network request. If suitable cached data already exists, TanStack Query can serve it immediately. Whether it also refetches depends on freshness rules and runtime triggers, which are worth treating separately from the core query contract. For now, the operational point is narrower: `useQuery` subscribes to query state, and network activity is a consequence of that state plus query policy, not the sole purpose of the hook.

Lifecycle state is two-dimensional

TanStack Query exposes several state fields, but two are foundational

for reasoning accurately:

- `status`
- `fetchStatus`

You need both because “do I have usable data?” and “is a fetch currently in flight?” are different questions.

`status` describes the logical resolution state of the query result. In practice, you will most often reason about:

- `'pending'`: the query has not successfully resolved data
- `'error'`: the last attempt ended in error
- `'success'`: the query has data

`fetchStatus` describes current execution activity:

- `'idle'`: no fetch is running now
- `'fetching'`: a fetch is running now

A dependency-gated or paused situation can also produce a non-fetching state even while the query has not resolved data yet, which is exactly why one boolean rarely tells the whole story.

The critical distinction is that `pending` does not mean “currently fetching.” A query can be `status === 'pending'` and `fetchStatus === 'idle'` if it has not run yet or is currently disabled by configuration. Likewise, a query can be `status === 'success'` and `fetchStatus === 'fetching'` when cached data is present and a background refetch is underway.

Those combinations lead to different UX decisions:

- Pending + fetching: first acquisition is happening now. A hard loading placeholder is often appropriate.
- Success + fetching: usable data exists and a refresh is happening. Preserve the existing view and show a softer refresh indicator if needed.
- Pending + idle: the query exists, but execution is not currently happening. Treating this as “loading” can be misleading if the real cause is missing prerequisites.
- Error + idle: the attempt failed and no fetch is active now. An error branch is usually correct, possibly with a retry affordance.

Why `isFetching` is not enough

Many UI bugs come from treating `isFetching` as a universal loading signal. It is useful, but it answers only one narrow question: is the query fetching right now? It does not tell you whether you already have data, whether the query has ever succeeded, or whether the current state reflects first load versus background synchronization.

If you write logic like this:

```
if (query.isFetching) {  
  return <Spinner />  
}
```

you will hide valid cached content during background refetches. The user sees a full-screen loading state even though the application already has data. That is usually a regression, not a simplification.

A safer pattern is to branch first on whether data acquisition has ever completed successfully, then layer fetch activity on top:

```
if (query.status === 'pending') {  
  return <Spinner />  
}
```

```
if (query.status === 'error') {  
  return <ErrorView message={query.error.message} />  
}  
  
return (  
  <>  
    {query.fetchStatus === 'fetching' ? <small>Refreshing...</small>  
    : null}  
    <TodoView todo={query.data} />  
  </>  
)
```

This keeps the first-load experience distinct from a background refresh. The user continues to see meaningful content while synchronization happens.

Observer-driven execution explains shared behavior

Because components observe query entries, TanStack Query can coordinate work across consumers. If two components declare the same query while one fetch is already running, the second observer does not need to launch an unrelated duplicate request just because it mounted later. Both observers can receive updates from the same cache entry as the query resolves.

This model also explains why a query can continue to exist meaningfully even when one component unmounts. The cache entry is not the same thing as a React component instance. React controls observers. TanStack Query controls query entries and their lifecycle in the cache. That separation is what makes reuse possible across route changes, tab switches, and repeated mounts within the cache retention window.

You do not need the full cache lifecycle model yet, but you do need this boundary: component render creates or attaches an observer; the observer points at a query entry; the query entry owns data state, fetch state, timestamps, and policy-driven behavior.

Use stable declarations, not accidental closures

Another recurring anti-pattern is writing a fetch function that closes over values in ways the key does not express. The issue is not that inline arrow functions are forbidden. Inline functions are common and often correct. The issue is whether the function's behavior varies with values that the key omits.

This version is coherent:

```
function UserPanel({ userId }) {
  return useQuery({
    queryKey: ['user', userId],
    queryFn: () => fetch(`/api/users/${userId}`)
      .then((r) => {
        if (!r.ok) throw new Error('Request failed')
        return r.json()
      })
  })
}
```

The closure uses `userId`, and the key also includes `userId`. Identity and retrieval stay aligned.

This version is not coherent if `locale` changes the payload:

```
function ProductPanel({ productId, locale }) {
  return useQuery({
    queryKey: ['product', productId],
    queryFn: () => fetch(
      `/api/products/${productId}?locale=${locale}`
    ).then((r) => r.json()),
  })
}
```

If the locale alters the server response, the key is incomplete. You may see stale or cross-locale data reuse even though the fetch function itself is technically valid.

Reusable query declarations with `queryOptions`

As queries spread across views, prefetch paths, and tests, repeating key and fetch logic becomes a source of drift. TanStack Query provides `queryOptions` to package a canonical query declaration for

reuse across APIs such as `useQuery` and prefetching methods on `QueryClient`.

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

function todoOptions(todoId) {
  return queryOptions({
    queryKey: ['todo', todoId],
    queryFn: async () => {
      const response = await fetch(`/api/todos/${todoId}`)

      if (!response.ok) {
        throw new Error('Failed to load todo')
      }

      return response.json()
    },
  })
}

export function TodoDetails({ todoId }) {
  const query = useQuery(todoOptions(todoId))

  if (query.status === 'pending') {
    return <p>Loading...</p>
  }

  if (query.status === 'error') {
    return <p>{query.error.message}</p>
  }

  return <h1>{query.data.title}</h1>
}
```

This pattern becomes more valuable once you reuse the same query declaration for route prefetching, server rendering, or direct cache reads. It also enforces a healthy habit: treat the query specification as a first-class object rather than scattering identity and fetch logic across unrelated call sites.

Do not assume mount implies network I/O

One of the most persistent misconceptions is “the component mounted, so TanStack Query fetched.” Sometimes it did. Sometimes it attached an observer to fresh cached data and did nothing else. Sometimes it attached to stale cached data and started a background refetch. Sometimes it remained idle because execution was disabled. Sometimes a fetch was already in progress due to another observer.

That is why event-style language such as “I triggered a query by rendering the component” is only partially accurate. Rendering expresses interest in a resource. The runtime then decides whether acquisition or refetch is necessary under the query’s policy and current cache state.

Freshness windows, focus-driven refetching, reconnect behavior, and garbage collection determine many of those decisions, but those mechanisms are easier to understand once the basic contract is stable in your mind. A query is identified, observed, and conditionally executed. It is not an imperative side effect disguised as a hook.

Map UI branches to semantics, not booleans

When you design UI around TanStack Query, branch on what the user actually needs to know:

- Is there usable data yet?
- Is the system currently trying to acquire or refresh it?
- Did the last attempt fail?
- Is execution unavailable because required inputs are missing?

Those questions map more directly to `status`, `fetchStatus`, and the presence of data than a single loading boolean ever can. That discipline pays off immediately. You stop erasing content during background refetches, stop displaying spinners for idle-but-disabled queries, and stop assuming every render cycle corresponds to a new request.

Once you treat `useQuery` as a declarative contract over a cache entry, the rest of TanStack Query becomes far easier to reason about. `queryKey` tells the system what resource identity you mean. `queryFn` tells it how to retrieve that resource when needed. The hook returns observable state that separates logical resolution from current fetch activity. Components do not own isolated requests; they observe shared server-state entries whose behavior depends on identity, policy, and runtime conditions. That vocabulary is what makes later decisions about key design and cache targeting feel mechanical rather than mysterious.

4.2. Modeling Dependent Reads with Enabled Queries

Screens rarely need just one resource. A dashboard might need session data before it can identify the current user, then the user before it can load organization settings or feature flags. Many teams solve that with chained `useEffect` calls, local “ready” booleans, and imperative guards around network functions. The code works until timing changes: route params arrive one render late, session hydration lags behind the first paint, or a component remounts and duplicate requests race each other. TanStack Query gives you a cleaner control surface. Instead of choreographing fetch timing manually, you declare when a query is valid to run.

A dependent read is still a query. It still needs a real `queryKey` and a real `queryFn`. The difference is that execution is gated by prerequisite data. In TanStack Query v5, the standard gate is `enabled`.

A concrete serial read looks like this:

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

async function getUserByEmail(email) {
```

```
const response = await fetch(
  `/api/users?email=${encodeURIComponent(email)}`
)

if (!response.ok) {
  throw new Error('Failed to load user')
}

return response.json()
}

async function getProjectsByUser(userId) {
  const response = await fetch(`/api/projects?userId=${userId}`)

  if (!response.ok) {
    throw new Error('Failed to load projects')
  }

  return response.json()
}

export function ProjectsScreen({ email }) {
  const userQuery = useQuery({
    queryKey: ['user', email],
    queryFn: () => getUserByEmail(email),
  })

  const userId = userQuery.data?.id

  const projectsQuery = useQuery({
    queryKey: ['projects', userId],
    queryFn: () => getProjectsByUser(userId),
    enabled: !!userId,
  })

  if (userQuery.status === 'pending') {
    return <p>Loading user...</p>
  }

  if (userQuery.status === 'error') {
    return <p>{userQuery.error.message}</p>
  }

  if (projectsQuery.status === 'pending' &&
    projectsQuery.fetchStatus === 'idle') {
    return <p>Waiting for user identity...</p>
  }
}
```

```
if (projectsQuery.status === 'pending') {
  return <p>Loading projects...</p>
}

if (projectsQuery.status === 'error') {
  return <p>{projectsQuery.error.message}</p>
}

return (
  <ul>
    {projectsQuery.data.map((project) => (
      <li key={project.id}>{project.name}</li>
    ))}
  </ul>
)
```

That code does not manually sequence requests. You do not call the second query from an effect after the first resolves. You declare two resources and a dependency boundary: the projects query is only eligible to execute when `userId` exists.

“Conditional fetching” is an imprecise mental model

Developers often describe this pattern as conditional fetching. The phrase is common, but it pushes your attention toward events instead of validity. The more useful question is not “should I fetch now?” It is “do I currently have enough information to describe a valid resource?”

That shift matters because some missing values are not business data; they are incomplete input. If a route parameter is still undefined on the first render, you do not have an empty result. You do not even have a valid query target yet. Likewise, if a user-scoped endpoint requires a session-derived tenant ID, firing a request with no tenant is not eager behavior; it is a malformed read.

enabled models this directly. When `enabled` is `false`, `TanStack Query` does not auto-fetch on mount, does not background-refetch, and ignores `invalidateQueries/refetchQueries`-driven refetches while

disabled. The query still exists as part of your component logic, but execution is withheld until prerequisites are satisfied.

State transitions for dependency-gated queries

A gated query has an observable lifecycle even before it runs. That is useful because the UI often needs to distinguish “not ready yet” from “actively loading.”

For a dependent query with no cached data, the documented lifecycle is:

- while disabled: `status === 'pending'` and `fetchStatus === 'idle'`
- once enabled: `fetchStatus` becomes `'fetching'`
- after success: `status === 'success'` and `fetchStatus === 'idle'`

This is one of the most practical reasons to track both fields. A disabled downstream query can still be `pending`, but it is not `fetching`. If you render every `pending` state as a spinner, you misrepresent the problem. The screen is not blocked by network activity; it is waiting for prerequisites.

That difference becomes especially valuable in multistage UIs. You can choose to show:

- a user-loading state while the first query resolves,
- a neutral placeholder while the second query is known but not yet runnable,
- and a projects-loading indicator only when the second query has actually started.

Those are materially different user experiences, and TanStack Query gives you the state to encode them without inventing parallel booleans.

Dependency edges do not replace cache identity

`enabled` controls execution, not identity. This is the most common conceptual mistake in dependent reads.

Suppose a `projects` query depends on `userId`. If you omit `userId` from the key and rely on `enabled` alone, you have delayed execution but not identified the resource variant correctly. The key still has to fully describe the data returned by the query function.

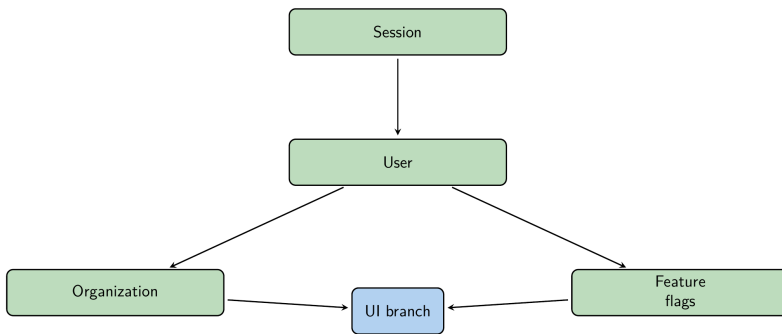
This is wrong:

```
const projectsQuery = useQuery({
  queryKey: ['projects'],
  queryFn: () => getProjectsByUser(userId),
  enabled: !!userId,
})
```

This is correct:

```
const projectsQuery = useQuery({
  queryKey: ['projects', userId],
  queryFn: () => getProjectsByUser(userId),
  enabled: !!userId,
})
```

The gate says when the query may run. The key says what query it is. Those jobs remain separate even when the dependency is simple.



Route parameters and optional inputs

A common dependent-read scenario is a route parameter that is undefined on the first render or temporarily absent during navigation transitions. Without a guard, your query function may issue requests like `/api/orders/undefined`. With a manual effect chain, you might scatter checks across lifecycle code. `enabled` keeps the logic adjacent to the query declaration.

```
function OrderDetails({ orderId }) {
  const orderQuery = useQuery({
    queryKey: ['order', orderId],
    queryFn: async () => {
      const response = await fetch(`/api/orders/${orderId}`)
      if (!response.ok) {
        throw new Error('Failed to load order')
      }
      return response.json()
    },
    enabled: !!orderId,
  })

  if (orderQuery.status === 'pending' &&
    orderQuery.fetchStatus === 'idle') {
    return <p>Select an order.</p>
  }

  if (orderQuery.status === 'pending') {
    return <p>Loading order...</p>
  }
}
```



```
    if (orderQuery.status === 'error') {  
      return <p>{orderQuery.error.message}</p>  
    }  
  
    return <OrderView order={orderQuery.data} />  
  }  
}
```

Notice the distinction between absent input and empty data. If no order is selected, the right UI is usually not “loading.” It is a selection prompt or empty-state instruction. If an order is selected and the query begins fetching, then a loading indicator becomes accurate. Encoding both cases under the same spinner hides a real state transition.

Authenticated resources and staged availability

Authenticated applications add another dependency layer: session materialization. A request for organization settings may require both an authenticated session and a stable organization ID. If you already have a session hook or user query, express the downstream gate in terms of the value that makes the resource valid.

For example, if feature flags are scoped by organization, the meaningful prerequisite is not merely “user exists” but “organization ID is known.” That might come directly from the session, from a user profile query, or from an organization selection control. The shape of the dependency chain matters less than the rule: gate on the smallest complete input that makes the resource valid.

This keeps you out of a frequent trap where downstream queries fire too early because an upstream object exists, even though the specific field needed for the request is still missing or stale. You want gates that represent semantic readiness, not incidental object presence.

Using `enabled` for lazy reads without abandoning declarative behavior

There is a legitimate lazy-read pattern where the user must supply input before any request makes sense. A search box is the classic example. You do not want to fetch results for an empty string on mount, but you still want the query to behave declaratively once a meaningful filter exists.

```
function SearchPanel({ filter }) {
  const resultsQuery = useQuery({
    queryKey: ['search', { filter }],
    queryFn: async () => {
      const response = await fetch(
        `/api/search?q=${encodeURIComponent(filter)}`
      )

      if (!response.ok) {
        throw new Error('Search failed')
      }

      return response.json()
    },
    enabled: !!filter,
  })

  if (!filter) {
    return <p>Enter a search term.</p>
  }

  if (resultsQuery.status === 'pending') {
    return <p>Searching...</p>
  }

  if (resultsQuery.status === 'error') {
    return <p>{resultsQuery.error.message}</p>
  }

  return <SearchResults items={resultsQuery.data} />
}
```

This remains declarative because the query becomes active as a function of data availability, not because you manually tell it when to fire after every keystroke or button click. Once `filter` is truthy, the query participates normally in caching and synchronization for that key.

Why permanently disabled queries are usually the wrong ab-

straction

Official guidance distinguishes lazy or dependent queries from permanently disabled queries. A permanently disabled query is generally non-idiomatic because it opts out of the declarative behaviors that make TanStack Query useful. When `enabled` is always `false`, the query does not auto-fetch on mount, does not background-refetch, and ignores invalidation-triggered refetches while disabled. You have effectively reduced a declarative read model to a manual command holder.

That anti-pattern often appears in code like this:

```
function ReportsPanel() {
  const reportQuery = useQuery({
    queryKey: ['reports'],
    queryFn: loadReports,
    enabled: false,
  })

  return (
    <>
      <button onClick={() => reportQuery.refetch()}>
        Load reports
      </button>
      {/* render reportQuery state */}
    </>
  )
}
```

You can make this work, but you give up normal reactive behavior. The query will not participate in background synchronization while disabled, and invalidation from elsewhere in the application will not refetch it. If the real requirement is “this query is invalid until the user has chosen parameters,” use those parameters as the gate. If the real requirement is a user-triggered command whose timing is not modeled by data dependencies, you may need a different interaction pattern rather than a permanently disabled query masquerading as one.

Type-safe disabling with `skipToken`

TypeScript users have an additional tool: `skipToken` from `@tanstack/react-query`. It provides a type-safe way to disable a query when no valid `queryFn` can be supplied. This is useful when the function cannot even be constructed safely until required input exists.

Use it when the absence of input means “there is no valid query function to install yet,” not merely “delay execution for a moment.” The caveat is important: `refetch()` does not work with `skipToken` because no `queryFn` is installed. If you need manual triggering, `enabled:false` is the supported path. If you need dependency-driven eligibility, `skipToken` can give you better type discipline than inventing non-null assertions around incomplete values.

Avoid hidden dependency loops

As query logic moves into custom hooks, dependency chains can become harder to see. A hook that loads a user, derives an organization ID, and then loads organization settings may be perfectly reasonable. Problems start when downstream queries indirectly influence the upstream values that gate them, or when hooks encode fallback behavior that silently changes keys and gates together.

Watch for these signals:

- a custom hook derives a prerequisite from data that itself depends on the downstream query,
- an “optional” parameter is coerced into a default value that creates an unintended key,
- the UI cannot distinguish invalid input from empty data because the hook flattens both into one boolean.

If a dependency chain feels hard to explain aloud, inspect each query separately: what is the resource, what values make it valid, what key

represents it, and what specific value unlocks `enabled`? Most read orchestration bugs reduce to one of those questions being answered ambiguously.

Operational guidance for dependent reads

You can keep dependent query design disciplined with a short checklist:

1. Identify the prerequisite value that makes the downstream resource valid.
2. Put every result-shaping variable into the downstream `queryKey`.
3. Use `enabled` to gate execution until those inputs exist.
4. Render `pending + idle` differently from `pending + fetching` when the distinction matters.
5. Avoid permanent disablement unless you genuinely want to opt out of normal refetch behavior.

That approach replaces timing choreography with data dependency declarations. Queries run when their inputs define a real resource, stay dormant when they do not, and expose enough state for you to reflect each phase honestly in the UI. Once you treat prerequisites as part of query validity rather than component control flow, even multi-step screens become easier to reason about, because each read answers a narrow question: do I know enough to identify this resource, and if so, should the cache start observing it now?

4.3. Designing Query Keys for Correctness and Scale

You can get surprisingly far with TanStack Query while treating keys as a bit of syntax attached to `useQuery`. That approach usually survives until the application has multiple screens, filtered lists, detail views, and a few months of accumulated features. Then the same underlying problems start surfacing under different names: stale reads, cross-screen data leakage, blunt invalidation, duplicate requests for semantically identical resources, and cache entries that no one can confidently describe. At that point, query keys stop being a convenience and become architecture.

A query key is the cache identity for a server-state read. If the key is wrong, everything built on top of that identity is unstable. If the key is disciplined, most runtime behavior becomes predictable. The design problem is not merely “what string should I put here?” It is “how do I encode resource identity so the cache can distinguish what must be separate and reuse what should be shared?”

A compact but realistic example helps anchor the rules:

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

const todoKeys = {
  all: ['todos'],
  lists: () => ['todos', 'list'],
  list: (filters) => ['todos', 'list', filters],
  details: () => ['todos', 'detail'],
  detail: (id) => ['todos', 'detail', id],
  infiniteList: (filters) =>
    ['todos', 'infinite-list', filters],
}

function todoListOptions(filters) {
  return queryOptions({
    queryKey: todoKeys.list(filters),
```

```

    queryFn: async () => {
      const params = new URLSearchParams({
        status: filters.status,
        page: String(filters.page),
      })

      const response = await fetch(`/api/todos?${params}`)

      if (!response.ok) {
        throw new Error('Failed to load todos')
      }

      return response.json()
    },
  })
}

export function TodoList({ status, page }) {
  const filters = { status, page }
  const query = useQuery(todoListOptions(filters))

  if (query.status === 'pending') {
    return <p>Loading...</p>
  }

  if (query.status === 'error') {
    return <p>{query.error.message}</p>
  }

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

```

That snippet shows several design choices that matter at scale:

- keys are arrays, not ad hoc strings,
- resource families use a readable hierarchy,
- every variable that changes the result is in the key,

- detail, list, and infinite variants have distinct identities,
- `queryOptions` packages the declaration for reuse.

Top-level arrays are the contract

TanStack Query requires the top-level `queryKey` to be an array. That is not a stylistic suggestion. It is the foundation of how keys are structured, hashed, and matched. If you are returning from older React Query habits or examples that overemphasized strings, update your mental model: the array is the canonical form.

The array shape gives you a useful composition vocabulary. A key can start with a broad resource label and grow more specific as you add scope:

```
['todos']  
['todos', 'detail', 42]  
['todos', 'list', { status: 'open', page: 2 }]
```

That progression does two things at once. It makes identity readable to humans, and it creates a structure that later cache operations can target with meaningful granularity. You are not just naming queries; you are building an address space.

Put every result-shaping variable into the key

The core correctness rule is simple: any variable used in `queryFn` that can change and affect the result should be included in `queryKey`. If you violate that rule, you get unintended cache sharing and stale reads. If you overreact by adding unstable, irrelevant values, you get cache fragmentation and unnecessary misses.

Consider a filtered list:

```
useQuery({  
  queryKey: ['todos', { status, page }],  
  queryFn: () => fetchTodos({ status, page }),  
})
```


This is coherent because `status` and `page` both shape the server response and both appear in the key.

This is not coherent:

```
useQuery({
  queryKey: ['todos'],
  queryFn: () => fetchTodos({ status, page }),
})
```

Now every list variant shares one identity. The cache cannot distinguish page 1 from page 2 or open items from completed items. The bug may appear only under navigation or tabbing behavior, but the root cause is immediate: the resource identity is incomplete.

The opposite failure is also common. Developers include values that do not define the resource itself, such as local sort order applied after fetch, modal open state, or a freshly created object with incidental fields. The result is needless key churn. The cache sees many unique resources where the server sees one.

A useful test is direct and pragmatic: if the server response would be different, put the variable in the key. If the variable only changes presentation after the data is already in the client, keep it out.

Objects are usually better than long positional arrays

TanStack Query supports nested objects inside query keys as long as they are serializable with `JSON.stringify`. This is more than a convenience. It gives you room to encode parameter sets without turning the key into a brittle positional puzzle.

For example, these two keys are equivalent because object property order does not affect identity:

```
['todos', { status, page }]
['todos', { page, status }]
```

That deterministic object hashing is extremely useful for maintainabil-

ity. You can add or reorder properties in the object literal without accidentally changing identity, as long as the semantic content is the same.

By contrast, array element order does affect identity. These are not equivalent:

```
['todos', status, page]
['todos', page, status]
```

This is why object segments often scale better than deeply positional keys. A long positional array forces everyone on the team to remember slot semantics exactly. A keyed object makes the meaning visible at the call site and more resilient to later extension.

That does not mean objects are always superior. Primitive segments are excellent for obvious path-like scopes such as resource label, variant, or primary ID:

```
['users', 'detail', userId]
['projects', 'list', { ownerId, status }]
```

A good rule is to use array segments for hierarchy and object segments for parameter bundles.

Design keys as a hierarchy, not a pile of values

Key design becomes much easier once you think hierarchically. Most domains have a small number of natural layers:

- the resource family,
- the variant or scope,
- the identifying primitive or parameter object.

For a posts feature, that might look like this:

```
['posts']
['posts', 'list']
['posts', 'list', { authorId, tag, page }]
```

```
['posts', 'detail', postId]
['posts', 'comments', postId]
```

The important part is not the exact labels. It is the intentional shape. Keys that share a prefix should do so because they belong to the same resource family or operational grouping, not because similar names happened to be chosen casually.

This is where scale pressure becomes visible. A flat scheme like `['posts-list']`, `['post-detail', id]`, and `['comments-for-post', id]` may work for individual queries, but it gives you little structural leverage later. A clean hierarchy lets you group related resources without conflating distinct ones.

Do not mix normal and infinite queries under one key

One of the most important practical warnings is to avoid reusing the same key for `useQuery` and `useInfiniteQuery`. Both interact with the same query cache, but the stored data shapes are different. A normal list query usually holds one result payload. An infinite query holds paginated page data. If both use the same identity, the cache will be asked to store incompatible structures under one address.

This is why the earlier example explicitly separated `list` from `infiniteList`:

```
const todoKeys = {
  list: (filters) => ['todos', 'list', filters],
  infiniteList: (filters) =>
    ['todos', 'infinite-list', filters],
}
```

The extra label may feel redundant at first, but it prevents a class of collisions that are hard to diagnose once the codebase grows. Identity should reflect not only the semantic resource but also the shape expectations of the consumers attached to that identity.

Use key factories, but keep them close to the feature

Hand-writing keys inline everywhere invites drift. One component uses `['todo', id]`; another uses `['todos', id]`; a third adds an extra string segment. All three may refer to the same resource conceptually, yet they occupy different cache entries. You can avoid much of that by using key factories and `queryOptions`.

`queryOptions` provides an official way to package `queryKey`, `queryFn`, and other query defaults together without losing the ergonomics of direct hook usage. Combined with a small feature-local key object, it creates a reliable source of truth.

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

export const projectKeys = {
  all: ['projects'],
  lists: () => ['projects', 'list'],
  list: (filters) => ['projects', 'list', filters],
  details: () => ['projects', 'detail'],
  detail: (id) => ['projects', 'detail', id],
}

export function projectDetailOptions(id) {
  return queryOptions({
    queryKey: projectKeys.detail(id),
    queryFn: async () => {
      const response = await fetch(`/api/projects/${id}`)

      if (!response.ok) {
        throw new Error('Failed to load project')
      }

      return response.json()
    },
  })
}
```

A practical convention is to colocate these factories near the feature rather than collecting every key in one global file. Feature colocation keeps the identity vocabulary close to the domain logic that owns it. A single global registry often starts tidy and ends up as a sprawling dependency hub where unrelated modules edit one another's cache addresses.

Choose names that expose semantics

Key labels are not merely implementation details. They become part of how you reason about the cache. Prefer names that reveal what kind of data a key holds and how narrowly it is scoped.

Compare these two families:

```
['data', 'item', id]
['data', 'items', filters]
```

versus

```
['users', 'detail', id]
['users', 'list', filters]
```

The second family gives you immediate semantic clarity. You can infer collection versus detail, and you can anticipate how related operations should group. That clarity becomes operationally important when you inspect cache state, debug invalidation scope, or audit overlapping reads.

Readable naming also reduces the temptation to over-generalize factories. A hyper-generic helper that builds arbitrary keys from string fragments may save a few lines at first, but it usually hides domain meaning and weakens consistency. Key factories should constrain choices, not erase them.

Common anti-patterns and their failure modes

A few mistakes recur often enough to deserve explicit rejection.

Omitting a dependency from the key. This causes accidental reuse across distinct resource variants. Symptoms include stale screens, mismatched filters, and detail views showing the wrong entity after navigation.

Embedding unstable values. Freshly created objects can be fine if their serialized content is stable and intentional. Functions, class in-

stances, and other non-serializable or semantically noisy values are not. They either fail the serializability expectation or create hard-to-reason-about identity churn.

Mixing transport metadata with identity. Retry counters, client timestamps, or request tracing IDs usually do not define the resource. If you include them, the cache treats every request attempt as a new identity and reuse collapses.

Flattening unrelated variants into one namespace. If list data, detail data, and infinite data all live under keys with no structural distinction, later targeting becomes broad and error-prone.

Over-centralizing all keys. Centralization promises consistency but often delivers indirection. Teams stop understanding what a key means locally because the naming contract lives far away from the query declaration.

Evaluate keys with three questions

A well-designed key answers three practical questions immediately.

1. What resource does this denote?
2. What variables distinguish one instance from another?
3. What family or prefix should related operations target later?

If you cannot answer the first question, the key is semantically vague. If you cannot answer the second, the key is likely incomplete or over-stuffed. If you cannot answer the third, the hierarchy probably lacks structure.

Apply that test to a candidate key such as:

```
[ 'orders', 'list', { customerId, status, page } ]
```

It clearly denotes an orders list, distinguishes instances by customer, status, and page, and sits inside a resource family that can later support broader operations on ['orders'] or narrower ones on ['orders', 'list']. That is the kind of legibility you want.

Design for the team you will become

Small applications can survive inconsistent key habits because one or two developers still remember every implicit rule. Larger applications cannot. Query keys eventually become shared infrastructure, even when no one names them that way. New contributors need to infer how to add a filtered variant without breaking reuse. Reviewers need to spot missing dependencies quickly. Debuggers need to inspect a cache entry and understand why it exists. Operations like invalidation, direct cache updates, and prefetching all become easier or harder depending on whether the key topology was designed deliberately.

A disciplined scheme does not need to be elaborate. It needs to be explicit. Use top-level arrays. Encode every result-shaping variable. Prefer hierarchical labels with object segments for parameter bundles. Keep normal and infinite queries distinct. Package canonical declarations with `queryOptions`. Colocate feature-level factories where the meaning stays visible. Once query identity is treated as a stable contract rather than inline syntax, the cache stops feeling like a bag of incidental entries and starts behaving like a system you can reason about confidently.

4.4. Operating Cache Reuse and Refetch Boundaries Through Key Design

A cache key is not just a label for a query. It defines who shares data with whom, which views update together, and what future cache operations can reach. That becomes visible the moment two screens appear

to agree until one of them changes route parameters, remounts from cache, or reacts to a write. If one screen updates and another stays stale, the cause is often not a broken refetch policy. More often, the key topology made the cache either too broad or too narrow for the behavior you expected.

A small `QueryClient` example makes the control surface concrete:

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

export const queryClient = new QueryClient()

const postKeys = {
  all: ['posts'],
  lists: () => ['posts', 'list'],
  list: (filters) => ['posts', 'list', filters],
  details: () => ['posts', 'detail'],
  detail: (id) => ['posts', 'detail', id],
}

export function postDetailOptions(id) {
  return queryOptions({
    queryKey: postKeys.detail(id),
    queryFn: async () => {
      const response = await fetch(`/api/posts/${id}`)

      if (!response.ok) {
        throw new Error('Failed to load post')
      }

      return response.json()
    },
  })
}

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

queryClient.setQueryData(
  postKeys.detail(42),
  { id: 42, title: 'Draft title' }
)
```


Two runtime facts are packed into that short snippet. First, `invalidateQueries` can match by key prefix, so a shared prefix like `['posts']` becomes the control plane for refetch scope. Second, `setQueryData` is synchronous and can create a cache entry even if none exists yet, which means a wrong key can manufacture phantom state immediately. Once you see keys as addresses into a live cache, design choices that looked cosmetic become operational.

Cache reuse is identity reuse

When two queries have the same effective identity, they share one cache entry. That means they can share data, fetch state, timestamps, and the effects of later invalidation. This is the desirable case when two surfaces truly represent the same resource. A post header widget and a post detail page should often observe the same detail key if they need the same payload. Reuse reduces redundant network work and makes the UI feel coherent.

The failure mode is accidental reuse. Suppose you key a filtered collection too broadly:

```
['posts', 'list']
```

and then let the `queryFn` vary by author, tag, or page. Every list variant now points to one cache address. Navigating from “all posts” to “posts by author 7” can briefly or persistently show the wrong dataset because the cache is being asked to treat distinct resources as one. The UI may appear correct in isolated tests because each screen works when loaded fresh. The bug emerges under reuse, not under initial load.

That is why the practical question is not “can these queries share data?” but “should these resources be considered the same cache identity?” The answer depends on whether the server can return materially different payloads for the input variation. If yes, they need distinct keys even if the UI layout looks similar.

Segmentation is a correctness mechanism

Good key design intentionally separates semantically adjacent resources. A summary list item and a full detail object may refer to the same entity but not the same representation. A global leaderboard and a user-scoped leaderboard may both contain scores but differ in authorization, filtering, and invalidation needs. A collection without filters and a collection with filters can share a family prefix while still occupying separate entries.

A disciplined schema might distinguish these cases:

```
['posts', 'list', { tag, page }]
['posts', 'detail', postId]
['leaderboard', 'global']
['leaderboard', 'user', userId]
```

A sloppy schema often compresses them into something like:

```
['data', id]
```

That overloaded key family forces unrelated resources to compete for one namespace. It may look concise, but it destroys operational precision. You cannot reliably reason about what a prefix means because the prefix no longer denotes a coherent domain. Broad operations become dangerous, narrow operations become guesswork, and direct cache writes become error-prone.

The right amount of segmentation is not maximal fragmentation. You want distinct keys where the response semantics differ and shared prefixes where future operations should intentionally group related reads.

Prefix matching turns hierarchy into an operations API

Once your keys are hierarchical, `QueryClient` methods can exploit that structure. `queryClient.invalidateQueries` matches queries by key or filter. If your post-related queries all live under `['posts', ...]`, you can invalidate the entire family with a prefix match:

```
queryClient.invalidateQueries({
  queryKey: ['posts'],
})
```

That operation marks matching queries stale and can refetch them according to `refetchType`. The official `refetchType` values are `'active'`, `'inactive'`, `'all'`, and `'none'`. This matters because invalidation is not only about data correctness; it is also about controlling where network work happens.

For example:

```
queryClient.invalidateQueries({
  queryKey: ['posts'],
  refetchType: 'active',
})
```

targets all post-related queries but refetches only those currently observed. That is often a good default after a write because visible screens refresh without waking every inactive variant in the cache.

A narrower invalidation remains possible when the hierarchy supports it:

```
queryClient.invalidateQueries({
  queryKey: ['posts', 'detail', 42],
})
```

This precision is impossible if your schema never separated detail from list or never grouped posts under a stable prefix. Key hierarchy is therefore not just a naming convention. It is the index that later operations search.

Invalidation is not deletion

A useful distinction gets lost surprisingly often: `invalidateQueries` keeps matching queries in the cache, while `removeQueries` removes them. If you conflate those operations, your mental model of reuse

becomes unstable.

Invalidation says, in effect, “this cached data may no longer be trustworthy; treat it as stale and refetch according to policy.” Deletion says “discard the cache entry entirely.” The difference matters because invalidated data can still support immediate rendering until refetch completes, while removed data cannot. If you want continuity across screens, invalidation is often the right tool. If you need to eliminate sensitive or obsolete entries altogether, removal is the stronger boundary.

This distinction also clarifies why broad prefixes are powerful but dangerous. Invalidating `['posts']` can be a reasonable way to resynchronize a domain after a write. Removing `['posts']` would throw away the whole family and forfeit the benefits of short-term reuse.

Direct cache writes amplify key mistakes

The query cache is not only read through hooks. `QueryClient` also lets you inspect and update it directly. That makes key correctness even more important because direct writes are immediate.

`queryClient.setQueryData` is synchronous and can create a cache entry if none exists. That is convenient when you intentionally seed detail data after a successful write or hydrate a known result. It is risky when the key is wrong. A typo or an over-broad factory can create an entry that looks real to later observers but does not correspond to the resource you thought you were updating.

```
queryClient.setQueryData(  
  ['posts', 'detail', 42],  
  { id: 42, title: 'Correct post' }  
)
```

If you accidentally write that same payload to `['posts', 'list']`, the mistake is not delayed until the next network round-trip. You have just reshaped a live cache entry synchronously.

For multi-query updates, `queryClient.setQueriesData` is the API designed for partial matching across existing entries. It updates only existing matching queries. That difference is subtle but important: `setQueryData` can create; `setQueriesData` updates matches that already exist. If you intend to touch a family of list variants but do not want to create new phantom entries, `setQueriesData` aligns better with that intent.

Reuse window and freshness window are separate boundaries

Two defaults strongly influence how key design feels at runtime: staleness and garbage collection. TanStack Query considers cached query data stale by default unless you configure `staleTime`. Inactive queries remain cached for five minutes by default through `gcTime`. Those are different mechanisms.

A key can therefore be reused from cache and still refetch immediately in the background. This is one reason developers sometimes misdiagnose “duplicate fetching.” The second mount is not necessarily missing the cache. It may be reusing cached data instantly and also refetching because the entry is stale.

You can see the separation in `QueryClient` defaults:

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

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 30_000,
      gcTime: 300_000,
    },
  },
})
```

Here, data is considered fresh for 30 seconds, but inactive entries can remain available for reuse for five minutes. That means a remount after

10 seconds will usually reuse data without an immediate stale-driven refetch, while a remount after 2 minutes can still reuse the cached data but may refetch in the background because freshness has expired. If you collapse those concepts, you will attribute behavior to the wrong cause.

Broader versus narrower segmentation is a trade-off

A narrower key schema increases correctness and targeting precision, but it also reduces reuse opportunities and can increase cache cardinality. A broader schema improves reuse only if the grouped resources are genuinely compatible. If they are not, broader keys create false sharing rather than efficient sharing.

Consider two approaches for posts:

- *Broad*: `['posts', id]`
- *Segmented*: `['posts', 'detail', id], ['posts', 'preview', id], ['posts', 'list', { tag, page }]`

The broad design is cheap to invent and expensive to operate. It assumes every post-shaped resource can coexist under one shallow pattern. The segmented design makes intent visible and enables more accurate family targeting. You pay with a slightly larger vocabulary, but you gain a cache topology that can support real application behavior.

A good decision signal is whether different views can safely consume the same cached shape without hidden assumptions. If not, segment them. Another is invalidation scope: if a future write should refresh all post lists but not every post detail, your keys need to preserve that distinction.

SSR and hydration expose freshness boundaries quickly

Server rendering adds a useful stress test for your key design and stal-

eness model. If you prefetch a query on the server and hydrate it on the client with the default `staleTime` of 0, the client can refetch in the background on page load even though data was already fetched during rendering. That is usually correct according to the default contract, but it surprises teams who assume hydration implies freshness.

A typical pattern looks like this:

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

const queryClient = new QueryClient()

await queryClient.prefetchQuery(postDetailOptions(42))

const dehydratedState = dehydrate(queryClient)
```

If you want to avoid an immediate client-side refetch for server-fetched data, increase `staleTime` for that query or its defaults. Staleness is measured from the server fetch time stored in the cache metadata, not from when the browser finishes hydrating. This is another place where key identity and freshness policy interact: the hydrated client is reusing the same cache entry, but policy still decides whether that reused entry is considered stale.

There is also a special boundary worth remembering: `staleTime: Infinity` still allows manual invalidation, while `staleTime: 'static'` blocks invalidation-triggered refetch behavior. If you set either value, do it intentionally. They are not interchangeable “never refetch” switches.

Inspect operations through the lens of key reach

Most production bugs around TanStack Query operations become easier to classify if you ask one question first: what exact set of keys can this action reach?

- A query mount reaches one concrete key.
- `invalidateQueries` reaches a matched key family.
- `refetchQueries` reaches a matched key family with explicit refetch intent.
- `setQueryData` reaches one exact key and may create it.
- `setQueriesData` reaches existing matched keys only.

That framing makes design flaws obvious. If an operation reaches too much, the hierarchy is too coarse or the prefix is too generic. If it reaches too little, the key family may be fragmented or semantically inconsistent. If it reaches the wrong data shape, variant separation is missing.

Treat the key schema as the cache's routing table

The cache behaves deterministically, but only relative to the addresses you give it. Well-shaped keys let related screens reuse data without conflation, let inactive entries remain useful during the `gcTime` window, and let invalidation or refetch operations target exactly the family you intended. Poorly shaped keys hide semantic differences, create phantom state through direct writes, and force you into broad repair strategies because no precise routing structure exists. Once key design becomes your routing table for cache operations rather than a naming afterthought, reuse, segmentation, and refetch scope stop feeling accidental and start becoming deliberate system behavior.

Chapter 5

Freshness, Refetching, and the Cache Lifecycle

Teams often misdiagnose TanStack Query behavior because they treat cache reuse, refetching, and cleanup as separate features rather than one lifecycle. This chapter unifies those mechanics into a single mental model so readers can predict why a query refetches, why another one survives navigation, and why a third silently disappears after inactivity.

5.1. Fresh, Stale, Active, Inactive, and Collected States

Two screens can ask for the same resource and still behave differently: one renders instantly from cache, one renders cached data and quietly refetches, and one has to fetch from the network because nothing remains in memory. If you describe all three cases as “cached” or “ex-

pired,” you lose the distinctions that actually drive TanStack Query behavior. You need three separate axes in your head:

- whether the cached data is considered *fresh* or *stale*,
- whether any mounted component is currently observing the query, making it *active* or *inactive*,
- whether the cache entry still exists at all, or has been *collected*.

Those axes are independent enough that teams often blur them together and then misdiagnose refetches, cold loads, and memory retention. TanStack Query becomes much easier to reason about once you treat freshness, observation, and retention as different properties of the same query entry.

A small example makes the distinction concrete. The query below defines a normal cached read. Nothing about it says whether the data is currently fresh, whether another component also observes it, or whether the entry will still exist after navigation. Those are lifecycle outcomes derived from time and observer presence.

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

async function fetchProject(projectId) {
  const res = await fetch(
    `https://api.example.com/projects/${projectId}`
  )

  if (!res.ok) {
    throw new Error('Failed to load project')
  }

  return res.json()
}

export function ProjectDetails({ projectId }) {
  const query = useQuery({
    queryKey: ['project', projectId],
    queryFn: () => fetchProject(projectId),
    staleTime: 60_000,
  })
}
```

```
      gcTime: 5 * 60 * 1000,
    })

    if (query.isPending) {
      return <p>Loading...</p>
    }

    if (query.isError) {
      return <p>{query.error.message}</p>
    }

    return <h1>{query.data.name}</h1>
  }
}
```

With that configuration, a successful fetch produces cached data that is treated as fresh for one minute. If every observer unmounts, the query becomes inactive, but the entry can remain in memory for up to five minutes before garbage collection removes it. If a component mounts again during that retention window, TanStack Query can reuse the same entry. That reuse may still be accompanied by a background refetch if the data is stale and refetch triggers allow it. The important point is that “stale” does not mean “gone,” and “unmounted” does not mean “removed.”

Fresh and stale describe age, not existence

Freshness answers a narrow question: *should TanStack Query currently treat this cached result as up to date enough to avoid normal background synchronization pressure?* It does not answer whether the data is present, correct, or being rendered by a component.

By default, cached query data is considered stale immediately after fetching unless you set `staleTime`. That default surprises many developers because the cache still works: TanStack Query can serve the cached result immediately, but the entry is also eligible for background revalidation on events such as mount, window focus, or reconnect. Staleness is therefore not a failure state. It is a signal that the data may need rechecking.

This distinction matters because stale data is often exactly what gives you a fast UI. Suppose you visit a list screen, fetch data, navigate away, then return thirty seconds later with `staleTime: 0`. The query is stale, but the data may still be in the cache. TanStack Query can render that stale result immediately and then refetch in the background. The user sees content without waiting on the network. If you incorrectly equate stale with missing, you will expect a loading screen where none is required.

Freshness is also time-based rather than truth-based. A query marked fresh is not guaranteed to match the server at this moment; it only means the library is currently willing to trust that cached value according to the policy you set. Likewise, a stale query may still be perfectly accurate. The fresh/stale boundary is a synchronization policy boundary, not a declaration of factual correctness.

Active and inactive describe observation, not quality

A query is active while at least one observer is attached to it, usually through a mounted `useQuery` or `useInfiniteQuery` consumer. Active means some part of your UI currently depends on that query entry. As long as the query remains active, it stays relevant to the running application and is not treated as unused cache residue.

When the last observer unmounts, the query becomes inactive. Inactive does not mean invalid, broken, or deleted. It means no mounted component currently needs the entry. TanStack Query keeps inactive queries in cache for later reuse because temporary absence from the UI is common: route transitions, tab switches inside the app, closing and reopening a modal, or leaving a detail screen and then returning moments later.

That inactive interval is operationally important. It is the difference between a warm return path and a cold one. If a user leaves a page and comes back before the inactive entry is garbage collected, the query

can often resume from existing data instead of starting from nothing. Whether that reused data is fresh or stale is a separate question. You can have an inactive query that is still fresh, an inactive query that is stale, or an inactive query that is eventually removed.

A useful mental shortcut is this:

- fresh/stale tells you how old TanStack Query considers the data,
- active/inactive tells you whether any mounted consumer currently cares,
- collected tells you whether the cache entry still exists.

Each property affects behavior, but they are not interchangeable.

Collected means removed from the cache

Once an inactive query remains unused long enough, garbage collection may remove it from the cache. Current TanStack Query terminology calls the retention setting `gcTime`. Older material often calls this `cacheTime`. In v5, `gcTime` is the accurate name because it describes when an inactive entry becomes eligible for garbage collection rather than implying some vague notion of “how long the cache exists.”

Collected is a useful explanatory term for the post-removal state. It is not a formal API status label you read from a hook. It simply means the cache no longer holds that query entry. After collection, there is nothing to reuse. A future mount with the same query key creates a new lifecycle from the perspective of the cache: the component asks for data, and TanStack Query must fetch or otherwise obtain it again.

This is stronger than staleness. Stale data still has reuse value because it can be rendered immediately while revalidation happens. Collected data has no reuse value because it is gone. If your team uses the word “expired” for both situations, debugging becomes harder than it needs

to be. Ask a sharper question instead: *is the data stale, inactive, or actually collected?*

Lifecycle state belongs to the query entry

These states belong to the query cache entry identified by the query key, not to an individual component instance. If two components mount with the same `queryKey`, they observe the same underlying entry. One observer mounting does not create a separate conceptual copy of server state. One observer unmounting does not necessarily make the query inactive if another observer is still attached.

That detail matters whenever multiple surfaces share a resource. If a sidebar and a detail view both read `['project', projectId]`, the query remains active as long as at least one of them stays mounted. If the detail view disappears but the sidebar remains, the query does not enter the inactive retention window. Teams sometimes assume route changes always make queries inactive, but layout persistence or shared components can keep an entry active longer than expected.

The same principle explains why cache behavior survives component churn. If a route remounts a component tree but reuses the same query key before the cache entry is collected, TanStack Query is not dealing with a brand-new resource. It is reconnecting observers to an existing entry whose freshness and retention clocks are already in motion.

Transitions between states

A successful fetch populates the cache entry. From that point forward, two clocks and one observer count shape the lifecycle.

First, freshness changes over time. If `staleTime` is greater than zero, the entry begins fresh and later becomes stale when enough time passes. If `staleTime` is zero, the entry becomes stale immediately after fetch while still remaining cached and reusable.

Second, observer count changes with UI activity. Mounting a component with the matching query attaches an observer. Unmounting removes one. When observer count drops to zero, the query becomes inactive.

Third, retention governs removal after inactivity. Once inactive, the query remains cached until the `gcTime` window expires. If another observer mounts before that deadline, the query becomes active again and the same entry continues its lifecycle. If not, the cache may collect it.

These transitions are easier to remember if you keep cause and effect separate:

- **Time since data update** changes fresh to stale.
- **Observer mounts and unmounts** change active to inactive.
- **Time since last observer detached** allows inactive data to be collected.

None of those transitions says anything, by itself, about whether the user must wait on a spinner. A stale active query may still render immediately. An inactive fresh query may be invisible because no component is using it. A collected query forces a colder path because the cache has nothing left to offer.

Operational flow: what the user experiences

Walk through a common navigation sequence and predict behavior from the vocabulary alone.

Stage 1: first visit. A component mounts with `['project', 42]`. No cached entry exists yet. TanStack Query fetches the data, stores

the result, and marks the query active because an observer is present. Depending on `staleTime`, the entry is either fresh for some interval or stale immediately after fetch.

Stage 2: navigate away. The component unmounts. If no other observer remains for `['project', 42]`, the query becomes inactive. The cache entry still exists. Nothing about navigation alone deletes it.

Stage 3: return quickly. A component mounts again before `gcTime` expires. TanStack Query reuses the existing cache entry. If the data is still fresh, it can usually be used without normal stale-triggered background synchronization. If the data is stale, it can still render immediately from cache and then refetch in the background if trigger settings allow. Either way, this is not a cold start because the query was inactive, not collected.

Stage 4: return later but before collection. The query is almost certainly stale now, but it may still be cached. The user often sees immediate content from the stale cache plus a background request. This is the scenario that most clearly separates stale from missing.

Stage 5: return after collection. The inactive retention window has elapsed, and garbage collection removed the entry. A mount with the same key no longer finds reusable data in memory. TanStack Query must recreate the entry and fetch again. This is the true cold path.

Those stages are predictable once you stop speaking loosely about data being “expired.” The behavior depends on whether the entry is present, whether it is observed, and whether it is considered fresh.

Precision in team language prevents configuration mistakes

Several recurring mistakes come from imprecise vocabulary rather than bad code.

Calling a query “expired” when it is merely stale pushes people toward

the wrong fixes. They may increase `gcTime` when the problem is over-aggressive synchronization, or increase `staleTime` when the real issue is repeated cold starts after collection.

Assuming an unmounted query disappears immediately leads to unnecessary loading-state design. In practice, inactive entries often survive route changes and support fast return navigation.

Using “cache” as if it meant a single state obscures the fact that TanStack Query deliberately serves cached stale data. That behavior is a feature, not a contradiction.

Version terminology can add noise here. If you encounter older examples using `cacheTime`, read them as discussing inactive retention. In modern v5 code, prefer `gcTime`. The rename sharpens the model: freshness is one timeline, observer activity is another, and garbage collection is a separate cleanup policy applied only after a query becomes inactive.

Once you name the states precisely, the rest of the lifecycle mechanics stop feeling mysterious. A query can be stale and still useful, inactive and still present, or fresh and still irrelevant because no observer currently needs it. The cache entry moves through those conditions according to time and observation, and the user experience follows directly from those transitions.

5.2. Using staleTime to Control Synchronization Pressure

A dashboard, a reference-data picker, and an editing screen rarely want the same read policy. The dashboard tolerates more network traffic because users care about timely numbers. The picker wants instant reuse because country names do not change often. The editing screen may

need stability more than freshness because background updates can distract the user or make values appear to drift while they are typing. `staleTime` is the control that lets you express those differences without rewriting your fetch logic or disabling the cache.

A concrete baseline makes the trade-off visible. The two queries below read server state with the same hook, but they apply radically different freshness windows because the underlying resources behave differently.

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

async function fetchTicker(symbol) {
  const res = await fetch(
    `https://api.example.com/ticker/${symbol}`
  )

  if (!res.ok) {
    throw new Error('Failed to load ticker')
  }

  return res.json()
}

async function fetchCountries() {
  const res = await fetch(
    'https://api.example.com/countries'
  )

  if (!res.ok) {
    throw new Error('Failed to load countries')
  }

  return res.json()
}

export function StockTicker({ symbol }) {
  return useQuery({
    queryKey: ['ticker', symbol],
    queryFn: () => fetchTicker(symbol),
    staleTime: 0,
  })
}

export function CountryList() {
  return useQuery({
```

```
    queryKey: ['countries'],  
    queryFn: fetchCountries,  
    staleTime: 60 * 60 * 1000,  
  })  
}
```

Both queries cache their results. The difference is how aggressively TanStack Query questions those cached results later. With `staleTime: 0`, the ticker becomes stale immediately after fetch, which keeps it eligible for background synchronization whenever normal stale-triggered events occur. With a one-hour `staleTime`, the country list remains fresh for long enough that route remounts, focus changes, and reconnects usually reuse the cached data without triggering unnecessary background requests.

What `staleTime` actually controls

`staleTime` controls freshness duration and therefore how often automatic refetch triggers actually do network work. That wording is precise and worth keeping. It does not tell TanStack Query how long to retain inactive data in memory. It does not schedule polling. It does not guarantee that cached data matches the server. It tells the cache how long to treat an existing result as fresh before it becomes eligible for normal stale-driven revalidation.

That distinction becomes clearer if you separate two questions:

- *Do you already have cached data for this query key?*
- *If you do, should TanStack Query currently trust it enough to avoid background synchronization pressure?*

The first question is about cache presence. The second is about freshness. `staleTime` answers only the second one.

By default, TanStack Query treats new query results as stale immediately after fetch. That default maximizes opportunities to revalidate.

It is a conservative stance toward server truth, but it can also increase network traffic because mount, focus, and reconnect events keep finding stale data that is eligible for background refetch. Many teams react to that behavior by disabling refetch triggers wholesale. That often throws away useful synchronization behavior when the simpler fix is to increase `staleTime` for resources that do not change often.

Freshness windows change the user experience

You can think of `staleTime` as a pressure valve on synchronization. A short window causes the client to question cached data frequently. A long window lets the client rely on the cache longer. The visible effect is not just request count; it is also how the interface feels when users move around the product.

If a query remains fresh during a route remount, the user often gets a stable, immediate render with no background churn from stale-triggered events. If the same query becomes stale almost instantly, the user may still get an immediate cached render, but the application will more often perform background refetches on remount, on focus, or after reconnect. That can be correct and desirable for volatile data, but wasteful for slow-changing reference data.

A short `staleTime` therefore biases toward correctness pressure. A long `staleTime` biases toward cache reuse and latency hiding. Neither is universally better. The useful choice depends on how costly stale reads are and how expensive repeated synchronization becomes.

A simple remount test

One of the easiest ways to feel the effect of `staleTime` is to set a short window, navigate away, and return within that interval. If the data remains fresh, the remount should reuse the cached result without stale-triggered background synchronization pressure.

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

```
async function fetchProfile(userId) {
  const res = await fetch(
    `https://api.example.com/users/${userId}`
  )

  if (!res.ok) {
    throw new Error('Failed to load profile')
  }

  return res.json()
}

export function UserProfile({ userId }) {
  const query = useQuery({
    queryKey: ['user', userId],
    queryFn: () => fetchProfile(userId),
    staleTime: 30_000,
  })

  if (query.isPending) {
    return <p>Loading...</p>
  }

  if (query.isError) {
    return <p>{query.error.message}</p>
  }

  return <p>{query.data.name}</p>
}
```

If you mount `UserProfile`, navigate away, and return within 30 seconds, the cached result is still fresh. TanStack Query can reuse it directly. If you return after 30 seconds, the data is stale. The cache entry may still exist and still render immediately, but it is now eligible for background synchronization under normal refetch triggers. That distinction is the practical value of `staleTime`: it determines whether a return path behaves like trusted reuse or like reuse plus revalidation pressure.

Choose `staleTime` from data volatility and UX tolerance

A useful stale-time decision starts with domain volatility. Ask how often the resource changes in ways that matter to the user. Then ask how

harmful it is to show an older value briefly.

For high-volatility data such as quotes, operational metrics, or live queue depths, a low `staleTime` is usually justified. You want the client to treat cached results as short-lived so that normal triggers can keep the view synchronized. Even then, `staleTime: 0` is not automatically the right choice for every volatile surface. If the backend is expensive and the UI already has explicit refresh controls or push updates, you may want a modest freshness window instead of immediate staleness.

For slow-changing reference data such as countries, currencies, product categories, or static policy text, long `staleTime` values are often the correct default. The user gains faster navigation and fewer needless requests, while the risk of seeing materially outdated information remains low.

For detail pages, the decision usually depends on how users navigate. If they move repeatedly between a list and a detail view, longer `staleTime` turns the cache into a strong latency-hiding layer. The page feels instant on return. If the resource is updated frequently by other actors and those updates matter immediately, shorten the window.

For editing flows, background change can be more harmful than slightly older reads. If the user is filling out a form from cached server data, aggressive revalidation can create distracting updates or force you to reconcile local edits against incoming data. A longer `staleTime` during the editing interaction often produces a calmer surface. That does not remove the need for invalidation after a successful mutation; it simply reduces incidental revalidation pressure while the user is working.

Zero, moderate, long, and infinite windows

Treat common `staleTime` ranges as policy categories rather than magic numbers.

Zero. With `staleTime: 0`, every successful fetch becomes stale immediately. This setting preserves maximum opportunity for background revalidation and is a reasonable default for highly dynamic data when you prefer correctness pressure over request minimization. It is also the default behavior even when you do not specify the option. The downside is that many ordinary lifecycle events can lead to repeated network work.

Moderate windows. Values such as 10 seconds, 30 seconds, or a few minutes are often the most practical. They still bound the age of trusted cache data, but they smooth out bursts caused by rapid navigation, tab switching, or transient remounts. Many products benefit more from these modest windows than from either extreme.

Long windows. Values measured in tens of minutes or hours work well for reference data and low-volatility lists. The cache becomes more of a local read-through layer and less of a continuous synchronization surface.

Infinity. `staleTime: Infinity` tells TanStack Query to keep considering the data fresh indefinitely until you explicitly invalidate or update it. This can be appropriate for immutable data, bootstrapped configuration, or application metadata that changes only through controlled administrative flows. Use it deliberately. Once you choose infinity, invalidation becomes the main path to freshness.

What `staleTime` does not do

Teams often misuse `staleTime` because the name sounds broader than it is. Three common misunderstandings cause most configuration errors.

First, `staleTime` is not a cache retention timer. It does not determine how long inactive data stays stored. An entry can become stale after 10 seconds and still remain available in memory for much longer if its re-

tention window allows it. Retention belongs to `gcTime`, not `staleTime`.

Second, `staleTime` is not a polling interval. Setting `staleTime: 5000` does not mean TanStack Query will refetch every five seconds. It means the data becomes stale after five seconds. Actual refetches still depend on events such as `mount`, `focus`, `reconnect`, or an explicit interval configuration.

Third, `staleTime` is not a correctness guarantee. Fresh means “trusted for now according to policy,” not “definitely current on the server.” A long window trades some freshness confidence for lower synchronization pressure and faster reuse.

Global defaults versus per-query policy

You usually want a sensible baseline and a few exceptions, not a random collection of per-component numbers. Configure a default freshness window that matches the broad shape of your application, then override it for data classes that are clearly more or less volatile.

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

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 5 * 60 * 1000,
    },
  },
})

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

That baseline gives you moderate reuse across route changes and

browser focus events. You can then override a volatile query downward or a stable query upward. This pattern produces more consistent behavior than scattering `staleTime: 0` across the codebase and then selectively silencing the resulting noise.

A good signal that your default is too low is widespread focus- or remount-driven traffic for resources that rarely change. A good signal that it is too high is user-visible old data lingering until manual refresh or explicit invalidation.

Server prefetch and hydration need matching freshness policy

Server prefetch is a place where `staleTime` often gets misunderstood. If you prefetch data on the server and then hydrate it into the client cache, the client still evaluates freshness when the mounted `useQuery` observer appears. If the hydrated data is considered stale on arrival, the client may refetch immediately. That is often surprising because the data already exists in the cache and just rendered successfully.

You can reduce that immediate post-hydration refetch pressure by giving prefetched queries a non-zero freshness window.

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

async function getPosts() {
  const res = await fetch(
    'https://api.example.com/posts'
  )

  if (!res.ok) {
    throw new Error('Failed to load posts')
  }

  return res.json()
}

export async function buildDehydratedState() {
```

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 60_000,
    },
  },
})

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

return dehydrate(queryClient)
}
```

That server-side default helps ensure the prefetched result is still fresh when the client mounts shortly afterward. One subtle rule matters here: a `staleTime` provided to `queryClient.prefetchQuery` affects that prefetch decision only. The corresponding `useQuery` call still needs its own `staleTime` if you want the same freshness behavior after hydration or later remounts. If you want consistent policy, express it in both places or in shared defaults.

Use invalidation for writes, not tiny stale windows everywhere

A common anti-pattern is using very small `staleTime` values as a substitute for explicit invalidation after writes. That strategy makes every read path pay the cost of aggressive revalidation, even when only a few mutation flows truly need immediate reconciliation.

A cleaner design is to choose `staleTime` based on read volatility and user interaction patterns, then invalidate the affected queries when mutations succeed. That preserves efficient reads for stable data while still letting write paths restore correctness quickly. Detailed mutation and invalidation strategy belongs with write coordination, but the practical rule is straightforward: use freshness windows to control background skepticism, and use invalidation to react to known state changes.

How to tune without guessing

You do not need exact science to improve `staleTime`, but you do need observable signals.

Start with a representative surface and ask four questions:

- How often does the underlying data change in meaningful ways?
- How costly is it if the user sees an older value briefly?
- How often do users remount this screen, refocus the tab, or reconnect?
- How expensive is extra background synchronization for your backend?

If the answers are “rarely changes,” “low harm,” “frequent revisits,” and “nontrivial backend cost,” increase `staleTime`. If the answers point toward rapid change and higher correctness demands, shorten it.

Then validate in behavior rather than theory. Navigate away and back quickly. Switch tabs. Reconnect after a transient offline period. Watch whether the interface remains stable, whether requests cluster unnecessarily, and whether users would plausibly notice stale values. The right number is usually the one that removes obvious waste without hiding changes that matter.

Treat `staleTime` as the primary business-level knob for synchronization pressure. Once you frame it that way, the option stops looking like a mysterious cache tweak and starts acting like a precise statement about how suspicious the client should be of its own data.

5.3. Understanding gcTime and Inactive Cache Retention

A query can stop being visible without becoming useless. That is the practical reason gcTime matters. Users move through route trees, close dialogs, switch tabs inside the product, and return to the same screen moments later. If the cache keeps an inactive query around, TanStack Query can reuse it and deliver a fast return path. If the entry is removed too aggressively, the same navigation pattern turns into repeated cold loads. gcTime is the control for that retention decision.

You can see the policy directly in a query configuration. The example below keeps project data in memory for ten minutes after the last observer unmounts. That does not make the data fresh for ten minutes. It only keeps the inactive cache entry available for reuse during that interval.

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

async function fetchProject(projectId) {
  const res = await fetch(
    `https://api.example.com/projects/${projectId}`
  )

  if (!res.ok) {
    throw new Error('Failed to load project')
  }

  return res.json()
}

export function ProjectScreen({ projectId }) {
  const query = useQuery({
    queryKey: ['project', projectId],
    queryFn: () => fetchProject(projectId),
    staleTime: 30_000,
    gcTime: 10 * 60 * 1000,
  })

  if (query.isPending) {
    return <p>Loading...</p>
  }
}
```

```
}

if (query.isError) {
  return <p>{query.error.message}</p>
}

return <h1>{query.data.name}</h1>
}
```

If a user opens `ProjectScreen`, leaves it, and comes back three minutes later, the query may already be stale, but it is still likely to be present because the ten-minute retention window has not expired. `TanStack Query` can therefore reuse the cached result immediately and decide separately whether to refetch in the background. If the user returns after twelve minutes and no other observer kept the query active, the entry may have been collected and the route loads cold.

What `gcTime` governs `gcTime` determines how long an inactive query remains in cache before being removed. That definition has two parts that you should keep separate.

First, the query must be *inactive*. `gcTime` only matters after a query loses all observers. As long as a component is still mounted and observing the query, retention is not under discussion. The query remains active and available.

Second, the timer governs *removal*, not freshness. A query can become stale within seconds and still remain cached for minutes or hours after the UI stops observing it. That is why `gcTime` solves a different problem than `staleTime`. `staleTime` answers “for how long should this data be treated as fresh?” `gcTime` answers “once nobody is using this query, how long should I keep it around in case somebody asks again?”

The official default inactive retention is five minutes. That default is long enough to make ordinary back-navigation and brief unmounts feel warm, but short enough to prevent the cache from retaining inactive

data indefinitely in typical client applications.

Inactive does not mean disposable It is easy to underestimate the value of inactive data because no mounted component is currently reading it. In practice, inactive query results often represent the highest-leverage part of the cache. They are the entries most likely to make navigation feel instant.

Consider a route flow with a list and a detail screen. The user opens a detail page, inspects a record, and returns to the list. The detail query becomes inactive the moment its last observer unmounts. If the same user then reopens that detail page thirty seconds later, the entry can still be sitting in memory. TanStack Query does not need to reconstruct everything from scratch. It can reuse the cached data immediately, then apply freshness logic independently.

That distinction matters because many production complaints that sound like “the cache is not working” are really retention problems. The data may be fetched correctly and reused while active, but a too-short retention window discards it before the navigation pattern can benefit.

A route-switch example A route-centric example makes the trade-off concrete. Suppose your app shows a customer list and a customer detail panel. Users often move back and forth between these two surfaces while handling support work.

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

async function fetchCustomer(customerId) {
  const res = await fetch(
    `https://api.example.com/customers/${customerId}`
  )

  if (!res.ok) {
    throw new Error('Failed to load customer')
  }
}
```

```
    return res.json()
  }

  export function CustomerDetails({ customerId }) {
    const query = useQuery({
      queryKey: ['customer', customerId],
      queryFn: () => fetchCustomer(customerId),
      staleTime: 60_000,
      gcTime: 10 * 60 * 1000,
    })

    if (query.isPending) {
      return <p>Loading...</p>
    }

    if (query.isError) {
      return <p>{query.error.message}</p>
    }

    return <section>{query.data.email}</section>
  }
}
```

This policy says two different things. The detail data stays fresh for one minute, which reduces background synchronization pressure on quick revisits. The cache retains inactive entries for ten minutes, which supports repeated back-and-forth navigation even after freshness has elapsed.

If a support agent flips between several customer details during the same session, a longer `gcTime` can significantly improve responsiveness. The entries may be stale by the time they are revisited, but stale inactive data still has reuse value. It can populate the screen immediately while a background refetch validates it. Shortening `gcTime` would not make those queries fresher; it would merely destroy the warm return path.

Why `gcTime` is not a freshness knob Teams often reach for the wrong option because they observe the wrong symptom. If the problem is excessive background requests when a screen remounts, your

first suspect is usually `staleTime`, not `gcTime`. A longer retention window will not suppress refetches for an active stale query. It only keeps inactive data from being removed.

The inverse mistake happens too: a team wants better back-navigation and increases `staleTime` dramatically. That may reduce refetch pressure, but it does not guarantee the entry will still exist after the user leaves the screen. If the query becomes inactive and the retention window expires, freshness no longer matters because the cache entry is gone.

That is why you should reason about `staleTime` and `gcTime` on different axes:

- `staleTime` manages synchronization pressure while data exists.
- `gcTime` manages reuse potential after observation ends.

Once you separate those roles, configuration decisions become much less ad hoc.

Memory is the cost side of the decision Longer retention improves reuse, but the retained data occupies memory. In a small application with modest payloads and a predictable navigation pattern, that cost may be negligible. In a large admin product with heavy payloads, wide parameter variation, and many low-reuse queries, it can be substantial.

Imagine an analytics console where each route loads a large report object, and operators rarely revisit the same report parameters within a session. If every report stays inactive in memory for a long time, the cache can accumulate many heavyweight entries with little reuse value. In that case, a shorter `gcTime` is often the better policy.

You can express that globally when the product shape justifies it:

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

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 30_000,
      gcTime: 60_000,
    },
  },
})

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

This configuration says: trust data briefly, and collect inactive entries quickly. That may fit a high-cardinality admin surface where memory pressure matters more than warm return paths. It would be a poor fit for a route flow that users bounce through repeatedly.

A longer `gcTime` makes sense when at least one of these is true:

- users frequently revisit the same resources,
- fetches are expensive or slow,
- payloads are moderate in size,
- route churn is common and warm reuse noticeably improves UX.

A shorter `gcTime` makes sense when these dominate instead:

- queries are numerous and parameter-rich,

- payloads are large,
- revisit rates are low,
- client memory is constrained,
- retaining old entries yields little practical benefit.

Infinity is a valid but narrow choice You can disable garbage collection for a query by setting `gcTime: Infinity`. That keeps inactive data available indefinitely unless you clear the client or otherwise remove the entry.

```
useQuery({
  queryKey: ['reference-data', 'countries'],
  queryFn: fetchCountries,
  staleTime: 24 * 60 * 60 * 1000,
  gcTime: Infinity,
})
```

This can be reasonable for a small set of long-lived reference queries that almost every route needs and that change rarely. It is much less appropriate for large or unbounded query spaces. If your key space includes user IDs, search terms, filter combinations, or report dimensions, infinite retention can quietly turn the cache into a memory leak with a very polite API.

Use `gcTime: Infinity` only when you can explain both the bounded size of the key space and the enduring value of each retained entry.

Version terminology matters Modern TanStack Query uses `gcTime`. Older tutorials and code snippets often use `cacheTime`. Treat them as the same conceptual setting in older material, but prefer `gcTime` in current code and documentation.

The rename improves precision. “Cache time” sounds like the total lifespan of cached data, which encourages exactly the misunderstanding-

ing you want to avoid. “Garbage-collection time” is more explicit: the timer becomes relevant when the query is unused and waiting to be removed. That wording reinforces the real model of inactive retention rather than vague cache duration.

If you are migrating older code or reading mixed-version examples, translate mentally:

`cacheTime` in older React Query material corresponds to
`gcTime` in current TanStack Query terminology.

Keep the underlying behavior straight even when the vocabulary varies.

Server-side behavior differs for good reasons Client-side retention and server-side retention do not serve the same operational goal. On the server, `gcTime` defaults to `Infinity`. That differs from typical client behavior because server rendering must avoid collecting prefetched data before rendering and hydration finish using it.

This difference matters most when developers try to optimize server memory too aggressively. Official guidance warns that `gcTime: 0` can cause hydration errors because data may be removed before rendering completes. If you are prefetching queries for server rendering and want a shorter server retention window, use a small nonzero value rather than zero.

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

async function getPosts() {
  const res = await fetch(
    'https://api.example.com/posts'
  )
```

```
    if (!res.ok) {
      throw new Error('Failed to load posts')
    }

    return res.json()
  }

export async function createState() {
  const queryClient = new QueryClient({
    defaultOptions: {
      queries: {
        gcTime: 2_000,
      },
    },
  })

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

  return dehydrate(queryClient)
}
```

A value like two seconds can be enough when you genuinely need short-lived server cache data, while still giving the rendering pipeline time to complete. In many SSR setups, leaving the server default alone is simpler and safer.

How to choose a practical `gcTime` Pick `gcTime` by looking at navigation reuse, payload cost, and memory constraints together.

If users often return to the same query within a few minutes, a retention window shorter than that revisit pattern leaves performance on the table. If users rarely come back and each query payload is large, long retention mostly wastes memory. Start from actual interaction patterns rather than abstract preference.

A practical tuning workflow looks like this:

1. Identify a query family with meaningful route churn.
2. Observe how often users revisit the same key within 1, 5, or 10 minutes.
3. Estimate payload size and number of concurrent inactive entries.
4. Increase `gcTime` when revisit rate is high and memory cost is moderate.
5. Decrease `gcTime` when revisit rate is low and inactive data accumulates broadly.

When a query is expensive to fetch but compact in memory, you can often justify a longer retention window. When a query is cheap to re-fetch but large and rarely reused, shorter retention is usually the better trade.

Anti-patterns to avoid A few mistakes recur often enough to treat them as design smells.

Using `gcTime` to reduce refetch noise. If queries refetch too often while mounted, adjust freshness or trigger policy. Retention only matters after inactivity.

Setting one very long global retention window without measuring. What helps a few core routes can harm a parameter-heavy product where thousands of inactive entries accumulate.

Assuming inactive queries are useless because they are stale. Stale inactive data can still provide immediate render value on revisit.

Papering over poor key design with long retention. If your query keys are too granular or unstable, increasing retention just stores more fragments for longer.

Forcing gcTime: 0 on the server. That optimization can break hydration before you ever gain meaningful memory savings.

If you keep those failure modes in mind, gcTime becomes a straightforward control instead of a mysterious cache setting. It is the policy that decides whether an unused query is worth keeping around for a likely return, and that decision is often the difference between a route that feels instantaneous and one that behaves like the user has never been there before.

5.4. Configuring Refetch on Mount, Focus, Reconnect, and Interval

Teams often describe TanStack Query refetches as random when the real problem is that they have not named the trigger. A screen remounts after navigation. A tab becomes visible again. The network drops and comes back. A polling interval fires while the user is reading the page. Each of those is an explicit synchronization event, and each interacts with freshness. Once you treat refetching as a trigger system layered on top of the cache lifecycle, the behavior becomes predictable.

A good starting point is a global baseline that keeps the useful defaults but makes your assumptions visible in code. TanStack Query configures these behaviors on `QueryClient` under `defaultOptions.queries`.

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

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 5 * 60 * 1000,
      refetchOnMount: true,
      refetchOnWindowFocus: true,
```

```
    refetchOnReconnect: true,
  },
},
})

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

This configuration means cached data stays fresh for five minutes. During that window, mount, focus, and reconnect normally do not trigger background refetches because those triggers normally refetch only when data is stale. After the five-minute window, the same events become synchronization opportunities again. That single observation ties the trigger system back to the freshness model: events do not act in isolation; they act against the stale or fresh state of each query.

The core rule: stale queries are the usual target

Three event-driven options matter most in everyday applications:

- `refetchOnMount`
- `refetchOnWindowFocus`
- `refetchOnReconnect`

All three default to `true` and normally refetch only when data is stale. That rule explains a large share of production behavior.

If a query remains fresh because `staleTime` has not elapsed, remounting a component does not usually produce a network request. Refocusing the window does not usually produce one either. Reconnecting after a brief offline period also stays quiet if the data is still fresh. Once

the query becomes stale, those same events become opportunities to synchronize in the background.

This gives you a compact operational matrix for a query with `staleTime: 5 * 60 * 1000`:

- **Remount after 30 seconds:** query is fresh, so mount usually reuses cache without refetch.
- **Refocus after 2 minutes:** query is fresh, so focus usually does not refetch.
- **Reconnect after 4 minutes:** query is fresh, so reconnect usually does not refetch.
- **Remount after 7 minutes:** query is stale, so mount may refetch in the background.
- **Refocus after 10 minutes:** query is stale, so focus may refetch in the background.

That behavior is often exactly what you want. Freshness limits request volume during routine UI churn, while stale data remains eligible for synchronization when the environment suggests the user may benefit from it.

Refetch on mount

`refetchOnMount` governs what happens when a query observer attaches during component mount. In route-driven interfaces, this is one of the most visible triggers because navigation frequently unmounts and remounts screens even when the user is conceptually revisiting the same data.

With `refetchOnMount: true`, a stale cached query typically refetches in the background when a new observer mounts. That preserves the

fast path of rendering cached data immediately while still checking whether the server has changed.

You can override the global baseline for a specific query when the surface is especially noisy or especially sensitive.

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

async function fetchAccount(accountId) {
  const res = await fetch(
    `https://api.example.com/accounts/${accountId}`
  )

  if (!res.ok) {
    throw new Error('Failed to load account')
  }

  return res.json()
}

export function AccountScreen({ accountId }) {
  const query = useQuery({
    queryKey: ['account', accountId],
    queryFn: () => fetchAccount(accountId),
    staleTime: 60_000,
    refetchOnMount: false,
  })

  if (query.isPending) {
    return <p>Loading...</p>
  }

  if (query.isError) {
    return <p>{query.error.message}</p>
  }

  return <h1>{query.data.name}</h1>
}
```

This disables mount-driven refetch for that query, even after it becomes stale. Use this sparingly. If a screen repeatedly remounts during the same user task and the underlying data changes rarely, disabling mount refetch can reduce noise. If the data changes often, the override may hide legitimate updates and push freshness responsibility else-

where.

Current documentation also allows 'always' for `refetchOnMount`. That forces refetch regardless of freshness except when `staleTime: 'static'` is used. Treat that as an advanced policy tool rather than a default instinct. It is appropriate only when you want mount itself to be a hard synchronization boundary.

Refetch on window focus

`refetchOnWindowFocus` addresses a simple reality: users leave your app and come back. They answer a message, switch tabs, inspect another dashboard, or open DevTools and then return. Focus-driven refetch gives stale queries a chance to synchronize at exactly the moment the user is likely to care again.

In current browser-oriented behavior, the default focus integration listens to document visibility changes. That detail explains a common debugging surprise: toggling between the app and browser tools can cause focus-driven refetches that feel mysterious until you remember that visibility changed.

If your application shows low-volatility data and users switch context frequently, global focus refetching may still be too chatty. You can reduce the pressure by increasing `staleTime` rather than immediately disabling `refetchOnWindowFocus`. That keeps the feature available for genuinely old data while suppressing unnecessary work for recent reads.

When a specific query should ignore focus changes, override it locally:

```
useQuery({
  queryKey: ['countries'],
  queryFn: fetchCountries,
  staleTime: 60 * 60 * 1000,
  refetchOnWindowFocus: false,
})
```

This is often appropriate for durable reference data. It is less appropriate for live operational views where returning focus is a strong signal that the user wants current information.

Like mount refetching, `refetchOnWindowFocus` also accepts `'always'` in current docs, again with the exception around `staleTime: 'static'`. An always-on focus policy can be useful for particularly sensitive data, but it is expensive if applied broadly.

Refetch on reconnect

`refetchOnReconnect` is the resilience trigger. When a device or browser regains network connectivity, stale queries can synchronize automatically instead of waiting for the user to navigate or refresh manually.

This behavior matters most in unreliable-network environments and mobile workflows, but it is also valuable on desktop applications that suspend network activity during tunnels, VPN changes, or transient offline periods. A stale query that failed to update while disconnected can recover naturally on reconnect if this trigger remains enabled.

The same freshness rule applies here: if the query is still fresh, reconnect usually does nothing. That means you can preserve reconnect correctness without causing needless requests every time the browser briefly toggles connectivity.

For data that is both low-volatility and high-volume, you may occasionally disable reconnect refetch on a per-query basis. More often, the better fix is to classify that data with a longer `staleTime` so reconnect does work only when the cached result is genuinely old.

Interval refetching is different

`refetchInterval` is not another stale-triggered event. It is scheduled polling. If you set it to a number, TanStack Query continuously

refetches at that cadence. This makes interval refetching qualitatively different from mount, focus, and reconnect. Those are opportunistic event hooks. Interval refetching creates ongoing synchronization pressure whether or not the user is navigating.

Use polling only when the surface truly needs recurring updates and no push-based alternative is practical. Notifications, job status summaries, and lightweight operational counters are reasonable candidates.

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

async function fetchNotifications() {
  const res = await fetch(
    'https://api.example.com/notifications'
  )

  if (!res.ok) {
    throw new Error('Failed to load notifications')
  }

  return res.json()
}

export function NotificationsBadge() {
  const query = useQuery({
    queryKey: ['notifications'],
    queryFn: fetchNotifications,
    staleTime: 0,
    refetchInterval: 15_000,
    refetchIntervalInBackground: false,
  })

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

  if (query.isError) {
    return <span>!</span>
  }

  return <span>{query.data.unreadCount}</span>
}
```

Here, the query polls every 15 seconds while the app is in the

foreground. `refetchIntervalInBackground: false` prevents that polling from continuing while the tab is in the background. That option is important because background polling can quietly create load that neither users nor developers notice until request volumes spike.

Polling does not replace freshness settings. `staleTime` still expresses whether cached data should be treated as fresh between interval ticks and other triggers. But in practice, once you introduce frequent polling, the interval becomes the dominant source of synchronization pressure. That is why broad use of `refetchInterval` is usually a smell.

Set defaults globally, override intentionally

A stable product usually benefits from a layered policy:

1. define sensible global defaults,
2. override only for clearly different data classes,
3. verify the result against real interaction patterns.

Global defaults give you a predictable baseline for most queries. Per-query overrides let you carve out exceptions for volatile dashboards, stable reference data, or interaction-sensitive editing surfaces. Without a global baseline, teams often accumulate a patchwork of local overrides that are impossible to reason about.

A useful default profile looks like this:

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 2 * 60 * 1000,
      refetchOnMount: true,
      refetchOnWindowFocus: true,
      refetchOnReconnect: true,
    },
  },
});
```

```
}))
```

From there, apply local overrides only when the domain justifies them. A stock ticker may need more aggressive synchronization. A country list may need much less. A form-backed detail page may want longer freshness and perhaps fewer event-driven updates while the user edits.

Custom focus integration

Browser visibility is a good default, but some runtimes do not map cleanly to standard browser focus semantics. Embedded shells, iframes, React Native, or specialized host environments may need custom wiring. TanStack Query exposes `focusManager.setEventListener` for that purpose.

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

focusManager.setEventListener((handleFocus) => {
  const onVisibilityChange = () => {
    handleFocus(document.visibilityState === 'visible')
  }

  window.addEventListener(
    'visibilitychange',
    onVisibilityChange,
    false
  )

  return () => {
    window.removeEventListener(
      'visibilitychange',
      onVisibilityChange,
      false
    )
  }
})
```

In a regular browser app, you usually do not need to replace the default handler. The point of the API is that you can bridge focus semantics from a host runtime into TanStack Query's synchronization model. In

React Native, official integration patterns bridge AppState into the focus manager for the same reason: the library needs a signal that means “the user has returned to an interactive foreground context.”

Use a custom focus listener only when the default browser behavior is wrong for your environment. Replacing it casually can break expectations across the app.

Custom online integration

Connectivity detection has the same shape. In ordinary browser applications, the default online handling is often sufficient. In embedded or mobile environments, you may need runtime-specific network signals. TanStack Query provides `onlineManager.setEventListener` for this.

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

onlineManager.setEventListener((setOnline) => {
  const onOnline = () => setOnline(true)
  const onOffline = () => setOnline(false)

  window.addEventListener('online', onOnline, false)
  window.addEventListener('offline', onOffline, false)

  return () => {
    window.removeEventListener('online', onOnline, false)
    window.removeEventListener('offline', onOffline, false)
  }
})
```

In React Native, official patterns typically bridge `NetInfo` into `onlineManager` rather than relying on browser events. The operational principle stays the same: reconnect-triggered refetch only works well if the runtime reports online state accurately.

Avoid accidental request storms

Most accidental refetch storms come from combining conservative freshness with aggressive triggers on too many queries at once. A short

or zero `staleTime` means many cached queries are stale most of the time. If you then enable broad focus refetching, reconnect refetching, and interval polling across list-heavy pages, the app can generate bursty traffic whenever the user returns to the tab or network conditions change.

A few patterns reliably reduce that risk:

- Increase `staleTime` for slow-changing resources before you disable triggers.
- Use polling only for surfaces that genuinely require repeated updates.
- Keep `refetchIntervalInBackground` conservative unless you have a clear need for background polling.
- Override refetch behavior per query only when the resource profile truly differs from the default.
- Treat 'always' as an exceptional policy, not a routine one.

If focus refetches look noisy in development, remember that switching between the page and DevTools can itself generate focus events. Validate behavior under normal user interaction, not just during debugging sessions.

Choose event policies from real user behavior

Trigger configuration works best when you map it to how users actually return to data. If users revisit screens by navigation, mount behavior matters most. If they spend time outside the app and come back, focus refetch matters. If they work through unstable connections, reconnect behavior matters. If they sit on one screen waiting for external changes, interval polling may be justified.

That framing keeps you from asking the wrong question. The question is not “should I turn refetching off?” The better question is “which events are meaningful signals that this user would benefit from synchronization for this class of data?” Once you answer that, the configuration surface becomes a set of precise tools rather than a collection of toggles.

5.5. Hydration, Activity, and Lifecycle Edge Cases

Hydrated data often looks deceptively simple on first paint: the page renders immediately, the cache already contains the query, and yet the browser still issues a request moments later. Teams commonly read that as a hydration bug when it is usually a normal consequence of the same lifecycle rules that govern every other query. Data restored into the cache still has freshness, activity, and retention semantics. Hydration changes when the cache entry appears, not the rules that govern what happens after it exists.

A compact server-prefetch example makes the tension concrete. The server fetches the query into a `QueryClient`, serializes it with `dehydrate`, and the client restores it with `HydrationBoundary`. The client component then mounts a normal `useQuery` observer against that same key.

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

async function getPosts() {
  const res = await fetch(
    'https://api.example.com/posts'
  )
```

```
    if (!res.ok) {
      throw new Error('Failed to load posts')
    }

    return res.json()
  }

export async function buildPage() {
  const queryClient = new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 60_000,
      },
    },
  })

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

  return dehydrate(queryClient)
}

function PostsPage() {
  const query = useQuery({
    queryKey: ['posts'],
    queryFn: getPosts,
    staleTime: 60_000,
  })

  if (query.isPending) {
    return <p>Loading...</p>
  }

  if (query.isError) {
    return <p>{query.error.message}</p>
  }

  return (
    <ul>
      {query.data.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

```
export function App({ dehydratedState, client }) {  
  return (  
    <QueryClientProvider client={client}>  
      <HydrationBoundary state={dehydratedState}>  
        <PostsPage />  
      </HydrationBoundary>  
    </QueryClientProvider>  
  )  
}
```

The important line in that example is not `dehydrate`; it is `staleTime`. Without a nonzero freshness window, the prefetched query can hydrate correctly, render immediately, and still refetch as soon as the client observer mounts because the query is already stale under the client policy. That outcome is often correct, but it surprises teams who thought “already in cache” automatically meant “no immediate network request.”

What hydration restores `dehydrate` serializes successful queries by default, and `hydrate` or `HydrationBoundary` restores them into a `QueryClient` on the client. That restoration creates or updates cache entries before the normal client-side observer lifecycle begins.

This is why hydrated UIs can render synchronously with real data on first mount. The client observer does not need to wait for an initial fetch if the query entry already exists in the cache. But restored presence is not the same as guaranteed freshness. The cache entry still has an update timestamp, a stale/fresh decision, and eventual refetch behavior driven by the same trigger rules described earlier.

Hydration also has a limited scope. `HydrationBoundary` only handles queries, not mutations. That boundary is useful because it clarifies what kind of state you are transferring. You are restoring server-read results, not replaying the entire client-side behavior of the application.

Why hydrated queries refetch immediately A hydrated query can refetch immediately for a simple reason: by the time the client observer mounts, the data is considered stale and the normal mount trigger is allowed to synchronize it. Nothing exotic is happening. The query entered the cache earlier than usual, but once present, it obeys the same rule as any other stale cached result.

This is the source of a common misunderstanding. Developers often collapse three separate facts into one:

- the data exists in the cache,
- the UI can render from that cache immediately,
- the library should suppress a follow-up request.

Only the first two are guaranteed by hydration. The third depends on freshness policy. Official guidance recommends a non-zero `staleTime` for server prefetch scenarios when immediate client revalidation is undesirable. If you actually want the client to verify server-rendered data right away, leave the query stale and let the mount behavior do its work.

This distinction is especially useful in SSR tuning. Some surfaces benefit from immediate client verification because the server response may already be a few seconds old by the time the browser runs. Other surfaces care more about reducing startup traffic and avoiding duplicate work. `staleTime` expresses that preference cleanly.

Hydration merges by recency Hydration is not a blind overwrite. Existing cache entries are overwritten only when the incoming dehydrated data is newer. That recency-based merge matters in route transitions and client navigations where overlapping queries may already exist in memory before a new dehydrated payload arrives.

Suppose the client already fetched ['posts'] after the initial page load, and later a route transition introduces another dehydrated payload for ['posts']. If the client copy is newer, hydration should not replace it with older server data. This protects the cache from moving backward in time.

That behavior solves a subtle class of bugs. Without recency-aware merging, a later navigation could accidentally overwrite a more up-to-date client entry with an older snapshot produced during server rendering. Instead, the cache preserves the newest known version.

Operationally, this means you should think of hydration as a timestamp-aware merge into the live cache rather than a wholesale replacement of client state.

Prefetched but unobserved queries still obey retention

Prefetching creates cache entries before the user necessarily sees them. That can improve responsiveness for anticipated navigations, but it does not exempt those entries from retention policy. If no observer ever mounts for a prefetched query, it can still be removed after `gcTime`.

This matters in route preloading strategies. You may prefetch data for a page the user never opens. That is usually acceptable, but remember what happens next: the query remains inactive because no observer attached, and the garbage-collection timer can eventually remove it. The cache is not a permanent holding area for speculative data unless you deliberately configure it to be one.

This behavior is healthy. Without it, aggressive prefetching could turn into unbounded memory growth. The lifecycle remains coherent: hydration or prefetch gives the query an earlier starting point, activity still depends on observers, and retention still determines how long unused data survives.

Seeding detail queries from list data `initialData` introduces a related edge case because it also populates the cache without waiting for a fetch. A common pattern is seeding a detail query from a list query you already have in memory. The detail screen can render instantly with matching data from the list instead of showing a loading state.

The subtle problem is freshness. `initialData` is persisted in the cache and is stale by default unless `staleTime` is set. More importantly, when you derive seed data from another query, you should provide `initialDataUpdatedAt` so TanStack Query can compute staleness correctly.

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

async function fetchTodo(todoId) {
  const res = await fetch(
    `https://api.example.com/todos/${todoId}`
  )

  if (!res.ok) {
    throw new Error('Failed to load todo')
  }

  return res.json()
}

export function TodoDetail({ todoId }) {
  const queryClient = useQueryClient()

  const query = useQuery({
    queryKey: ['todo', todoId],
    queryFn: () => fetchTodo(todoId),
    staleTime: 60_000,
    initialData: () => {
      const todos = queryClient.getQueryData(['todos'])

      return todos?.find((todo) => todo.id === todoId)
    },
    initialDataUpdatedAt: () => {
      const state = queryClient.getQueryState(['todos'])
      return state?.dataUpdatedAt
    },
  })
}
```

```
    if (query.isPending) {  
      return <p>Loading...</p>  
    }  
  
    if (query.isError) {  
      return <p>{query.error.message}</p>  
    }  
  
    return <h1>{query.data.title}</h1>  
  }  
}
```

Without `initialDataUpdatedAt`, the detail query can make the seeded data appear newer or older than it really is relative to your `staleTime`. That leads to confusing refetch behavior. If the list data was fetched ten minutes ago, the detail query should not pretend it was updated just now merely because you seeded it at mount time. Passing the originating timestamp preserves the real freshness relationship.

Route transitions can hide activity changes Some lifecycle surprises have nothing to do with hydration itself and everything to do with observer churn during navigation. A query may appear continuous to the user while actually crossing active and inactive boundaries under the hood. Layout swaps, modal routing, suspense boundaries, and nested route replacement can detach the last observer for a moment and then mount a new one shortly after.

Whether that brief inactivity matters depends on timing and policy. If a new observer mounts quickly, the same cache entry may be reused with no visible effect. If the query is stale, the remount may trigger a background refetch. If the transition delays long enough and retention is short, the entry could even be collected before the new observer arrives, turning a warm transition into a cold one.

That is why route behavior can feel inconsistent until you map it back to the underlying state machine. The library is not reasoning about “pages” in the abstract. It is reasoning about cache entries, observers,

freshness timestamps, and garbage-collection windows.

A practical debugging question is therefore not “why did this page refetch?” but “did the old observer unmount, was the query stale when the new observer mounted, and did the entry remain in cache long enough to be reused?” Those questions align directly with the lifecycle model.

Restored cache state is presence, not authority Persistence layers and hydration share an important property: they restore cache presence, not authoritative truth. A restored query may be fresh, stale, active, inactive, or soon to be collected depending on timestamps, observers, and configuration. You should not treat restored state as a special category that escapes the normal rules.

This is especially important in offline-capable applications. A restored cache gives the app something useful to render immediately, but it does not imply that the data is still current or that background synchronization should be suppressed. Once the environment permits it, the same freshness and trigger policies apply.

This perspective also prevents a category error during debugging. If a restored query refetches on mount, that does not mean restoration failed. It usually means restoration succeeded and the client then applied its normal synchronization policy.

Server memory and gcTime edge cases Server-side query clients behave differently enough that retention needs extra care. Official guidance warns against `gcTime: 0` on the server because data may be removed before rendering completes, causing hydration errors. On the server, `gcTime` defaults to `Infinity` specifically to avoid premature collection during SSR.

If you need early server memory release, do not force a zero retention window during rendering. A safer approach is to clear the client after response handling when the dehydrated state has already been produced and the render pipeline is done.

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

async function buildState() {
  const queryClient = new QueryClient()

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

  const dehydratedState = dehydrate(queryClient)

  queryClient.clear()

  return dehydratedState
}
```

That pattern protects correctness first. Server rendering is a short-lived process, so predictable completion matters more than micro-optimizing in-flight cache lifetime.

Failure modes worth recognizing quickly A handful of recurring symptoms map directly to lifecycle rules.

- “Hydration worked, but the browser still fetched.” The query was present but stale, and mount-triggered synchronization ran as configured.
- “Seeded detail data refetches unexpectedly.” The seed likely lacked an accurate freshness timestamp, or the detail query’s `staleTime` policy does not match the intended reuse window.

- “Client data disappeared after prefetching but before navigation.” The prefetched query never became active and later aged out under `gcTime`.
- “A route transition sometimes feels warm and sometimes cold.” Observer timing, stale status, or retention expiration changed across those navigations.
- “Server-rendered data occasionally fails to hydrate.” Server retention may be too aggressive, especially if `gcTime: 0` was used.

These are not unrelated edge cases. They are ordinary lifecycle outcomes expressed in SSR, prefetch, and restoration contexts.

Keep the model uniform The most useful discipline in advanced TanStack Query work is refusing to invent separate mental models for hydration, persistence, and seeded data. A hydrated query is still just a query entry with data, timestamps, observers, and retention. A restored cache is still just cached state subject to freshness and triggers. Seeded `initialData` is still cached data whose age matters.

Once you keep that model uniform, the surprising cases stop looking magical. The question is no longer whether hydration is special, but when the entry was created, how old the data is, whether an observer is attached, and what the configured triggers are allowed to do when the client becomes active again.

Chapter 6

Mutations, Invalidation, and Consistency After Writes

A successful write is not the moment the server responds; it is the moment every affected view in the application becomes trustworthy again. This chapter shows how TanStack Query approaches that problem without a normalized client graph, teaching readers when to lean on invalidation, when to repair cached read models directly, and how to survive the edge cases created by optimism, concurrent user actions, and partial failure.

6.1. Modeling Server Writes with `useMutation`

Reads and writes fail in different ways, and that difference drives nearly every design choice on the write path. A query can usually be run again,

observed from another component, or refreshed in the background with little ambiguity. A mutation is a request to change remote state. Once you send it, the server may accept it, reject it, partially process it, or apply side effects that other parts of the interface now depend on. That makes a mutation less like “a query you run manually” and more like a coordinated operation with execution, state tracking, and follow-up responsibilities.

A concrete example makes the boundary clear. Suppose you need to update a todo item’s title from a form submit. The mutation should handle the server call and expose operational state to the component. It should not be expected to automatically make every cached list and detail view correct again. That later consistency work belongs to invalidation, direct cache repair, or optimistic coordination, each chosen deliberately rather than assumed.

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

type UpdateTodoInput = {
  id: number
  title: string
}

type Todo = {
  id: number
  title: string
  completed: boolean
}

async function updateTodo(
  input: UpdateTodoInput
): Promise<Todo> {
  const response = await fetch(`~/api/todos/${input.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ title: input.title }),
  })

  if (!response.ok) {
    throw new Error('Failed to update todo')
  }
}
```

```
    return response.json()
  }

  export function TodoTitleForm() {
    const mutation = useMutation({
      mutationFn: updateTodo,
    })

    function onSubmit(event: React.FormEvent<HTMLFormElement>) {
      event.preventDefault()
      const formData = new FormData(event.currentTarget)
      const title = String(formData.get('title') ?? '')
      mutation.mutate({ id: 42, title })
    }

    return (
      <form onSubmit={onSubmit}>
        <input name="title" />
        <button
          type="submit"
          disabled={mutation.isPending}
        >
          {mutation.isPending ? 'Saving...' : 'Save'}
        </button>

        {mutation.isError && (
          <p>{String(mutation.error)}</p>
        )}

        {mutation.isSuccess && (
          <p>Saved: {mutation.data.title}</p>
        )}
      </form>
    )
  }
}
```

That small example carries the essential mutation vocabulary. You define a `mutationFn` that performs the server write. You call `mutation.mutate(variables)` in response to an event such as a submit or button click. TanStack Query tracks mutation state for you and exposes that state as `status`, `isPending`, `isSuccess`, `isError`, `data`, `error`, `variables`, and related fields. The component uses those fields to disable controls, render inline progress, and display a result

or error.

Mutations are imperative by design

This event-driven execution model is the first important conceptual break from queries. Queries are declarative subscriptions to server state: you describe what data the component needs, and TanStack Query decides when to fetch based on the key, observers, and freshness policy. Mutations are different because the user or application must explicitly decide when the write occurs. A save button, delete confirmation, or drag-and-drop reorder action is an imperative event, so `useMutation` exposes an imperative trigger.

That imperative trigger does not mean you should move unrelated application control flow into the mutation hook. The mutation exists to model a server action and surface its operational state. It is appropriate to call it from an event handler. It is a mistake to invoke it during rendering, to hide it inside memoized computation, or to treat it as a generic async task runner for UI-only work. If no remote state change is involved, `useMutation` is usually the wrong abstraction.

The object-based API in v5

TanStack Query v5 uses an object-style hook signature. That is more than a stylistic preference. It keeps option names explicit and makes the lifecycle surfaces readable when a mutation becomes more sophisticated. The core options you will use most often are `mutationFn`, `mutationKey`, `onMutate`, `onSuccess`, `onError`, `onSettled`, `retry`, `retryDelay`, `networkMode`, `gcTime`, `meta`, `scope`, and `throwOnError`.

Start with `mutationFn`. It receives the variables passed to `mutate` or `mutateAsync`. Those variables represent the input to the write operation, not a cache key and not a fetch parameter bundle that TanStack Query replays automatically. In practice, this means you should shape

mutation variables around the command you are issuing: { `id`, `title` } for an update, { `id` } for a delete, or { `title`, `dueDate` } for a create. This command-oriented style reads more clearly than forcing writes into the mental model of data lookup.

State transitions and what they mean

A mutation instance moves through a small but useful state machine: `idle`, `pending`, `success`, or `error`. You can read the raw `status` string or the derived booleans `isIdle`, `isPending`, `isSuccess`, and `isError`. These states are operational signals for the UI.

Idle means no submission is in flight and no completed result is currently represented by the mutation state. *Pending* means the mutation function has been triggered and has not yet settled. *Success* means the most recent mutation completed successfully and produced data. *Error* means the most recent mutation failed and exposed an error object.

Those states let you draw a sharp line between execution feedback and consistency feedback. A disabled submit button, a spinner, or a “Saving...” label are direct reactions to `isPending`. A success message based on `isSuccess` indicates that the mutation call resolved successfully. It does *not* prove that every affected query in the application is already synchronized. That distinction matters because readers often overload mutation success with broader guarantees than the hook provides.

The mutation object also exposes `variables` and `submittedAt`. `variables` is especially useful when rendering the exact input associated with a pending or failed request. For example, a form can show the title that is currently being submitted, or a list can temporarily render a pending row based on submitted values. `submittedAt` can help you order or reason about overlapping submissions when you need lightweight timing context.

Lifecycle callbacks as coordination points

The mutation lifecycle callbacks are not interchangeable. Each one exists at a different point in the operational flow and should carry a distinct category of work.

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

type CreateTodoInput = {
  title: string
}

type Todo = {
  id: number
  title: string
  completed: boolean
}

async function createTodo(
  input: CreateTodoInput
): Promise<Todo> {
  const response = await fetch('/api/todos', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(input),
  })

  if (!response.ok) {
    throw new Error('Failed to create todo')
  }

  return response.json()
}

export function NewTodoForm() {
  const queryClient = useQueryClient()

  const createTodoMutation = useMutation({
    mutationFn: createTodo,
    mutationKey: ['todos', 'create'],
    onMutate: async (variables, context) => {
      await context.client.cancelQueries({
        queryKey: ['todos'],
      })
      return { submittedTitle: variables.title }
```

```
    },
    onSuccess: (data) => {
      console.log('Created todo', data.id)
    },
    onError: (error, variables, onMutateResult) => {
      console.error('Create failed', error)
      console.log('Failed title', variables.title)
      console.log(
        'Submitted title',
        onMutateResult?.submittedTitle
      )
    },
    onSettled: () => {
      return queryClient.invalidateQueries({
        queryKey: ['todos'],
      })
    },
  },
})

return null
}
```

onMutate: preflight work

`onMutate` runs before the mutation function resolves. It is the place for preflight coordination that must happen immediately after submission, often before the network result returns. In advanced patterns, this is where you cancel relevant queries, snapshot existing cache data, or prepare rollback material for optimistic updates. The value returned from `onMutate` is carried forward to later callbacks, which makes it the natural place to capture context specific to that mutation invocation.

You should keep `onMutate` focused. Use it for mutation-local preflight work, not for broad UI orchestration. If a modal should close only after a confirmed success, closing it in `onMutate` is usually premature. If a query might overwrite an optimistic patch, cancellation in `onMutate` is exactly the right move.

onSuccess: server confirmation

`onSuccess` runs after the mutation resolves successfully. Use it for

actions that depend on confirmed server acceptance: reading server-returned data, enqueueing success notifications, or beginning post-write consistency work. A common mistake is to treat `onSuccess` as a dumping ground for every side effect triggered by a save. That quickly tangles mutation execution with routing, form resets, dialog management, analytics, and cache policy. The callback remains more maintainable when you keep it close to the semantics of “the server accepted the write.”

Another subtle point matters under concurrency: if you trigger multiple requests from the same mutation hook instance, `onSuccess` fires only after the latest call. Do not assume that a hook instance gives you a callback stream for every overlapping request in completion order. If per-call behavior matters, structure the interaction carefully and avoid attaching fragile assumptions to shared mutation state.

`onError`: failure handling

`onError` runs when the mutation fails. That failure might be a network exception, a non-ok HTTP response that you converted into an error, or domain validation surfaced by your API layer. This callback is where you restore rollback state for optimistic updates, translate transport failure into user-facing feedback, or log enough context to debug a bad submission path.

A good `onError` implementation preserves the difference between request failure and user correction. If the server rejects invalid form data, you may want field-level errors in component state instead of a generic toast. If the network drops during a non-idempotent write, blindly retrying may be unsafe. TanStack Query defaults mutation retry to 0, which is a conservative default for exactly this reason: many writes should not be replayed automatically without domain guarantees.

`onSettled`: cleanup and reconciliation

`onSettled` runs after either success or error. It is the right place for cleanup that should happen regardless of outcome, and it is often the right boundary for consistency work that must always run after the write attempt finishes. When invalidating related queries from `onSettled`, return the Promise from `invalidateQueries` so the mutation remains pending until refetch-driven reconciliation completes. That keeps mutation state aligned with user-visible reality more accurately than flipping to success while dependent views are still catching up.

Choosing between `mutate` and `mutateAsync`

Most UI event handlers should start with `mutate`. It is simple, explicit, and aligns well with React event-driven code. You trigger the write, and the hook exposes state for rendering pending, success, and error feedback. This style avoids turning every event handler into a hand-written async control-flow graph.

Use `mutateAsync` when you genuinely need a Promise in local control flow, such as coordinating several awaited operations in sequence or integrating with code that already expects Promise-based composition. Even then, keep the trade-off in view: once you await a mutation in an event handler, it becomes tempting to move too much UI orchestration into imperative `try/catch` logic. That can duplicate what mutation state and lifecycle callbacks already model cleanly.

```
const mutation = useMutation({
  mutationFn: updateTodo,
})

async function handleSubmit() {
  try {
    const saved = await mutation.mutateAsync({
      id: 42,
      title: 'Write chapter',
    })
    console.log(saved)
  } catch (error) {
    console.error(error)
  }
}
```

```
}  
}
```

This is valid code, but it should be a deliberate choice rather than a default habit. If all you need is “run the write and reflect its state in the UI,” `mutate` is often the cleaner tool.

Separating execution, cache coordination, and UI feedback

A reliable mental model for mutations has three layers. First, execute the server command. Second, coordinate server-state consistency. Third, present user feedback that reflects both execution and reconciliation accurately. `useMutation` directly owns the first layer and exposes signals for the third. It does not automatically solve the second.

That separation prevents several common mistakes. A successful mutation response does not automatically synchronize affected queries. A disabled button while `isPending` is not the same thing as a globally trustworthy read model. A success toast fired from `onSuccess` may be premature if active views still need invalidation and refetch to reflect the write. Treating those responsibilities separately makes later consistency strategies easier to reason about because you are not asking one hook to stand in for a whole state-management architecture.

Practical patterns for create, update, and delete

Creates, updates, and deletes all fit the same operational flow, but each tends to surface a different emphasis.

For a **create**, the variables usually hold a command payload with fields from a form. Pending state often disables submission and may render a temporary placeholder based on `variables`. Success commonly returns a canonical server-created record with an assigned identifier.

For an **update**, the variables should include both target identity and patch data. Mutation success may return the full updated object or a

smaller acknowledgement payload. If the server returns the authoritative updated resource, that can later support direct cache repair; if not, invalidation is usually safer.

For a **delete**, the variables are often minimal, but the operational risk is higher than the payload size suggests. A delete may remove an item from lists, invalidate detail screens, affect counts, and interact badly with retries if the server contract is not clearly idempotent. The mutation itself still only models the command execution. You make later consistency choices explicitly.

Common anti-patterns

Several mistakes appear repeatedly in production code.

Calling `mutate` during render or from render-driven effects is the first. Mutations represent events, not subscriptions. Trigger them from user intent or explicit application actions.

Assuming mutation success automatically updates all related queries is the second. TanStack Query does not infer the blast radius of a write for you.

Overloading `onSuccess` with unrelated UI choreography is another. Keep server-acceptance logic separate from route transitions, modal lifecycles, or analytics unless those actions are truly inseparable from confirmed success.

Using `mutateAsync` everywhere because Promises feel familiar is also costly. It encourages imperative control flow where state-driven rendering is often clearer.

Finally, avoid treating mutation state as a global truth source. The hook tells you about the operation you triggered. It does not replace disciplined query-key design, cache invalidation strategy, or read-model repair.

Once you adopt that boundary, `useMutation` becomes easier to use correctly. It is the execution model for server writes: explicit input, explicit trigger, explicit lifecycle, and explicit state. That clarity is what makes later post-write consistency work tractable instead of accidental.

6.2. Using Targeted Invalidation as the Default Consistency Strategy

A write rarely affects just one screen. You edit a single record, but that record may also appear in a detail view, several filtered lists, a count badge, and a dashboard summary. The tempting response is to patch every cache entry that might contain the changed data. That approach feels efficient at first because it avoids refetching, but it quietly turns the client into a fragile reconstruction of server-side read models. TanStack Query pushes you toward a different discipline: after a successful write, mark the affected query families stale and let the established read path fetch authoritative state again.

That choice is not accidental. TanStack Query intentionally does not maintain a normalized client graph, and mutations are not automatically linked to affected queries. Once you accept that design, invalidation becomes the default post-write tool not because it is the most aggressive option, but because it scales better as read models multiply, filtering logic changes, and server-side business rules evolve. You keep the client honest by asking the server for fresh truth rather than by simulating a local source of truth that TanStack Query was never designed to provide.

A concrete example shows why invalidation earns that role. Suppose a user edits a todo title on a detail page. The same todo may appear in the unfiltered list, an “open” list, a search result list, and a sidebar

count of overdue items. If the write response returns only the updated todo object, you can update the detail cache directly. But you still do not know whether the title change affects search results, list ordering, server-computed badges, or other derived views. Targeted invalidation lets you pay the network cost only where the application actually needs fresh data.

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

type UpdateTodoInput = {
  id: number
  title: string
}

async function updateTodo(input: UpdateTodoInput) {
  const response = await fetch(`/api/todos/${input.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ title: input.title }),
  })

  if (!response.ok) {
    throw new Error('Failed to update todo')
  }

  return response.json()
}

export function useUpdateTodo() {
  const queryClient = useQueryClient()

  return useMutation({
    mutationFn: updateTodo,
    mutationKey: ['todos', 'update'],
    onSuccess: async (_data, variables) => {
      await Promise.all([
        queryClient.invalidateQueries({
          queryKey: ['todos', 'detail', variables.id],
        }),
        queryClient.invalidateQueries({
          queryKey: ['todos', 'list'],
        })
      ])
    }
  })
}
```



```
    }},  
    }},  
    },  
    }},  
  }  
}
```

This example does two useful things. First, it keeps the write itself inside `mutationFn`. Second, it treats cache trust as a separate concern handled in `onSuccess`. The detail query for the edited item is invalidated narrowly by identifier. The list family is invalidated more broadly because multiple list variants may include the same item under different filters or sorts. You do not patch each list variant manually. You declare which query families may now be stale and let TanStack Query reconcile them.

What invalidation actually does

Invalidation does not mean “refetch everything immediately.” That misconception leads many teams either to avoid invalidation or to overestimate its cost. When you call `invalidateQueries`, TanStack Query marks matching queries stale. Active queries usually refetch because they currently have observers and the application is looking at them. Inactive matching queries remain stale in the cache until something observes them again. That detail matters operationally. It means invalidation is often cheaper than manual repair for broad write effects: the queries the user can currently see are refreshed soon, while dormant variants are not eagerly reloaded.

This behavior also explains why invalidation composes well with rich query-key design. If you have several cached variants of the same resource—for example, `['todos', 'list', { status: 'open' }]` and `['todos', 'list', { status: 'done' }]`—invalidating the `['todos', 'list']` family marks all of them stale, but only the active ones normally refetch right away. The inactive ones wait until the user navigates back to them. That gives you broad correctness coverage

without the immediate cost of refetching every cached variant.

Hierarchical query keys are the targeting surface

Invalidation only works as well as your query keys allow. If your keys are inconsistent, flat, or overloaded, you have no precise way to express the blast radius of a write. Hierarchical keys give you a practical namespace for post-write coordination. A common pattern is to organize keys from domain to shape to parameters:

```
['todos']
['todos', 'list']
['todos', 'list', { status: 'open' }]
['todos', 'list', { status: 'done' }]
['todos', 'detail', 42]
['todos', 'count', 'overdue']
```

With keys shaped like this, you can invalidate at different scopes:

- ['todos', 'detail', 42] for one known entity view.
- ['todos', 'list'] for all list variants.
- ['todos'] for the entire todo domain.

That hierarchy is not just cosmetic. It encodes how much uncertainty you have after a write. If you are certain that only a detail view is stale, use the full key. If any list derived from the resource may be affected, invalidate the list family. If the write may influence counts, aggregates, and multiple query shapes, invalidate the broader domain slice instead of pretending you can patch every consequence locally.

You should also avoid using invalidation as a patch for poor key discipline. If every read concerning todos shares a single vague key, you lose the ability to distinguish detail, list, and aggregate caches. The result is either over-broad invalidation or risky under-invalidation. Good query keys are the precondition for cheap correctness after writes.

Scoping invalidation by certainty, not optimism

The key design question is not “How little can I invalidate?” The better question is “How much do I know about the server-side effects of this write?” That shift changes the default posture. Narrow invalidation is attractive when the blast radius is well understood. Broader invalidation is safer when the consequences of the write flow through filters, ordering, permissions, server-generated fields, or aggregate statistics.

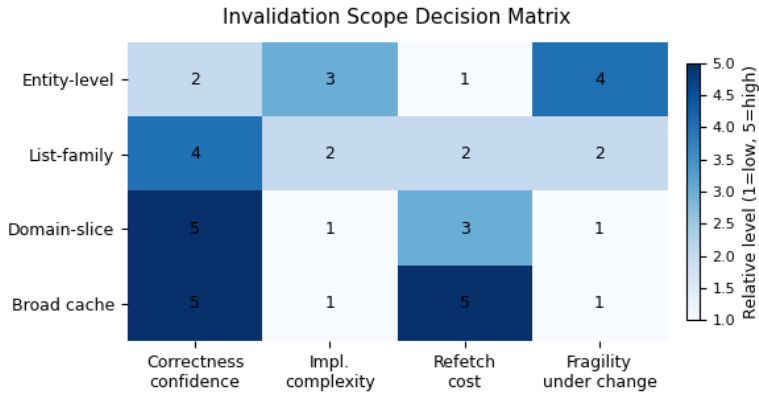
Consider four common scopes:

Entity-level invalidation targets one detail query or one tightly identified cache entry. Use it when the server confirms a change to one resource and you know no broader materialized view depends on fields altered by the write.

List-family invalidation targets every cached variant of a collection view. Use it when one record may move in or out of filtered lists, change sort position, alter search matching, or otherwise affect multiple list projections.

Domain-slice invalidation targets a wider namespace, such as all todo-related queries. Use it when a write can affect both lists and aggregates, or when the server applies rules whose client-visible consequences span several query shapes.

Broad cache invalidation marks very large portions of the cache stale. This preserves correctness but often imposes unnecessary refetch work and can degrade perceived performance if used habitually.



Why invalidation is usually safer than direct repair

Direct cache updates with `setQueryData` are valuable, but they are a precision tool. They work best when the mutation response is authoritative for one known read model. As soon as the same resource appears in several filtered, sorted, paginated, or aggregated views, the maintenance burden shifts quickly. You now have to understand not only the updated entity, but also every read model in which that entity might appear and every server rule that might reposition or transform it.

Invalidation avoids that trap by preserving server ownership. You do not need to duplicate filtering logic, recompute counts, or guess at list ordering rules in the client. You simply mark a query family stale and let the normal query functions re-establish the read models. That is why invalidation remains the safer baseline when the complete effect of a write is uncertain.

A practical signal helps you decide: if your post-write code starts enumerating multiple query shapes and custom merge rules, you are already paying the cost of a hand-maintained graph. At that point, broader invalidation is often both simpler and more reliable.

Choosing narrow versus broad invalidation

You can make invalidation decisions using a small set of operational questions.

How certain are you about affected read models? If the answer is “only this detail page,” invalidate narrowly. If the answer includes “probably some lists” or “maybe the dashboard count,” widen the scope.

How expensive is a refetch compared with client repair code? If a list refetch is cheap and user-visible correctness matters, invalidate. If the server returns a canonical updated object and only one detail entry is affected, direct repair may be justified.

How often does the write happen? High-frequency writes can magnify the network impact of broad invalidation. Even then, do not jump directly to client-side choreography. First ask whether your keys let you invalidate a narrower family without sacrificing correctness.

How stable are the read models? If list filters, computed badges, and dashboard summaries evolve often, manual repair code becomes fragile faster than invalidation policies do.

Does the server response fully describe the post-write truth? If the response omits derived fields, related counts, or computed statuses, invalidation is the safer choice.

Policy patterns for larger applications

On larger teams, you often want some consistency between mutation types and invalidation targets. One architectural pattern is to standardize `mutationKey` values and then pair them with invalidation policy in shared code. Another is to attach invalidation intent in `meta` and interpret it centrally. Treat these as architectural conventions rather than built-in framework behavior. They can reduce duplication, but

they also create indirection. Use them only when the application has enough write traffic or enough repeated patterns to justify the abstraction.

A conservative local pattern is often clearer:

```
return useMutation({
  mutationFn: updateTodo,
  mutationKey: ['todos', 'update'],
  onSuccess: (_data, variables, _result, context) => {
    return Promise.all([
      context.client.invalidateQueries({
        queryKey: ['todos', 'detail', variables.id],
      }),
      context.client.invalidateQueries({
        queryKey: ['todos', 'list'],
      }),
      context.client.invalidateQueries({
        queryKey: ['todos', 'count'],
      }),
    ])
  },
})
```

This keeps invalidation policy close to the mutation that necessitates it. When the write affects several read-model families, you can still express that explicitly without hiding the behavior behind a global rule.

Common anti-patterns

Blanket invalidation of the entire cache is the most obvious misuse. It preserves correctness, but if you make it your default habit, you erase the value of disciplined query keys and create avoidable network churn.

The inverse mistake is under-invalidation. Developers often patch the detail view, maybe one visible list, and assume they are done. The stale count badge in the sidebar or the outdated filtered list on another route appears later as a bug that is hard to reproduce because it depends on which views were active at the time of the write.

Another anti-pattern is treating manual cache repair as more authorita-

tive than the server. A local patch may look fast, but if the server applies permission checks, computed fields, deduplication, or reordering, the cache repair becomes speculative. Invalidation restores the boundary: the server owns truth; the client marks which truth may have changed.

Finally, do not use invalidation to compensate for poor key structure. If invalidating one changed record requires invalidating an entire product area, the fix is often to redesign query keys rather than to build increasingly clever invalidation exceptions.

Operational expectations after invalidation

When you invalidate matching queries, active observers usually refetch soon, and inactive ones remain stale until they are observed again. That behavior is often exactly what you want after a write. Users looking at affected data receive fresh results promptly. Users not looking at those views do not pay an immediate cost. This makes invalidation especially effective for applications with many filtered or parameterized list variants cached over time.

It also helps you reason about user-visible trust. A mutation's success means the server accepted a write. Invalidation then turns that accepted write into consistent reads by re-running the established query path where needed. You avoid the false promise that one successful mutation callback has already repaired every representation of the changed state.

Targeted invalidation works because it respects the division of labor TanStack Query is built around. Mutations express intent to change remote state. Query keys describe the families of server state you cache. Invalidation connects the two without forcing the client to become a normalized data engine. When the read surface around a resource grows more complex, that restraint becomes an advantage rather than a limitation.

6.3. Repairing Cached Read Models with Direct Cache Updates

Invalidation is the safest default after writes because it keeps the server in charge of truth. Still, there are cases where a refetch is unnecessarily blunt. If the server has already returned the authoritative post-write object, and you know exactly which cached read models depend on it, you can repair those read models directly and avoid an immediate round trip. That is the narrow opening where direct cache updates make sense.

The key word is *narrow*. TanStack Query does not maintain a normalized client graph, so a direct cache write does not automatically fan out across related queries. When you call `queryClient.setQueryData(queryKey, updaterOrValue)`, you are not “updating an entity.” You are updating one specific cached read model identified by one specific query key. If the same entity appears elsewhere, you must repair each affected cache entry explicitly or accept temporary inconsistency and use invalidation instead.

A straightforward detail-view update shows the best-case scenario. A user edits a todo title. The server returns the complete updated todo. The detail query for that todo already exists in the cache. Rather than refetching the same record immediately, you can replace it with the server response.

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

type Todo = {
  id: number
  title: string
  completed: boolean
}

type UpdateTodoInput = {
```



```
    id: number
    title: string
  }

  async function updateTodo(
    input: UpdateTodoInput
  ): Promise<Todo> {
    const response = await fetch(`~/api/todos/${input.id}`, {
      method: 'PATCH',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ title: input.title }),
    })

    if (!response.ok) {
      throw new Error('Failed to update todo')
    }

    return response.json()
  }

  export function useUpdateTodo() {
    const queryClient = useQueryClient()

    return useMutation({
      mutationFn: updateTodo,
      onSuccess: (updatedTodo) => {
        queryClient.setQueryData(
          ['todos', 'detail', updatedTodo.id],
          updatedTodo
        )
      },
    })
  }
}
```

This is the ideal shape for direct repair. The mutation response is authoritative. The target cache entry is known precisely. The semantics are simple: replace the detail record with the canonical server result. You avoid a network request that would merely return the same data, and every observer of that detail query sees the new value synchronously.

Read-model repair, not entity mutation

That last point needs discipline. When you update a cache entry directly, you are repairing a read model, not mutating a shared object database. This distinction matters because the same resource can be materialized in different forms: a detail record, a paginated list item, a filtered collection, a count badge, or a dashboard summary. TanStack Query does not infer relationships among those forms. Direct repair is therefore a manual, per-view operation.

This is why the primitive is powerful and dangerous at the same time. It is powerful because it is exact. It is dangerous because exactness does not scale well when the number of read models grows. The question is never “Can I write the cache directly?” You usually can. The real question is “Can I guarantee that I know every read model that must change, and can I repair each one correctly without duplicating server logic?”

Updating a detail record and one list item together

A common next step is to patch a detail query and a simple list query at the same time. That is still a reasonable use of direct repair if the list shape is stable and the server response includes all fields required by the list item.

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

type Todo = {
  id: number
  title: string
  completed: boolean
}

async function updateTodo(
  input: { id: number; title: string }
): Promise<Todo> {
  const response = await fetch(`/api/todos/${input.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
```

```
    },
    body: JSON.stringify({ title: input.title }),
  })

  if (!response.ok) {
    throw new Error('Failed to update todo')
  }

  return response.json()
}

export function useUpdateTodo() {
  const queryClient = useQueryClient()

  return useMutation({
    mutationFn: updateTodo,
    onSuccess: (updatedTodo) => {
      queryClient.setQueryData(
        ['todos', 'detail', updatedTodo.id],
        updatedTodo
      )

      queryClient.setQueryData(
        ['todos', 'list'],
        (old: Todo[] | undefined) => {
          if (!old) {
            return old
          }

          return old.map((todo) =>
            todo.id === updatedTodo.id
              ? updatedTodo
              : todo
          )
        }
      )
    },
  })
}
```

Two constraints keep this code safe enough to justify itself. First, the list is a simple array of the same record shape, so replacing one item preserves cache structure. Second, you know that `['todos', 'list']` is the only relevant collection view. If either constraint breaks, the optimization becomes harder to defend.

Notice the updater function. Use updater functions whenever the new value depends on the previous cached value. That preserves correctness when the cache entry may be absent and keeps the update immutable. Do not mutate the existing array or record in place. Create and return a new structure. React components and TanStack Query observers depend on referential change to propagate updates reliably.

Why direct repair feels attractive

The performance case for direct updates is real. If the server has already sent the new state, an immediate refetch may add latency with no informational gain. Direct repair avoids that extra request, removes a visible delay on fast edits, and reduces network churn when users make repeated small changes. In detail-heavy interfaces, that can materially improve responsiveness.

The implementation case is also appealing at first. One hook, one mutation, one `setQueryData` call. The code looks local and explicit. That clarity is exactly why direct repair is valuable for limited, well-understood read models. You are not introducing a global policy or forcing a broad refetch after every edit. You are saying, with precision, “I know which cached value changed, and I have the canonical replacement in hand.”

Where brittleness enters

The trouble begins when the same entity exists in multiple cached projections. Suppose you keep these query keys:

```
['todos', 'list']  
['todos', 'list', { status: 'open' }]  
['todos', 'list', { status: 'done' }]  
['todos', 'detail', 42]  
['todos', 'search', { q: 'report' }]
```

Now consider an update that changes a todo’s title and completion state. You can still patch the detail query. But what about the unfiltered

list, the open list, the done list, and the search results? If the completion state changed from open to done, the item should disappear from one filtered list and appear in another. If the title changed, it may enter or leave a search result set. If the server also adjusts an ordering field or computes a derived status, the client must now replicate those effects across several query shapes.

This is where direct repair stops being simple replacement and becomes local reimplementing of read-model logic. The maintenance burden rises sharply because every derived view is now part of your client write path. The code grows, and the risk of partial correctness grows with it.

A concrete failure mode

A failure example makes the risk more obvious. Suppose you patch only the unfiltered list after a successful update because that is the list currently visible on screen. Another route still has the open-only list cached. The todo was marked complete, but that filtered cache still contains it because you never repaired that read model. The interface now shows two contradictory truths depending on where the user navigates.

```
queryClient.setQueryData(  
  ['todos', 'list'],  
  (old: Todo[] | undefined) => {  
    if (!old) {  
      return old  
    }  
  
    return old.map((todo) =>  
      todo.id === updatedTodo.id  
      ? updatedTodo  
      : todo  
    )  
  }  
)
```

This code is not wrong for `['todos', 'list']`. It is incomplete for

the application. That distinction is the core hazard of direct repair in a non-normalized cache. A local patch can be perfectly correct for one query key and still leave the overall interface inconsistent.

Knowing when to fall back to invalidation

The right decision signal is not whether you *can* patch multiple queries. It is whether the cost of maintaining those patches is lower than the cost of refetching authoritative state. As soon as you have filtered lists, sorted collections, duplicated entities across several views, or server-side derivations that are not trivial to reproduce, invalidation usually becomes the safer and cheaper long-term choice.

A useful rule is to ask whether your write response is authoritative for every affected read model, not just for the updated entity. If the answer is no, direct repair is already on unstable ground. A single updated record does not necessarily tell you how every list should reorder, which filters should include it, or how aggregate views should change.

This is why broad direct repair often ages badly. It may work when the application has one list and one detail screen. Then product requirements add filters, badges, tabs, summaries, and pagination. The original “simple cache patch” quietly becomes a brittle network-avoidance strategy that leaks stale data into edge cases.

Using `setQueriesData` carefully

TanStack Query also provides `setQueriesData` for updating multiple matching queries. It can reduce repetition when several cache entries share the same shape and truly should receive the same transformation. Use it only when that assumption is solid. If different query variants require different inclusion, ordering, or aggregation logic, a broad multi-query update can spread the wrong patch more efficiently.

When you consider `setQueriesData`, pause and test your mental model: are you updating several instances of the same read model

shape, or are you forcing a generic transformation onto distinct materialized views? If it is the latter, invalidation is usually the more honest tool.

Preserving cache shape and immutability

Direct repair code must preserve the exact structure that readers of the query expect. If a query caches an array, return an array. If it caches an object with nested metadata, preserve that envelope while replacing only the changed part. If you drift from the original shape, you create failures that are harder to diagnose than a stale value because components may now read fields that no longer exist or rely on pagination metadata you accidentally discarded.

Immutability is equally important. Never edit cached objects or arrays in place. Even if a mutation seems to “work” locally, in-place mutation can break memoization assumptions and cause subtle stale renders. Always return new objects, new arrays, or a carefully reconstructed wrapper object with only the necessary parts changed.

Direct repair for create and delete

Creates and deletes make the trade-off sharper. If a create mutation returns the full new record and you have one obvious collection to append to, direct repair can work well. But the moment the new item might appear conditionally in several filtered lists or affect counts and summaries, the apparent simplicity disappears.

Deletes are similar. Removing an item from a detail query and one list is straightforward. Removing it from every cached filtered list, paginated page, search result, and aggregate count is much less so. In practice, direct repair for delete operations is often defensible only when the resource appears in a very small number of explicit read models.

Pragmatic decision criteria

You can justify direct cache updates when most of the following are true:

- The mutation response contains the authoritative post-write data.
- You know the exact query keys that must change.
- The affected read models are few and structurally stable.
- The client update does not need to reproduce complex filtering, sorting, pagination, or aggregation rules.
- Avoiding a refetch produces meaningful latency or network savings.

You should prefer invalidation when any of these conditions fail:

- The same entity appears in many cached variants.
- A write may change list membership or ordering.
- Aggregates, counts, or summaries depend on the write.
- The server enriches or transforms the stored data.
- The patch logic is starting to look like a hidden normalized layer.

A balanced pattern

A practical compromise is often to repair the narrowest obvious read model and invalidate the broader family. For example, replace the detail query immediately for responsive UI, then invalidate the relevant list prefix so collections reconcile from the server. That pattern gives you fast local confirmation without committing to full client-side repair across every variant.


```
onSuccess: (updatedTodo) => {
  queryClient.setQueryData(
    ['todos', 'detail', updatedTodo.id],
    updatedTodo
  )

  return queryClient.invalidateQueries({
    queryKey: ['todos', 'list'],
  })
}
```

This keeps the optimization bounded. The detail page updates instantly because you already have the canonical record. The broader collection views remain trustworthy because they refresh through the standard query path rather than through increasingly intricate merge code.

Direct cache updates are at their best when you use them as a surgical optimization for explicit, local read models. Once you find yourself reasoning about every place an entity might appear, every filter it might enter or leave, and every summary it might influence, you are no longer doing a small performance optimization. You are taking ownership of read-model synchronization in a system that is intentionally designed to keep that ownership with the server.

6.4. Implementing Optimistic Updates and Roll-back Policies

The fastest write is the one the user sees before the server replies. That speed is also the source of risk. Once you render an expected result ahead of confirmation, you are no longer merely tracking a request. You are temporarily asserting a future state that may be rejected, modified, or overtaken by other server-side rules. Optimistic updates therefore sit on a different risk curve than invalidation and direct cache re-

pair. They improve perceived responsiveness, but they do so by moving some of the server's job into the client.

A good starting point is to separate two forms of optimism that are often conflated. TanStack Query v5 documents a UI-only pattern that uses mutation variables to render pending state without touching the cache. It also documents cache-based optimism using `onMutate`, where you write anticipated results into cached queries before the server confirms them. The first approach is much simpler and should be your default optimistic technique when only one place in the interface needs to show the pending change. The second approach affects every observer of the touched query data and demands a rollback strategy.

A minimal UI-only optimistic append illustrates why this pattern is attractive. You can render a temporary list item from the mutation's submitted variables while the request is pending, using `submittedAt` as a stable temporary key.

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

type Todo = {
  id: number
  title: string
  completed: boolean
}

type CreateTodoInput = {
  title: string
}

async function createTodo(
  input: CreateTodoInput
): Promise<Todo> {
  const response = await fetch('/api/todos', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(input),
  })
}
```

```
    if (!response.ok) {
      throw new Error('Failed to create todo')
    }

    return response.json()
  }

export function NewTodoRow() {
  const mutation = useMutation({
    mutationFn: createTodo,
  })

  return (
    <>
      <button
        onClick={() => mutation.mutate({ title: 'Write tests' })}
        disabled={mutation.isPending}
      >
        Add
      </button>

      {mutation.isPending && mutation.variables && (
        <li key={mutation.submittedAt}>
          {mutation.variables.title} (saving...)
        </li>
      )}

      {mutation.isError && (
        <p>Save failed: {String(mutation.error)}</p>
      )}
    </>
  )
}
```

This pattern is optimistic because the user sees the intended new item immediately, but it never rewrites shared cache state. If the mutation fails, the temporary row simply disappears or changes into an error affordance. If the mutation succeeds, the normal read path can refresh the actual list. You get most of the perceived responsiveness with much less rollback complexity.

When UI-only optimism is enough

Use UI-only optimism when the pending result matters in one local

rendering context: a form that should show a submitted row, a button that should toggle its own label, or a detail screen that needs a temporary badge while a request is pending. Because you are not altering cached queries, you do not need to reason about cross-component consistency, cancellation of in-flight reads, or restoring prior cache state. The optimistic artifact is just presentation state driven by the mutation object.

This matters more than it may first appear. Many teams reach for cache-level optimism too early because they equate “fast UI” with “rewrite the cache immediately.” In practice, a local pending affordance often delivers most of the user experience benefit without taking ownership of shared state reconciliation. If only one place needs to show the expected change before confirmation, keep the optimism local.

Cache optimism changes the consistency model

Cache-based optimism is different. Here you write an expected result into the query cache before the server answers. Every component observing the affected query now sees the optimistic value as if it were already real. That broader reach is useful when multiple views must stay in sync instantly, but it also means failure is no longer local. If the assumption proves wrong, you must restore trust in all affected observers.

The documented sequence is precise:

- Cancel relevant queries.
- Snapshot previous cache state with `getQueryData`.
- Apply the optimistic patch with `setQueryData`.
- Return rollback material from `onMutate`.
- Restore on `onError`.

- Reconcile with `invalidateQueries` in `onSettled`.

Skipping any of these steps weakens the safety of the pattern. The cancellation step is especially important. If a stale refetch is already in flight and you do not cancel it, that response can arrive after your optimistic patch and overwrite it with older data. Optimism without cancellation is often optimism without isolation.

A cache-based optimistic toggle is a good example because the server-side effect is small and locally predictable.

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

type Todo = {
  id: number
  title: string
  completed: boolean
}

async function toggleTodo(
  input: { id: number; completed: boolean }
): Promise<Todo> {
  const response = await fetch(`/api/todos/${input.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      completed: input.completed,
    }),
  })

  if (!response.ok) {
    throw new Error('Failed to toggle todo')
  }

  return response.json()
}

export function useToggleTodo() {
  const queryClient = useQueryClient()
```

```
return useMutation({
  mutationFn: toggleTodo,
  onMutate: async (variables) => {
    await queryClient.cancelQueries({
      queryKey: ['todos', 'detail', variables.id],
    })

    const previousTodo = queryClient.getQueryData<Todo>([
      'todos',
      'detail',
      variables.id,
    ])

    queryClient.setQueryData<Todo>(
      ['todos', 'detail', variables.id],
      (old) => {
        if (!old) {
          return old
        }

        return {
          ...old,
          completed: variables.completed,
        }
      }
    )

    return { previousTodo }
  },
  onError: (_error, variables, onMutateResult) => {
    queryClient.setQueryData(
      ['todos', 'detail', variables.id],
      onMutateResult?.previousTodo
    )
  },
  onSettled: (_data, _error, variables) => {
    return queryClient.invalidateQueries({
      queryKey: ['todos', 'detail', variables.id],
    })
  },
})
}
```

This is the canonical shape of cache optimism. You snapshot exactly the cache entry you are about to patch. You return that snapshot so later callbacks can restore it. You invalidate after settlement so the

final state comes from the server, not from your local assumption. Returning the Promise from `invalidateQueries` keeps the mutation pending until reconciliation completes, which aligns the mutation life-cycle with the actual visibility of trustworthy data.

Why cancellation comes first

Query cancellation is not a cosmetic preflight step. It prevents one of the most common race conditions in optimistic systems: an in-flight read returns older data after you have already patched the cache optimistically. If that stale response is allowed to commit, the UI appears to flicker backward. Users see their action take effect, then disappear, then reappear after settlement. Even worse, if the final invalidation never fires because of an error path bug, the stale read may linger.

TanStack Query supports `queryClient.cancelQueries({ queryKey })`, and query functions receive an `AbortSignal` through their context. Cancellation only fully protects you if your query functions respect that signal. If they ignore it, the transport may continue and the old response may still race back into the cache. For optimistic writes on hot data, treat signal-aware queries as part of the same correctness story.

Rollback is about trust, not aesthetics

A failed optimistic mutation is not just a failed request. It is a moment where the UI has already shown the user something that did not actually happen. Your rollback policy has to repair that breach of trust.

Three rollback strategies are useful in practice.

Refetch-only rollback works when a simple invalidation can restore an authoritative view quickly and reliably. This is acceptable if you did not lose essential prior state and the affected query can be reloaded without ambiguity.

Snapshot restore uses the value returned from `onMutate` to put the

cache back exactly as it was before the optimistic write. This is the most explicit rollback and is usually the right choice when the prior state is available locally and the affected query is small enough to snapshot cheaply.

Partial preserve with error marking can make sense when the optimistic change should remain visible as user intent, but must now be marked unconfirmed or failed. TanStack Query does not provide a special built-in API for this pattern; you model it in your own UI or cache shape if you choose to support it. Use it carefully because it introduces a hybrid state that is neither purely local nor server-confirmed.

The choice depends on failure type. A transient network error may justify preserving user intent visually while retry options are offered. A server-side validation error usually requires rollback because the optimistic state is semantically invalid. A permission failure almost always requires explicit reversal because the user was never allowed to make the change.

Successful confirmation is still reconciliation

Even when the server accepts the optimistic write, you should not assume your optimistic value matches the canonical response perfectly. Servers often enrich records with timestamps, normalize formatting, compute derived fields, assign identifiers, or apply business rules that are invisible from the client. That is why `onSettled` invalidation remains part of the documented pattern. The optimistic patch buys you time and responsiveness; the refetch restores authority.

This is especially important for creates. A locally rendered optimistic row based on input variables may lack a server-generated identifier, a normalized title, an ordering position, or derived status flags. You can show the row immediately, but you still need reconciliation to replace the optimistic placeholder with the real record.

Complexity rises faster than expected

Optimistic cache logic is manageable for toggles, tiny field edits, or append placeholders. It becomes much harder once the server applies rules that the client does not fully own. Filtered lists are the classic example. A change to one field may remove an item from the current list, insert it into another, or reorder several siblings. Reordered collections are harder because the client must predict placement accurately. Generated server fields are harder because the optimistic object is incomplete by construction. Permission checks and related-resource side effects are harder because the write may succeed partially or alter data the client did not even touch directly.

The deeper issue is duplication of server logic. Every optimistic cache rule you add is an assumption about how the server will resolve the write. If that assumption is wrong, rollback becomes more frequent and the user experience becomes less trustworthy. Optimism pays off when the likely success path is both common and easy to predict. It loses value when correctness depends on server-side decisions the client cannot model cheaply.

A practical decision heuristic

You can evaluate optimistic techniques with four questions.

Is the success path highly predictable? A boolean toggle is predictable. A category change that affects several filtered views is less so.

How many query shapes are affected? One local component favors UI-only optimism. Several shared queries increase the case for cache optimism, but only if the change remains simple.

Can you roll back cleanly? If you can snapshot and restore the prior state precisely, the risk is manageable. If rollback requires reconstructing several list variants, optimism becomes expensive.

What does failure mean to the user? If a failed optimistic change would be confusing or harmful, you need a very strong reason to show it early.

These questions often narrow the field quickly. Use UI-only optimism when one component needs immediate feedback. Use cache optimism for simple, high-confidence transitions shared across observers. Use neither when the write's consequences are broad, policy-heavy, or difficult to reverse.

Anti-patterns to avoid

The first anti-pattern is optimistic cache writes without a rollback plan. If you cannot describe how the application recovers from failure, the optimistic patch is premature.

The second is optimistic updates across too many query shapes. Patching detail, several filtered lists, counts, and summaries in `onMutate` is usually a sign that invalidation should remain the primary strategy.

Another is treating the optimistic value as final and skipping reconciliation. Even when the write succeeds, the server remains the authority on canonical state.

A subtler mistake is using cache-level optimism where a local pending affordance would provide almost the same experience. If one button label or one nearby list row is the only thing that needs to react immediately, do not take on global rollback complexity for local UX.

Blending optimistic techniques with broader consistency policy

Optimistic updates do not replace invalidation or direct cache repair; they compose with them. UI-only optimism often pairs naturally with later invalidation. Cache-based optimism often pairs with snapshot rollback and a final reconciliation refetch. In some cases you can patch

one detail query optimistically while still invalidating broader list families after settlement. That layered approach is often safer than trying to predict every downstream effect eagerly.

The discipline is to keep each mechanism responsible for one part of the job. Optimism improves perceived latency. Rollback restores trust on failure. Invalidation re-establishes server-authoritative state. When you blur those boundaries, the write path becomes hard to reason about under stress.

Optimistic updates are most valuable when they are small, intentional lies that you can correct quickly and completely. Once the lie requires the client to simulate filtering rules, ordering logic, generated fields, and cross-resource side effects, the cost of feeling instant starts to exceed the value of actually being trustworthy.

6.5. Operating Mutations Under Concurrency, Failure, and User Feedback Constraints

A mutation flow that looks correct in isolation can still fail under real user behavior. A user double-clicks Save. Another tab edits the same record. A background refetch returns while an optimistic patch is active. One request times out, another succeeds, and the interface shows a success toast over stale data. None of those failures require exotic code. They appear naturally once writes overlap in time, the network becomes unreliable, and several UI surfaces react to the same state transition.

The operational challenge is not just to make writes succeed. You need the interface to remain trustworthy when success, failure, and synchronization do not line up neatly. TanStack Query gives you the pieces: mutation state, mutation filters, cache invalidation, query cancellation, and global mutation observation. The hard part is choosing the right

combination for each failure mode instead of applying one blanket rule.

A concrete example helps frame that discipline. Suppose you want to disable duplicate submissions for a save action and show a global indicator when matching writes are pending elsewhere in the app. TanStack Query's mutation filtering surfaces are built for exactly this kind of operational control.

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

type UpdatePostInput = {
  id: number
  title: string
}

async function updatePost(input: UpdatePostInput) {
  const response = await fetch(`/api/posts/${input.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ title: input.title }),
  })

  if (!response.ok) {
    throw new Error('Failed to update post')
  }

  return response.json()
}

export function SavePostButton() {
  const queryClient = useQueryClient()

  const mutation = useMutation({
    mutationFn: updatePost,
    mutationKey: ['post', 'update'],
    onSuccess: (_, data, variables) => {
      return queryClient.invalidateQueries({
        queryKey: ['post', 'detail', variables.id],
      })
    },
  })
}
```

```
const pendingUpdates = useMutationState({
  filters: {
    mutationKey: ['post', 'update'],
    status: 'pending',
  },
  select: (mutation) => mutation.state.variables,
})

const hasPendingUpdate = pendingUpdates.length > 0

return (
  <button
    disabled={mutation.isPending || hasPendingUpdate}
    onClick={() =>
      mutation.mutate({
        id: 42,
        title: 'Revised title',
      })
    }
  >
    {hasPendingUpdate ? 'Saving...' : 'Save'}
  </button>
)
```

This code does two things worth carrying into production practice. It scopes the write with a `mutationKey`, which gives you a durable targeting surface for observation and control. It also uses `useMutationState` to inspect matching pending mutations across the application, not just inside the current hook instance. That distinction matters because local hook state is often too narrow once you have list rows, dialogs, and detail screens all capable of triggering the same logical operation.

Observe mutations at the right scope

For single-button feedback, the state returned by one `useMutation` hook is enough. For application-wide behavior, it usually is not. `useMutationState` is the official v5-friendly way to observe multiple pending mutations across the app. You can filter by `mutationKey` and select specific fields such as `mutation.state.variables`. That lets you

build cross-cutting behavior without inventing a separate mutation registry.

This is useful for more than spinners. You can display a badge saying that three saves are pending, highlight rows currently being updated, or suppress a duplicate action for a specific resource. Because the selector can return submitted variables, you can also tell which entity is affected, not just that “something is pending.”

Duplicate submissions are a concurrency bug first

Users do not intend to create concurrency problems; they usually just want reassurance that the click worked. A button that remains enabled during a slow request invites a second click, and that second click may produce duplicate records, back-to-back updates, or confusing out-of-order notifications. The first defense is simple: disable the trigger while the relevant mutation is pending. The second defense is narrower filtering so you do not freeze unrelated actions unnecessarily.

If you only disable based on the local mutation instance, you may still allow duplicates triggered from another component. Mutation filters help close that gap. You can scope pending checks by `mutationKey`, by exactness, or by predicate-based matching when the application needs more precise duplicate prevention. If a duplicate would be harmful only for the same record, filter on both the logical operation and the target identity.

A practical design principle helps here: disable by intent, not by widget. If “update post 42” is already in flight, disable every control that would issue the same write, regardless of which component renders it.

Hook-local callbacks do not imply request ordering

One of the less intuitive mutation behaviors matters immediately under overlap: when multiple requests are made from one mutation hook instance, `onSuccess` fires only after the latest call. You cannot assume

that submission order and callback order will align neatly. If the user edits a field twice quickly and both submissions race, the mutation hook does not become a per-request event log.

That means your UI should not casually bind important ordering decisions to a shared mutation instance. If the application needs a strict “latest intent wins” policy, implement that policy deliberately. If it needs serialization, prevent overlap. If it needs per-request visibility, observe the mutation cache more broadly rather than treating one hook instance as the authoritative history of all writes.

Overlapping optimistic writes create windows of inconsistency

Optimistic updates amplify concurrency problems because they change the visible state before the server confirms anything. If two optimistic writes target the same resource, the earlier one may settle and trigger invalidation while the later one is still pending. The refetch from that invalidation can then overwrite the newer optimistic patch temporarily. Even if the later write eventually succeeds, the user experiences a confusing flicker or reversal.

This is the classic window of inconsistency. Query cancellation reduces one major race by preventing stale in-flight reads from overwriting optimistic state. It does not eliminate all overlap hazards between multiple writes. Once several optimistic mutations coexist, you need a policy: serialize writes to the same resource, track the newest user intent explicitly, or accept temporary UI instability in exchange for more responsiveness.

The practical guidance is conservative. Reserve cache-level optimism for simple, low-conflict operations. If users can issue rapid, repeated edits to the same record, prefer less aggressive techniques such as local pending affordances plus post-success invalidation.

Cancellation protects optimism, but only partway

Using `cancelQueries` before optimistic writes reduces one major race: stale in-flight reads overwriting the optimistic state. That protection is essential whenever you patch cached query data in `onMutate`. Still, cancellation is not a global concurrency solution. It only addresses reads already in flight for the targeted queries. It does not coordinate multiple mutations, and it does not decide which user intent should win when several writes overlap.

This is why cancellation should be treated as a prerequisite, not as a guarantee. It protects the optimistic patch from old reads. You still need separate decisions about duplicate submits, overlapping edits, and mutation completion ordering.

Retries for writes require server semantics, not optimism

Mutation retry defaults to 0, and that is a good default. Reads are often safe to retry because they are observational. Writes are not. A retry may create a duplicate resource, repeat a side effect, or apply a state change twice. Whether a retry is safe depends on the server contract and the HTTP semantics of the operation, not on client convenience.

At a high level, `PUT` and `DELETE` are generally idempotent, while `POST` and `PATCH` are not guaranteed to be. Even that distinction is only a starting point. An endpoint exposed as `PUT` may still have non-idempotent side effects in a specific domain, and a `POST` endpoint may support idempotency keys or other safeguards. The point is that retry policy belongs to the application protocol, not just to a loading-state preference.

Configure retries for mutations only when the server gives you a strong reason to believe replay is safe. Otherwise, surface the error, preserve enough context for the user to recover, and avoid silently resubmitting commands whose effect is uncertain.

Separate transport failures from semantic conflicts

Not all failures mean the same thing. A timeout, a lost connection, and a stale-write conflict should not all produce the same toast and the same retry button. Transport failures mean the command may not have reached the server or the response may not have reached the client. Validation failures mean the server rejected the payload as submitted. Conflict failures mean the payload was structurally valid, but the state you edited is no longer current.

For true conflict handling, server-supported optimistic concurrency control is far better than blind retry. If the server exposes an ETag for a resource, you can send it back with `If-Match`. If the resource changed since the client last fetched it, the server can reject the update with `412 Precondition Failed`. That gives you a crisp, semantic signal: the write lost a race against newer state.

```
async function updatePostWithVersion(input: {
  id: number
  title: string
  etag: string
}) {
  const response = await fetch(`/api/posts/${input.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
      'If-Match': input.etag,
    },
    body: JSON.stringify({ title: input.title }),
  })

  if (response.status === 412) {
    throw new Error('Write conflict')
  }

  if (!response.ok) {
    throw new Error('Failed to update post')
  }

  return response.json()
}
```

When you receive a conflict like this, do not silently retry. Refetch the resource, surface that the record changed elsewhere, and give the user a merge-or-retry path. Silent retry under conflict destroys trust because it turns the user's stale intent into an overwrite without acknowledgment.

User feedback must reflect state, not just request completion

Many production UX bugs come from feedback surfaces that fire too early or too broadly. A global success toast triggered the moment a mutation promise resolves may be misleading if relevant queries have not reconciled yet. A form that clears itself on success while a refetch later reveals server-side validation adjustments can also feel wrong. The user experiences the UI as inconsistent even if each callback did technically run.

Align feedback granularity with actual certainty. Use button disabling and inline indicators for `isPending`. Use localized success confirmation when the mutation has settled and any required invalidation or refetch is complete. Use error messages that match the failure class: validation near the form fields, conflict messaging with explicit recovery steps, and transient transport errors with a clear retry option only when replay is safe.

The more global the feedback surface, the more careful you need to be. Global toasts should usually signal events with broad confidence, not intermediate local transitions. A banner that says "Saved" while a visible list still shows stale data teaches users to distrust the application.

Design for the smallest trustworthy pending scope

A useful way to keep interfaces responsive without creating noise is to choose the smallest scope that honestly reflects the write. If one row is being updated, show pending state on that row. If one form is blocked, disable that form. Use application-wide indicators only when

the pending work matters across multiple screens or when background activity itself is meaningful to the user.

Mutation filters help you implement that scope precisely. You can filter by `mutationKey`, use `exact` matching when a broader prefix would overreach, or provide a `predicate` for custom grouping. Those filters are what let you disable only the conflicting action rather than freezing half the interface.

A resilient mutation flow under pressure

When hostile conditions are common, a robust write path usually follows a disciplined sequence:

- Accept user intent and classify the operation.
- Prevent harmful duplicates with targeted pending checks.
- Apply local or optimistic feedback only at a scope you can roll back cleanly.
- Cancel conflicting reads before optimistic cache writes.
- Execute the mutation with retry policy derived from server semantics, not convenience.
- Distinguish validation, transport, and conflict failures.
- Reconcile server state with invalidation or direct repair.
- Present feedback only when it matches the application's actual trust level.

This flow uses the earlier tools under stress instead of replacing them. Invalidation still handles broad correctness. Direct cache repair still helps with narrow, authoritative updates. Optimistic techniques still

improve perceived latency when the write is simple enough. The difference is that concurrency, retries, and feedback are now treated as first-class operational constraints rather than as afterthoughts.

The quality of a mutation system is easiest to judge when things go wrong. If duplicate clicks are harmless, overlapping edits are visible rather than silent, stale writes are rejected instead of overwriting newer state, and UI feedback tracks trustworthy state instead of wishful state, the application remains coherent even under load. That coherence is what turns isolated mutation APIs into dependable write-path engineering.

Chapter 7

Scaling Read Patterns: Pagination, Infinite Loading, and Prefetching

List-heavy interfaces fail in subtle ways long before they fail spectacularly: page transitions flicker, cursor histories drift, retained data bloats memory, and “helpful” prefetching quietly turns into bandwidth waste. This chapter treats those failures as design signals, then builds a precise operational model for page-parameter identity, infinite page stacks, bounded retention, and prefetch timing so advanced readers can choose the right read pattern for each navigation surface.

7.1. Paginated Queries as Parameterized Cache Entries

Pagination looks trivial until users stop behaving politely. They change page size after landing on page 7, type a search term, clear it, sort by a different column, open a detail view, then navigate back and expect the exact list state they just left. If you treat pagination as “one list that changes over time,” you end up fighting the cache. TanStack Query uses a different model: each distinct parameter set identifies a distinct cache entry. Page 1 and page 2 are different queries. “Open issues sorted by newest” and “open issues sorted by oldest” are different queries. A search term added to either of them creates another query identity again.

That model sounds stricter than many ad hoc list implementations, but it matches how server state actually behaves. The server does not return an abstract list that you mutate locally page by page. It returns one result for one request shape. TanStack Query makes that request shape explicit through the query key.

A minimal paginated query already shows the pattern clearly:

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

function ProjectsTable({
  page,
  pageSize,
  sort,
  status,
}) {
  const result = useQuery({
    queryKey: [
      'projects',
      { page, pageSize, sort, status },
    ],
    queryFn: () =>
      fetchProjects({
```

```
    page,
    pageSize,
    sort,
    status,
  )),
  placeholderData: keepPreviousData,
})

const { data, isPending, isFetching,
  isPlaceholderData } = result

// render table controls and rows
}
```

The key detail is not the hook call itself. It is the identity encoded in `queryKey`. You are not storing one cache entry named `'projects'` and rewriting it as users paginate. You are creating a family of related entries under the `'projects'` namespace, each one keyed by the parameter object. That distinction drives revisitation behavior, loading behavior, invalidation precision, and how reliably the UI can move between pages without flicker or confusion.

Pagination is a cache-identity problem

A page control is only one source of variability. Real production lists are usually parameterized by at least four dimensions:

- page number or offset,
- page size,
- sort field and sort direction,
- filter or search criteria.

If any of those values changes the server response, it belongs in the query key. Leaving it out produces cache aliasing: two semantically different requests map to the same cache entry. When that happens,

the UI may show rows from the wrong filter set, refetches may seem nondeterministic, and invalidation becomes too broad or too narrow.

A good mental model is: if the backend would consider two requests distinct, your queryKey should consider them distinct too.

TanStack Query requires query keys to be top-level arrays. Inside that array, you can include serializable nested objects. That matters for paginated keys because a structured object lets you encode parameters by name rather than by fragile position. Compare these two approaches:

```
// Positional and easy to misuse
['projects', page, pageSize, sort, status]

// Named and easier to evolve
['projects', { page, pageSize, sort, status }]
```

The second form is usually easier to maintain. Object property order does not affect identity, while array item order does. That deterministic hashing behavior means these two keys are equivalent:

```
['projects', { page: 2, sort: 'name' }]
['projects', { sort: 'name', page: 2 }]
```

By contrast, array position is part of identity, so these are not equivalent:

```
['projects', 2, 'name']
['projects', 'name', 2]
```

For paginated reads, named parameter objects reduce accidental misses when you add or reorder dimensions later. They also make bugs easier to spot during review because the key mirrors the request contract.

Design keys to match user-visible state

Readers often under-specify paginated keys by including only the page number:

```
// Incomplete when filters or sort also change data
```

```
['projects', page]
```

That only works if page is the only factor that changes the response. The moment you add search text, a status filter, or a sort order, the old key shape stops matching user intent. A user expects page 3 of “status=open” to be different from page 3 of “status=closed.” If both requests share the same key, cached results can bleed across states.

A more realistic key often looks like this:

```
const projectsListKey = ({
  page,
  pageSize,
  search,
  status,
  sortBy,
  sortDir,
}) => [
  'projects',
  {
    page,
    pageSize,
    search,
    status,
    sortBy,
    sortDir,
  },
]
```

A small key factory like that is not required by the library, but it helps teams keep key structure consistent across list views, prefetch logic, and invalidation calls. It also makes it harder to accidentally omit one parameter in one code path and include it in another.

Keep the namespace stable and the parameter set explicit. For example, if your product has both a paginated table view and a summary count query, do not overload them under indistinguishable keys. A total count and a page of rows are different resources from the cache’s perspective even if they come from related endpoints.

Separate standard pagination from infinite data shapes

One easy mistake is to think of standard pagination as the first stage of an infinite list. In TanStack Query, they are different models. A paginated `useQuery` entry typically stores one page result. An infinite query stores a structured object with `pages` and `pageParams`. Because the data shapes differ, you should not reuse the same key for both patterns.

That matters even before you reach infinite loading. If a table view uses ordinary page navigation, treat each page as a separately addressable cache entry. Do not manually merge multiple pages into one local cache object while also trying to let TanStack Query manage those pages individually. You end up with two competing sources of truth: the query cache and your hand-assembled page store.

Parameter changes are identity changes

Once you accept that keys define identity, page transitions become mechanically simple. The user clicks “next page.” Your component updates the page state. The derived `queryKey` changes. TanStack Query checks whether that exact key already exists in cache and whether the data is fresh enough under the active policy. If a warm entry exists, the UI can reuse it immediately. If not, a fetch begins for the new key.

That means page navigation is not “refetching the same query with a different argument” in the mental model many developers bring from imperative data loaders. It is subscribing to a different query entry.

This distinction is what makes revisitation predictable. If the user goes from page 2 to page 3 and back to page 2, the cache can serve page 2 directly because page 2 retained its own identity. You do not need to manually preserve it in component state. You only need a key that names it correctly.

Preserving continuity with placeholder data

The strict identity model introduces a practical UI issue. If each page is

a new query, changing pages can cause the view to swing from success to pending on every click. The rows disappear, the layout jumps, and pagination feels slower than the network actually is.

TanStack Query provides a direct solution for this style of list transition through `placeholderData: keepPreviousData`. The example earlier used exactly that option. When the key changes from one page to another, the previous successful result remains visible while the new request is in flight. This gives you continuity without collapsing the cache model into one mutable list.

A fuller example makes the control flow clearer:

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

function fetchProjects(params) {
  const query = new URLSearchParams({
    page: String(params.page),
    pageSize: String(params.pageSize),
    sort: params.sort,
    status: params.status,
  })

  return fetch(`/api/projects?${query.toString()}`)
    .then((res) => res.json())
}

export function ProjectsScreen() {
  const [page, setPage] = useState(0)
  const [pageSize] = useState(20)
  const [sort] = useState('name')
  const [status] = useState('active')

  const {
    data,
    isPending,
    isFetching,
    isPlaceholderData,
  } = useQuery({
    queryKey: [
      'projects',
```

```

    { page, pageSize, sort, status },
  ],
  queryFn: () =>
    fetchProjects({
      page,
      pageSize,
      sort,
      status,
    }),
  placeholderData: keepPreviousData,
})

if (isPending && !data) {
  return <p>Loading projects...</p>
}

return (
  <>
    <table>
      <tbody>
        {data.projects.map((project) => (
          <tr key={project.id}>
            <td>{project.name}</td>
          </tr>
        ))}
      </tbody>
    </table>

    <button
      onClick={() => setPage((old) => Math.max(old - 1, 0))}
      disabled={page === 0}
    >
      Previous
    </button>

    <button
      onClick={() => {
        if (!isPlaceholderData && data.hasMore) {
          setPage((old) => old + 1)
        }
      }}
      disabled={isPlaceholderData || !data.hasMore}
    >
      Next
    </button>

    {isFetching ? <p>Refreshing...</p> : null}
  </>
)

```

```
| )  
| }
```

Two details matter here.

First, `isPending` and `isFetching` are not interchangeable. `isPending` tells you there is no resolved data for the current observer state yet. `isFetching` tells you a request is active, including background work. In a paginated UI with placeholder continuity, you often show the old rows and a smaller “refreshing” indicator rather than replacing the whole table with a loading screen.

Second, `isPlaceholderData` tells you the currently rendered data is transitional. That is operationally important. If the UI enables forward navigation while it is still showing placeholder rows from the previous page, users can outrun the server and create confusing jumps. Guarding the “next” action while placeholder data is active avoids speculative multi-step navigation based on data that does not yet represent the selected page.

When to reset the page index

A page parameter rarely changes alone. Users also change filters, sort order, or page size. That creates a product decision: should the interface preserve the current page number or reset to the first page?

Most list UIs should reset to page 1 when any parameter change alters the meaning of the result set substantially. Search text, status filters, and sort order usually fall into that category. If a user is on page 8 of all projects and switches to “status=archived,” staying on page 8 often yields a sparse or empty result that looks like a bug. The cache is behaving correctly; the UX contract is wrong.

A practical rule is to preserve page only when the user’s navigational intent is still about the same ordered set, just revisited under the same search and filter semantics. Reset page when the parameter change

defines a new collection slice.

Implement that rule in your local state transitions, not by omitting parameters from the key. For example:

```
function onStatusChange(nextStatus) {
  setStatus(nextStatus)
  setPage(0)
}

function onSortChange(nextSort) {
  setSort(nextSort)
  setPage(0)
}
```

That keeps identity honest. The key still fully describes the request, and the page reset reflects UI intent explicitly.

Offset, page number, and backend contracts

The frontend key should mirror the pagination primitive the backend actually uses. If the API is page-number based, encode page numbers. If the API expects offset and limit, encode those. Do not hide one scheme inside another unless you have a stable translation layer and a strong reason to normalize.

For a backend that accepts offset pagination, this is legitimate:

```
useQuery({
  queryKey: [
    'projects',
    { offset, limit, sort, status },
  ],
  queryFn: () =>
    fetchProjects({ offset, limit, sort, status }),
})
```

What matters is consistency. If one screen thinks in pages and another in offsets, but both target the same logical list, they will not reuse cache entries unless their key strategy also aligns. Sometimes that is fine because the experiences are intentionally distinct. Sometimes it silently defeats revisitation and prefetching. Decide deliberately.

Common failure modes

Several bugs recur in paginated systems because they blur identity boundaries.

Under-specified keys. You leave `sort` or `filter` out of the key because the query function closes over those values anyway. The fetch sees the right parameters, but the cache does not. Results collide.

Over-instability. You include values that are not semantically meaningful or that change shape unnecessarily. A new array order or a meaningless wrapper object can create avoidable cache misses. Keep the parameter object minimal and stable.

Single-key pagination. You use `['projects']` for all pages and manually refetch as the page changes. That throws away built-in page revisitation benefits and makes it harder to reason about what is cached.

Manual page merging. You collect multiple paginated responses into local component state while still using `useQuery`. That duplicates server state management and usually signals that the screen actually wants an infinite-query model instead.

Shape collisions across patterns. You reuse the same key for a standard paginated query and an infinite query. The cache cannot safely treat those as interchangeable because the stored data structures differ.

A disciplined paginated flow

A robust paginated screen usually follows a short chain of decisions:

- Represent the user's current list state as explicit parameters.
- Encode every server-visible parameter in the query key.
- Reset page index when a parameter change defines a new result set.

- Use `placeholderData: keepPreviousData` when preserving visual continuity matters.
- Use `isPlaceholderData` to prevent controls from racing ahead of the new page.
- Keep standard pagination as many related cache entries, not one hand-mutated cache blob.

Once you work this way, pagination stops being a fragile UI widget and becomes a predictable cache segmentation strategy. Each page and filter combination has a name, the cache can answer revisits precisely, and navigation logic stays aligned with the actual request contract rather than with a locally improvised notion of “current list state.”

7.2. Implementing Infinite Queries with Page-Param Contracts

Infinite loading changes the user experience without changing a basic truth: the client is still fetching discrete pages from the server. What changes is where those pages live and how you advance from one to the next. A paginated `useQuery` flow stores each page under a separate cache identity. An infinite query stores multiple fetched pages inside one cache entry, preserving their order and the parameters that produced them. If you blur those models together, you either lose revisit behavior from standard pagination or you break the continuity that makes infinite scrolling useful.

A concrete cursor-based example makes the structure visible before the terminology gets dense:

```
import { useInfiniteQuery }  
  from '@tanstack/react-query'  
  
function fetchProjects(cursor) {
```

```
const qs = new URLSearchParams()

if (cursor !== null) {
  qs.set('cursor', cursor)
}

return fetch(`/api/projects?${qs.toString()}`)
  .then((res) => res.json())
}

export function ProjectsFeed() {
  const {
    data,
    fetchNextPage,
    hasNextPage,
    isFetchingNextPage,
    status,
  } = useInfiniteQuery({
    queryKey: ['projects-feed'],
    queryFn: ({ pageParam }) =>
      fetchProjects(pageParam),
    initialPageParam: null,
    getNextPageParam: (lastPage) =>
      lastPage.nextCursor,
  })

  if (status === 'pending') {
    return <p>Loading feed...</p>
  }

  return (
    <>
    {data.pages.map((page, pageIndex) => (
      <div key={pageIndex}>
        {page.items.map((project) => (
          <article key={project.id}>
            {project.name}
          </article>
        ))}
      </div>
    ))}
    <button
      onClick={() => fetchNextPage()}
      disabled={!hasNextPage || isFetchingNextPage}
    >
      {isFetchingNextPage
        ? 'Loading more...'
      }
    </button>
  )
}
```

```
: hasNextPage
? 'Load more'
: 'No more results'}
</button>
</>
)
}
```

This example already shows the core contract. The query function does not invent the next cursor. It receives `pageParam`. The server response returns a cursor for the next fetch. `getNextPageParam` extracts that cursor. If the response has no further cursor and returns `null` or `undefined`, the infinite sequence ends. The client is not guessing navigation; it is interpreting it.

Infinite query data is structurally different

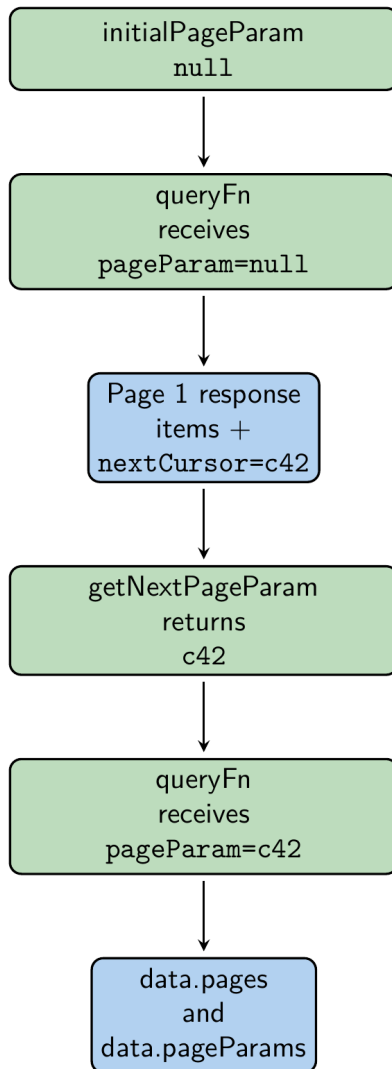
The most important shift is data shape. A standard query usually gives you a single value for one request. An infinite query gives you an object with two coordinated arrays:

- `data.pages`, the ordered page payloads,
- `data.pageParams`, the ordered parameters used to fetch them.

That shape is not an implementation detail you can ignore. It is the defining structure of the cache entry. Each element in `pages` corresponds to an element in `pageParams`. If the first page was fetched with `null`, the first entry in `pageParams` is `null`. If the next page used cursor `"c42"`, the next `pageParams` entry is `"c42"`.

That pairing is what allows TanStack Query to preserve ordered page history inside one query result instead of scattering each page into separate entries. It also explains why you must not reuse the same query key between `useQuery` and `useInfiniteQuery`: the cached data structures are fundamentally different.

The minimum option set reflects that structural difference. An infinite query needs `queryKey`, `queryFn`, and `initialPageParam`. In practice, it also usually needs `getNextPageParam`. If you support loading in both directions, you also need `getPreviousPageParam`.



The page-param contract

The phrase “page-param contract” is not decorative. It names the exact boundary where correctness lives or fails. For infinite queries, the backend and frontend must agree on three things:

- what value starts traversal,
- what value fetches the next or previous slice,
- what response field indicates continuation or termination.

For a cursor API, the contract often looks like this:

- start with `null` or another sentinel value,
- fetch a page using that cursor,
- read `nextCursor` from the response,
- stop when `nextCursor` is `null`.

That contract is carried directly by the hook options:

```
useInfiniteQuery({
  queryKey: ['projects-feed'],
  queryFn: ({ pageParam }) =>
    fetchProjects(pageParam),
  initialPageParam: null,
  getNextPageParam: (lastPage) =>
    lastPage.nextCursor,
})
```

Notice what is absent: the current page number stored in local UI state, a hand-maintained cursor stack, or code that increments some client-side counter and hopes the server agrees. The server response is the authority for continuation. If you derive the next parameter from UI state instead of from returned page data, you sever the client from the backend’s pagination semantics.

That mistake is especially costly for mutable collections. If new rows are inserted while the user is scrolling, a cursor returned by the backend can still preserve traversal intent. A client-computed offset often cannot.

Offset and cursor are different contracts

TanStack Query gives you one `useInfiniteQuery` surface, but the meaning of `pageParam` depends entirely on your backend model. Offset-based and cursor-based APIs both fit the hook, yet they fail in different ways.

An offset-based implementation might look like this:

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

function fetchAuditPage(offset) {
  const qs = new URLSearchParams({
    offset: String(offset),
    limit: '50',
  })

  return fetch(`/api/audit?${qs.toString()}`)
    .then((res) => res.json())
}

useInfiniteQuery({
  queryKey: ['audit-log'],
  queryFn: ({ pageParam }) =>
    fetchAuditPage(pageParam),
  initialPageParam: 0,
  getNextPageParam: (lastPage, pages) =>
    lastPage.hasMore ? pages.length * 50 : undefined,
})
```

This is reasonable when the dataset is append-only or relatively stable and your backend exposes offsets directly. It is also easy to understand operationally. But offsets embed an assumption that record positions remain meaningful while the user is paging. In insertion-heavy feeds, that assumption breaks quickly. Records can shift between offsets, producing duplicates or gaps.

Cursor-based APIs push more work to the server and usually expose opaque tokens instead of arithmetic positions. The benefit is stronger traversal correctness for changing datasets. The trade-off is that you now depend completely on the backend's cursor semantics. If the server returns inconsistent cursors, the client cannot recover by clever local logic.

Choose the contract that matches the data. For a social feed or event stream, cursor semantics are often safer. For a stable administrative table or archival export viewer, offsets may be sufficient and simpler.

Bidirectional loading adds real complexity

Forward-only infinite lists are common because they align with feed-like experiences. Bidirectional traversal is more demanding. Once users can load older and newer segments around an anchor point, you must define both continuation directions explicitly.

TanStack Query supports this through `getPreviousPageParam` in addition to `getNextPageParam`. A minimal bidirectional setup looks like this:

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

function fetchMessages(cursor) {
  const qs = new URLSearchParams()

  if (cursor !== null) {
    qs.set('cursor', cursor)
  }

  return fetch(`/api/messages?${qs.toString()}`)
    .then((res) => res.json())
}

useInfiniteQuery({
  queryKey: ['conversation', { roomId: 'ops' }],
  queryFn: ({ pageParam }) =>
    fetchMessages(pageParam),
  initialPageParam: null,
  getNextPageParam: (lastPage) =>
```



```
lastPage.nextCursor,  
getPreviousPageParam: (firstPage) =>  
  firstPage.previousCursor,  
})
```

The correctness burden rises immediately. Your backend must produce continuation tokens that remain coherent when moving in either direction. Your UI must preserve list anchoring so prepending older pages does not make the viewport jump unexpectedly. Your cache now represents a window through an ordered sequence rather than a simple append-only stack.

Once you add bounded page retention later with `maxPages`, bidirectional correctness becomes even stricter because page eviction depends on fetch direction. If you expect two-way traversal, both previous and next parameter derivations must be exact.

Do not flatten away the structure too early

Developers often write this immediately after adopting `useInfiniteQuery`:

```
const items = data.pages.flatMap((page) => page.items)
```

That flattening is fine for rendering, but only if you still treat page boundaries as meaningful elsewhere. If you discard page structure completely, you lose context needed for several operational concerns:

- deriving subsequent page params from the correct page,
- preserving scroll anchors,
- debugging duplicates and gaps,
- trimming or refetching with awareness of boundaries.

A healthy pattern is to flatten only at the presentation edge. Keep the

original pages and pageParams intact in your mental model and in any nontrivial list logic.

Placeholder and initial data must match the infinite shape

Standard paginated queries can often use a single page value as placeholder data. Infinite queries cannot. If you provide `placeholderData` or `initialData`, it must match the infinite-query structure with both `pages` and `pageParams`.

This is structurally wrong:

```
// Wrong: flat array does not match infinite-query shape
placeholderData: []
```

This at least matches the required container shape:

```
placeholderData: {
  pages: [],
  pageParams: [],
}
```

If you seed one initial page, both arrays need corresponding entries. The shape contract matters because TanStack Query and your components will read the result as an infinite-query object, not as an arbitrary convenience payload.

Operational flow for append and prepend

A practical infinite-query interaction follows a short but strict sequence.

For append-driven loading:

1. Set `initialPageParam`.
2. Fetch the first page through `queryFn`.
3. Store the result in `data.pages[0]` and the input in `data.pageParams[0]`.

4. Extract the next token with `getNextPageParam`.
5. Expose `hasNextPage` based on whether that derivation returns a value.
6. Call `fetchNextPage` only when a next page exists and a next-page fetch is not already active.

For prepend-driven loading, the same logic applies in reverse with `getPreviousPageParam` and `fetchPreviousPage`. The important discipline is that the next or previous token always comes from fetched page data, not from a guessed local counter.

That is why some hypermedia-style APIs are a natural fit. If a response exposes navigation semantics such as next and previous links, you can interpret those protocol-level relations into page parameters. Whether the token is an opaque cursor, an offset embedded in a URL, or a typed navigation link, the client-side contract remains the same: consume the provided continuation primitive, do not synthesize your own.

Choosing infinite queries deliberately

Infinite loading is attractive because it feels smooth and modern, but it is not just pagination with a different control. Use it when users benefit from continuous accumulation of context: feeds, activity streams, search results meant for exploration, chat history, and event timelines. Avoid it when users need deterministic jumps to arbitrary pages, exact positional awareness, or strong revisitation to named page numbers. Those are standard pagination strengths.

A good decision signal is whether the product language is about traversal or about addressability. “Load more” and “show older messages” point toward infinite queries. “Go to page 17” and “return to page 4” point toward parameterized paginated queries.

Common failure modes

Most infinite-query bugs are contract bugs, not rendering bugs.

Cursor drift. `getNextPageParam` extracts a token that does not actually correspond to the backend's next slice. The result is duplicate pages, skipped records, or termination too early.

UI-derived continuation. You compute the next page param from a local counter instead of the server response. This often survives testing on static data and then fails on mutable collections.

Shape mismatch. You seed `placeholderData` or `initialData` with a flat list or a single page object instead of `{ pages, pageParams }`.

Key collision with standard queries. You reuse a key already used by `useQuery`. The cache now has incompatible expectations for the same identity.

Premature flattening. You erase page boundaries in application logic and make it harder to reason about cursors, anchoring, and retention.

When the model is correct, `useInfiniteQuery` gives you one cache entry that behaves like an ordered stack of server-sourced pages. The stack grows through explicit page parameters, not through accidental state accumulation, and every additional fetch remains tied to the continuation semantics the backend actually guarantees.

7.3. Bounding Infinite List Retention with max-Pages and Cache Lifecycle Policy

Infinite queries solve a navigation problem by accumulating pages into one ordered result, but accumulation has a cost. A feed that feels effortless for the first five screens can become surprisingly heavy after twenty minutes of scrolling. The browser now holds many page payloads, Re-

act has more data to diff, and any background synchronization has a larger structure to reason about. If you never place a bound on retained pages, memory behavior becomes an accident of user patience.

A concrete configuration shows where the control point lives:

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

function fetchFeed(cursor) {
  const qs = new URLSearchParams()

  if (cursor !== null) {
    qs.set('cursor', cursor)
  }

  return fetch(`/api/feed?${qs.toString()}`)
    .then((res) => res.json())
}

export function Feed() {
  const {
    data,
    fetchNextPage,
    hasNextPage,
    isFetchingNextPage,
  } = useInfiniteQuery({
    queryKey: ['feed'],
    queryFn: ({ pageParam }) =>
      fetchFeed(pageParam),
    initialPageParam: null,
    getNextPageParam: (lastPage) =>
      lastPage.nextCursor,
    maxPages: 5,
    staleTime: 30_000,
    gcTime: 10 * 60 * 1000,
  })

  const items = data?.pages.flatMap((page) => page.items)
  ?? []

  return (
    <>
      {items.map((item) => (
        <article key={item.id}>{item.title}</article>
      ))}

      <button
        onClick={() => fetchNextPage()}
      />
    </>
  )
}
```

```
        disabled={!hasNextPage || isFetchingNextPage}
      >
        Load more
      </button>
    </>
  )
}
```

This configuration does three different jobs. `maxPages: 5` bounds how many pages stay inside the infinite query result. `staleTime: 30_000` defines how long the data remains fresh before TanStack Query considers background refetching appropriate. `gcTime: 10 * 60 * 1000` controls how long the entire inactive query stays in the cache before garbage collection removes it. Those levers interact, but they do not mean the same thing. Many retention bugs come from treating them as interchangeable.

Three independent policy surfaces

You need to separate three concerns cleanly.

In-entry page retention. `maxPages` limits the number of pages stored inside one infinite query result. If `maxPages` is undefined or 0, page retention is unbounded. The query entry can keep growing as the user fetches more pages.

Freshness. `staleTime` determines whether cached data is still considered fresh. Fresh data can be reused immediately without an automatic refetch. Stale data may still render immediately, but TanStack Query is free to synchronize it in the background based on the active triggers you configured elsewhere.

Whole-entry lifetime. `gcTime` determines how long an inactive query remains in cache at all. The default is 5 minutes unless you override it. Once the query has no observers and the garbage-collection timer expires, the entire entry is removed, including every retained page.

That means a bounded infinite list can still feel instant during short navigations away and back if the query remains in cache and within freshness expectations. It also means a carefully retained five-page window disappears entirely after enough idle time if the entry becomes inactive and `gcTime` elapses.

What `maxPages` actually removes

When `maxPages` is reached and you fetch a new page, TanStack Query removes either the first or last page depending on fetch direction. In a forward-only feed, fetching the next page pushes the retained window forward and typically drops the oldest retained page. In a reverse-loading history view, fetching a previous page can drop the newest retained page at the other end.

This matters because `maxPages` is not “compression.” It is eviction. You are explicitly choosing to forget part of the local traversal history in exchange for bounded memory use.

For a feed, that is often acceptable. Users usually care more about continuing downward smoothly than about instantly revisiting content many screens above. For a legal document viewer, audit trail explorer, or support timeline where users move back and forth around a point of interest, directional eviction may be actively harmful.

A feed and a document viewer want different policies

Consider two interfaces that both use `useInfiniteQuery`.

A social-style feed loads newer content as the user moves downward. The user rarely jumps back thirty pages, and if they do, a refetch is tolerable. Here, `maxPages: 5` or `maxPages: 10` may be a strong default. It limits memory growth and narrows the amount of data that background sync logic touches.

A bidirectional document or conversation viewer is different. Users

often scroll up to recover context, then back down to the current point. If you set a small `maxPages`, you may evict exactly the pages the user expects to revisit a few seconds later. Once that happens, the UI can still function, but it no longer behaves like a stable local window. It behaves like a moving aperture that refetches context as the user oscillates.

The technical rule is straightforward: small `maxPages` works best when revisit probability to distant pages is low and the fetch cost of reacquiring them is acceptable. Increase `maxPages` when the interface depends on short-term backtracking or strong scroll-restoration fidelity.

Bounded retention depends on correct page-param contracts

Once you allow page eviction, page-param derivation becomes more than a fetch-next convenience. The documentation explicitly requires that if `maxPages > 0`, `getNextPageParam` and `getPreviousPageParam` must be defined properly to support fetching in both directions after eviction.

That warning is easy to underestimate. Suppose you keep only five pages in a bidirectional query. The user loads older pages, then newer pages again. If your previous and next parameter functions do not mirror backend semantics exactly, the retained window can drift. You may fetch a page that overlaps with the current first page, skip a segment entirely, or fail to reconstruct the correct neighboring page after one end has been evicted.

A forward-only feed is simpler because only one continuation direction matters operationally. Even there, the contract must remain exact: if the backend cursor semantics are unstable, dropping old pages makes recovery more visible because there is less retained history to mask errors.

Retention inside the entry versus retention of the entry

A common debugging mistake is to blame `maxPages` for behavior actually caused by `gcTime`, or vice versa.

Imagine this timeline:

1. The user loads eight pages of a feed.
2. `maxPages: 5` trims the in-memory result to the most recent five retained pages.
3. The user opens a detail screen.
4. The feed query becomes inactive.
5. Two minutes later, the user navigates back.
6. The query is still in cache, so the retained five-page window can be reused.

Now change one variable. The user spends twelve minutes on the detail screen, and `gcTime` is set to five minutes. When they return, the infinite query entry is gone entirely. You do not have “five old pages” or “one stale page.” You have nothing cached for that entry.

That distinction is operationally important when product teams complain that the app “forgets scroll state.” Sometimes the problem is page eviction inside an active cache entry. Sometimes the entry aged out completely while inactive. The fix depends on identifying which lifecycle boundary was crossed.

Freshness changes how revisits feel

`staleTime` affects perceived continuity more than many teams expect. If a user revisits an infinite query while the retained pages are still fresh, the interface can reuse them immediately without automatic refetch pressure. If the data is stale, the same retained pages may still render first, but background synchronization can begin right away.

For volatile feeds, that may be desirable. For expensive, mostly static archives, it can create needless work.

Use freshness to encode how much change the user expects between visits. A fast-moving activity stream often benefits from a short `staleTime`. A historical viewer or catalog browser may deserve a longer one. The key is to decide freshness separately from size. A five-page retained window can be very fresh or very stale. Those are different questions.

Observe the hidden work

Bounded retention is partly about memory, but it also affects network and rendering work. As users page through an infinite list, background fetches continue to happen even when the visual state looks calm. You can expose that work during development with existing query instrumentation.

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

function FeedDebugPanel() {
  const queryClient = useQueryClient()

  const {
    isFetchingNextPage,
  } = useInfiniteQuery({
    queryKey: ['feed'],
    queryFn: ({ pageParam }) =>
      fetchFeed(pageParam),
    initialPageParam: null,
    getNextPageParam: (lastPage) =>
      lastPage.nextCursor,
    maxPages: 5,
  })

  useEffect(() => {
    const count = queryClient.isFetching()
    console.log('active fetches:', count)
  })
}
```

```
return (
  <p>
    next page fetch active:
    {isFetchingNextPage ? 'yes' : 'no'}
  </p>
)
```

`isFetchingNextPage` tells you whether the current observer is actively loading another page. `queryClient.isFetching()` gives you a broader count of active fetches across the client, including background activity. During tuning, these signals help you distinguish “the list is large” from “the list is synchronizing aggressively” from “the list was dropped from cache and had to restart.”

If you log these counters while repeatedly scrolling, navigating away, and returning, you can watch the effect of different `maxPages`, `staleTime`, and `gcTime` combinations without guessing.

Choosing a `maxPages` value

There is no universal good value for `maxPages`. Choose it by combining product behavior and payload economics.

Start with these questions:

- How deep do typical sessions go before the user leaves?
- How often do users reverse direction shortly after scrolling?
- How expensive is one page to render and hold in memory?
- How expensive is it to refetch an evicted page?
- Does the dataset change so quickly that old retained pages are unlikely to stay useful anyway?

If each page holds a few short text rows, retaining ten or twenty pages

may be cheap. If each page contains media cards, expanded metadata, or large embedded previews, even five pages may be substantial. Measure with realistic payloads, not synthetic toy data.

A practical strategy is to begin with a conservative bound and test three behaviors: long continuous scroll, brief navigation away and back, and short-range backtracking within the list. If one of those feels wrong, change the policy that actually governs it. Increase `maxPages` for backtracking problems. Increase `gcTime` for navigation-gap loss. Increase `staleTime` when the query is refetching more eagerly than the experience warrants.

Scroll anchoring and user expectations

You can configure retention perfectly and still ship a list that feels broken if scroll anchoring is not compatible with page eviction. When early pages are removed, the DOM structure above the viewport changes. If your UI relies on those nodes for stable positioning, the visible content may jump even though the cache is behaving as configured.

This is one reason to think of `maxPages` as a product decision rather than a raw performance knob. The question is not only “How much memory can I save?” It is also “What local history does the user expect the interface to preserve without movement or delay?” A scrolling feed can often tolerate losing far-off content. A transcript viewer centered on a specific message often cannot.

Common anti-patterns

Several mistakes recur when teams first tune infinite-list retention.

- *Leaving retention unbounded by default.* If `maxPages` stays undefined everywhere, the longest session becomes your effective retention policy.
- *Using one policy for every list.* A tiny reference list and a media-

heavy discovery feed should not automatically share the same page cap or cache lifetime.

- *Extending gcTime to hide weak retention design.* A very long gcTime keeps inactive entries around longer, but it does nothing to reduce memory pressure while the query is active and accumulating pages.
- *Treating staleTime as a memory control.* Freshness affects refetch timing, not page count. A query can be fresh and still too large.
- *Ignoring reverse traversal after eviction.* In bidirectional experiences, incorrect `getPreviousPageParam` logic may only become obvious once older pages are dropped and must be recovered from the server.

A practical tuning loop

You get better results if you tune retention with a repeatable operational loop instead of intuition alone:

1. Pick a realistic dataset and page size.
2. Set an explicit `maxPages` rather than leaving it unbounded.
3. Choose `staleTime` based on data volatility, not memory concerns.
4. Choose `gcTime` based on expected navigation gaps.
5. Scroll deeply enough to force page rotation.
6. Navigate away and back before and after the `gcTime` window.
7. Watch `isFetchingNextPage` and `queryClient.isFetching()` to see hidden work.

8. Validate whether the retained window matches real revisit behavior.

When you do that, `maxPages`, `staleTime`, and `gcTime` stop feeling like abstract cache knobs. They become a concrete policy for how much list history stays local, how trustworthy that history remains over time, and how much effort the client spends preserving the illusion of an endless surface.

7.4. Prefetching for Latency Hiding and Navigation Smoothness

Prefetching is attractive because it can make navigation feel instant, but it is easy to turn it into invisible waste. If you fetch everything a user might possibly want next, you burn bandwidth, fill the cache with entries that never get observed, and still miss the one request that actually matters because its identity or freshness policy did not match the eventual screen. The useful mental model is not “make things faster in the background.” It is “warm the exact cache entry the user is likely to need soon enough to matter, but not so aggressively that the warm data expires or pollutes the cache before it is used.”

A simple paginated example captures the mechanics. Suppose a user is reading page 4 of a product list. Hovering the “Next” button is a strong intent signal that page 5 may be needed. You can prefetch exactly that future entry:

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

function fetchProducts({ page, pageSize, sort }) {
  const qs = new URLSearchParams({
```

```

    page: String(page),
    pageSize: String(pageSize),
    sort,
  })

  return fetch(`/api/products?${qs.toString()}`)
    .then((res) => res.json())
}

export function ProductsPage() {
  const queryClient = useQueryClient()
  const [page, setPage] = useState(4)
  const pageSize = 20
  const sort = 'name'

  const { data, isFetching } = useQuery({
    queryKey: ['products', { page, pageSize, sort }],
    queryFn: () =>
      fetchProducts({ page, pageSize, sort }),
    placeholderData: keepPreviousData,
    staleTime: 30_000,
  })

  const nextPage = page + 1

  function prefetchNextPage() {
    queryClient.prefetchQuery({
      queryKey: [
        'products',
        { page: nextPage, pageSize, sort },
      ],
      queryFn: () =>
        fetchProducts({
          page: nextPage,
          pageSize,
          sort,
        }),
      staleTime: 30_000,
    })
  }

  return (
    <>
    <button
      onMouseEnter={prefetchNextPage}
      onFocus={prefetchNextPage}
      onClick={() => setPage(nextPage)}
      disabled={!data?.hasMore}
    />
  )
}

```

```
>  
Next  
</button>  
  
{isFetching ? <p>Refreshing...</p> : null}  
</>  
)  
}
```

This code illustrates three rules that carry through every prefetch strategy. First, the prefetched query must use the same `queryKey` structure that the consuming `useQuery` will use later. Second, the prefetched `queryFn` must fetch the same resource contract, not a close approximation. Third, the prefetch call is speculative. `queryClient.prefetchQuery` returns `Promise<void>`, not the data itself, and it does not throw errors. If you need data now or need thrown errors for control flow, you use `fetchQuery` instead.

Prefetching is cache warming, not alternate data loading

That distinction matters because prefetching should not create a second, parallel state machine in your components. You do not prefetch into local state and then manually hand it to the screen. You warm the query cache under the exact identity the screen will later observe. If the user reaches that screen while the data is still in cache and fresh enough, the UI benefits. If not, the normal query path still works.

This is why query identity discipline matters so much. A prefetch with the wrong page number, omitted filter values, or a different sort parameter is not harmless. It creates false confidence: network activity happened, but the eventual screen still cold-loads because the warmed entry does not match the mounted key.

For paginated UIs, adjacency-based prefetch is often the highest-value starting point. Page $n + 1$ is a much better prediction than some unrelated route the user might eventually visit. For detail views, hover or focus on a specific link is an even stronger intent signal because the

user has already pointed at one concrete resource.

Understand what prefetch freshness actually means

Prefetch freshness decisions use `staleTime`. If the target cache entry already exists and is still fresh enough, TanStack Query can skip the network request during prefetch. If the target is stale, the prefetch can refresh it. That sounds simple, but there is an important scope rule: a `staleTime` passed to `prefetchQuery` affects the prefetch operation itself, not the future consumer hook.

That means this is a valid but subtle configuration:

```
queryClient.prefetchQuery({
  queryKey: ['products', { page: 5 }],
  queryFn: () => fetchProducts({ page: 5 }),
  staleTime: 60_000,
})
```

If the mounted `useQuery` for page 5 does not also use a matching freshness policy, the prefetched result may still mount as stale and refetch immediately. Sometimes that is acceptable. Sometimes it erases the user-perceived benefit of prefetching. The fix is not to prefetch more aggressively. The fix is to align the consuming query's policy with the reason you bothered warming it.

A prefetched entry can still disappear

Prefetching does not pin data in memory forever. If no observer mounts a prefetched query, it may still be garbage-collected later according to `gcTime`. That is one of the clearest examples of why speculative work needs policy discipline. A hover-triggered prefetch on a busy page can generate many short-lived entries that the user never consumes. If `gcTime` is long, those entries linger and crowd the cache. If `gcTime` is short, they may evaporate before the user clicks, making the prefetch ineffective.

You should treat that as a tuning problem, not a reason to avoid

prefetching entirely. Match the trigger to the likely decision horizon. Hover-triggered prefetch is usually appropriate because the navigation, if it happens, tends to follow within seconds. Route-level prefetch for a future workflow step often needs a longer confidence window and more careful payload sizing.

Route-level, hover-level, and adjacency-based triggers

Most prefetch triggers fall into three useful categories.

- **Route-level prediction.** You know that a transition is likely because the user has entered a workflow or a router has enough context to predict the next screen. This can work well for moderate-size payloads with strong reuse probability. It is weaker when the next step is ambiguous or highly personalized.
- **Interaction-driven signals.** Hover, focus, and similar UI intent cues are strong because they are concrete and immediate. They usually justify narrow, precise prefetches such as one detail record or one neighboring paginated result.
- **Adjacency-based warming.** The current screen implies a small set of likely next reads: the next page of results, the previous page, or the first page of a related detail screen. This often produces the best value-per-byte because the prediction is local and bounded.

The common failure mode across all three is broadening the scope too early. If you prefetch whole neighborhoods of possible states instead of one carefully chosen target, bandwidth costs climb faster than perceived performance improves.

Infinite-list prefetch is not just “load more early”

Infinite queries already have a notion of fetching the next page through `fetchNextPage`. Prefetch adds a different question: should you warm

future pages before the user explicitly asks for them? Often the right answer is yes, but only near the viewport and only for a limited number of pages.

By default, `queryClient.prefetchInfiniteQuery` only fetches the first page. That behavior is useful for warming an infinite list route before it mounts, because the first page usually provides the initial visual response. Prefetching additional pages requires a `pages` option plus `getNextPageParam`. That extra work should be justified. Two warmed pages may help a media-rich discovery surface avoid an immediate second spinner after mount. The same choice may be wasteful for an administrative log where most users scan only the top screen.

A route-level first-page prefetch can look like this:

```
queryClient.prefetchInfiniteQuery({
  queryKey: ['feed'],
  queryFn: ({ pageParam }) =>
    fetchFeed(pageParam),
  initialPageParam: null,
  getNextPageParam: (lastPage) =>
    lastPage.nextCursor,
  staleTime: 15_000,
})
```

If you have evidence that warming two pages materially improves the interaction, extend the prefetch deliberately rather than by habit:

```
queryClient.prefetchInfiniteQuery({
  queryKey: ['feed'],
  queryFn: ({ pageParam }) =>
    fetchFeed(pageParam),
  initialPageParam: null,
  getNextPageParam: (lastPage) =>
    lastPage.nextCursor,
  pages: 2,
  staleTime: 15_000,
})
```

The gain is smoothness at the top of the list. The cost is extra transfer and a larger speculative cache footprint. On slow or metered networks,

that trade-off often favors first-page-only warming unless the second page is very likely to be consumed immediately.

IntersectionObserver is the right primitive for near-viewport triggers

For scroll-driven interfaces, manual scroll handlers often become noisy and expensive because they poll geometry too frequently. The browser already provides a better trigger for “this element is close enough to matter” through `IntersectionObserver`. It is asynchronous, designed for visibility-related tasks such as lazy loading and infinite scrolling, and lets the browser optimize detection.

A sentinel-based setup gives you a practical way to trigger either `fetchNextPage` or a `prefetch` as the user approaches the end of the currently rendered content:

```
import { useEffect, useRef } from 'react'
import { useInfiniteQuery } from '@tanstack/react-query'

export function Feed() {
  const sentinelRef = useRef(null)

  const {
    fetchNextPage,
    hasNextPage,
    isFetchingNextPage,
  } = useInfiniteQuery({
    queryKey: ['feed'],
    queryFn: ({ pageParam }) =>
      fetchFeed(pageParam),
    initialPageParam: null,
    getNextPageParam: (lastPage) =>
      lastPage.nextCursor,
  })

  useEffect(() => {
    const node = sentinelRef.current
    if (!node) return

    const observer = new IntersectionObserver(
      (entries) => {
        const entry = entries[0]
```

```
    if (
      entry.isIntersecting &&
      hasNextPage &&
      !isFetchingNextPage
    ) {
      fetchNextPage()
    }
  },
  {
    root: null,
    rootMargin: '300px 0px',
    threshold: 0,
  }
)

observer.observe(node)

return () => observer.disconnect()
}, [fetchNextPage, hasNextPage, isFetchingNextPage])

return <div ref={sentinelRef} />
}
```

The key control here is `rootMargin`. A positive bottom margin such as `'300px 0px'` tells the browser to trigger before the sentinel enters the viewport. That gives your fetch time to complete before the user hits the visual boundary. Larger margins start work earlier and smooth slow networks, but they also increase the chance of fetching data the user never reaches. `threshold` is usually left simple for this use case because you care about proximity, not exact overlap percentages.

You can use the same sentinel pattern to prefetch rather than fetch immediately when the screen design separates “warm soon” from “append now.” The choice depends on whether the content should expand automatically or only after a discrete user action.

Helper hooks and Suspense-oriented setups

TanStack Query also provides `usePrefetchQuery` and `usePrefetchInfiniteQuery`. These are especially useful in React trees where you want to kick off warming before a Suspense

boundary is crossed. The same identity rule still governs them: the warmed query must match the later consumer exactly. If your route transition mounts a different key shape than the prefetch helper used, you get no practical benefit.

The helper hooks do not change the underlying discipline. They simply move prefetch intent into a React-friendly form that composes well with component lifecycles.

Choose triggers by intent confidence

A useful way to judge whether a prefetch is worth it is to ask two questions together: how likely is the user to consume this data soon, and how expensive is it to fetch unnecessarily? That gives you a simple decision matrix:

- High confidence, low-to-moderate payload: prefetch aggressively.
- High confidence, heavy payload: prefetch narrowly, perhaps one page or one record.
- Low confidence, small payload: prefetch only if the interaction is frequent and timing-sensitive.
- Low confidence, heavy payload: usually skip prefetch and optimize the on-demand request instead.

This is why adjacency-based page warming often works well. The next page in a sequence has moderate-to-high intent confidence and a bounded payload. By contrast, shotgun prefetching all possible branches of a dashboard often has low confidence and high cumulative transfer cost.

Common anti-patterns

Several patterns make prefetching look sophisticated while quietly reducing its value.

- **Mismatched keys.** You prefetch one key shape and mount another. The app still feels cold because the warmed entry is never observed.
- **Ignoring consumer freshness policy.** The prefetch uses one `staleTime`, but the eventual query mounts with a much shorter one and refetches immediately.
- **Aggressive infinite prefetch depth.** Warming many pages of an infinite list can transfer far more data than users will read, especially on mobile connections.
- **Using prefetch where the real problem is payload size.** If each request is too large, speculative fetching only hides the issue temporarily.
- **Treating prefetch success as guaranteed.** `prefetchQuery` and `prefetchInfiniteQuery` do not throw and do not return data. They are speculative warm-up calls, not control-flow checkpoints.

Validate prefetch by user-visible outcome

A prefetch strategy is only successful if it changes what the user feels at the moment of navigation or continuation. Watch the actual transition: does the target screen mount with warm data, or does it still show a pending state? Does a near-viewport infinite list continue smoothly, or do users still outrun the fetch? Do prefetched entries survive long enough to be used, or does `gcTime` reclaim them before the interaction happens?

When you align trigger timing, query identity, `staleTime`, and `gcTime`, prefetching becomes a precise latency-hiding tool. It warms the next

likely read without pretending that every possible future state deserves immediate background work.

Chapter 8

Server Rendering, Hydration, Offline Resilience, and Production Patterns

The most expensive TanStack Query mistakes rarely appear in toy examples; they emerge when preloaded data is hydrated twice, when caches leak across requests, when persisted state outlives its truthfulness, or when offline writes return in the wrong order. This chapter treats those situations as first-class architectural problems and shows how to design SSR, persistence, and synchronization policies that stay reliable under real production pressure.

8.1. Implementing Server Prefetch, Dehydration, and Client Hydration

Server rendering only pays off if the browser can continue from the server's work instead of repeating it. If your server fetches route data, renders HTML, and then the client immediately issues the same queries again, you add origin load, increase network chatter, and often produce a subtle flicker as hydrated markup is replaced by a client refetch result. TanStack Query solves that handoff with a precise cache transfer pipeline: create a request-scoped server cache, prefetch the queries needed for the initial render, dehydrate that cache into a serializable payload, then hydrate a browser-side cache before query hooks begin observing.

A minimal implementation has four moving parts: a `QueryClient` for the server request, prefetch calls such as `queryClient.prefetchQuery()`, a dehydrated snapshot created by `dehydrate()`, and a client tree wrapped with `HydrationBoundary`. The exact framework integration point differs across React runtimes, but the dataflow is stable.

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

async function getPost(id) {
  const res = await fetch(
    `https://example.com/api/posts/${id}`
  )
  if (!res.ok) {
    throw new Error('Failed to fetch post')
  }
  return res.json()
}
```

```
export async function renderRequest(postId) {
  const queryClient = new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 60 * 1000,
      },
    },
  })

  await queryClient.prefetchQuery({
    queryKey: ['post', postId],
    queryFn: () => getPost(postId),
  })

  const dehydratedState = dehydrate(queryClient)

  return { dehydratedState }
}

function PostView({ postId }) {
  const { data } = useQuery({
    queryKey: ['post', postId],
    queryFn: () => getPost(postId),
  })

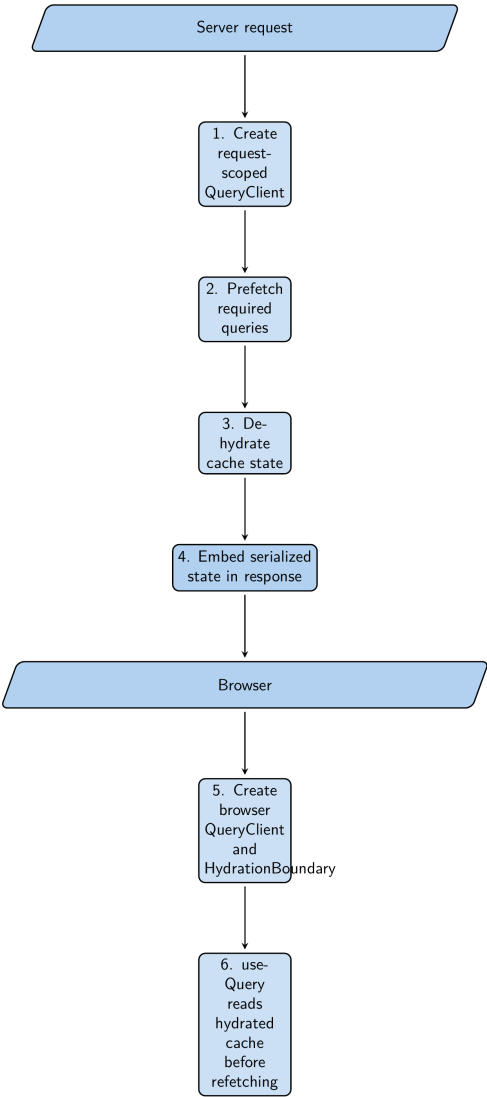
  return <article>{data.title}</article>
}

export function App({
  queryClient,
  dehydratedState,
  postId,
}) {
  return (
    <QueryClientProvider client={queryClient}>
      <HydrationBoundary state={dehydratedState}>
        <PostView postId={postId} />
      </HydrationBoundary>
    </QueryClientProvider>
  )
}
```

That example captures the contract you need to preserve in every SSR integration. The server-side `QueryClient` performs the fetch once. `dehydrate(queryClient)` turns successful query state into a frozen transport object. The browser creates its own long-lived `QueryClient`,

receives the dehydrated state, and exposes it to query observers through `HydrationBoundary`. When `PostView` calls `useQuery()`, `TanStack Query` finds the matching cache entry instead of treating the page as a cold start.

The operational flow is linear, but the boundaries matter: request-scoped cache creation on the server, controlled prefetch, serialization, browser cache restoration, and then normal hook-based consumption.



Request scope is the first correctness boundary Do not create a server-side `QueryClient` at file scope or process scope. In SSR, that turns one request's cache into shared mutable state for later requests. The obvious failure is cross-request data leakage: user A's cached result can become visible to user B if keys collide or route rendering reuses stale state. The less obvious failure is performance degradation from a cache that grows with traffic patterns unrelated to the current response. The correct rule is simple: instantiate a fresh `QueryClient` for each incoming request on the server, use it only for that render, and then allow it to be discarded.

That rule differs from the browser. In the browser, you usually want one `QueryClient` for the lifetime of the application so route transitions and background refetching can reuse state. Server and browser therefore use the same abstraction with different ownership models: per-request on the server, long-lived in the client runtime.

Prefetch only what the initial render needs Prefetching is not a license to fetch your entire route graph. The value of SSR comes from moving latency-sensitive, render-critical reads forward so the response can ship with useful content. If you prefetch every possible query used anywhere beneath the route, you often serialize work that would have been cheaper to defer. That pushes up time to first byte, grows the dehydrated payload, and increases transient memory pressure on the server.

Use `queryClient.prefetchQuery()` when you want cache population without needing the returned data directly. Use `queryClient.fetchQuery()` when server render logic must branch on the result or handle thrown errors explicitly. Use `queryClient.prefetchInfiniteQuery()` or `queryClient.ensureInfiniteQueryData()` only when the initial view truly depends on paginated data being warm before the client starts. The practical filter is straightforward: prefetch route

shell data, above-the-fold records, and data required for SEO-visible content; defer interaction-driven or secondary panels to normal client fetching.

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 60 * 1000,
    },
  },
})

await Promise.all([
  queryClient.prefetchQuery({
    queryKey: ['post', postId],
    queryFn: () => getPost(postId),
  }),
  queryClient.prefetchQuery({
    queryKey: ['author', authorId],
    queryFn: () => getAuthor(authorId),
  }),
])

const dehydratedState = dehydrate(queryClient)
```

That pattern keeps prefetch orchestration near the request or route boundary, where you can see the cost of each added query. Scattering server prefetch inside leaf components usually produces one of two bad outcomes: duplicated orchestration logic, or hidden waterfalls where one component's data dependency delays discovery of another. Route-level coordination is easier to test and easier to reason about under load.

Dehydration turns cache state into a transport artifact `dehydrate(queryClient)` does not serialize your whole application. It serializes TanStack Query cache state into a frozen structure intended for SSR handoff or persistence. By default, dehydration includes currently successful queries. That default is useful because it avoids shipping transient failures or incomplete fetch attempts as if they were stable initial state.

You should treat the dehydrated payload as part of your response budget. Large result sets, deeply nested records, and broad prefetch policies can materially increase HTML size or embedded script payload size. Since dehydration transfers data across an execution boundary, the right question is not “can the cache hold this?” but “should the initial response ship this?” Data that is highly volatile, user-initiated, or irrelevant to initial paint usually belongs on the client side after hydration.

Another subtle point matters when you have more than one hydration event over time. `HydrationBoundary` merges incoming dehydrated queries with an existing client cache using update timestamps. That behavior prevents a later hydration payload from blindly overwriting newer browser state. It also means timestamp semantics affect which version of data survives when navigations, prefetches, or nested boundaries overlap.

Hydration must happen before query hooks observe

The browser half of the pipeline is easy to under-specify. You need a browser `QueryClient`, a `QueryClientProvider`, and a `HydrationBoundary` whose `state` prop contains the server’s dehydrated snapshot. Place the boundary high enough that components using prefetched queries mount underneath it on the first interactive render. If a component using `useQuery()` mounts before the hydrated state is available, TanStack Query sees an empty cache and starts a fresh request. By the time hydration catches up, you have already lost the continuity guarantee.

A typical browser bootstrap keeps the client stable across renders:

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

let browserQueryClient

export function getBrowserQueryClient() {
  if (!browserQueryClient) {
    browserQueryClient = new QueryClient({
```

```
    defaultOptions: {
      queries: {
        staleTime: 60 * 1000,
      },
    },
  })
}

return browserQueryClient
}
```

You can then pass that client into `QueryClientProvider` and wrap the route tree in `HydrationBoundary`. The important invariant is not the exact component structure; it is that query observers should see hydrated cache state before they decide whether to fetch.

Key identity must match exactly Hydration works by cache identity, not by semantic similarity. If the server prefetched `['post', 42]` and the client renders `['post', '42']`, those are different keys. If the server used `['posts', 42]` and the client uses `['post', 42]`, hydration does nothing for that observer. The result is often misdiagnosed as a hydration bug when the real problem is key drift between environments.

Keep key construction centralized. If you already use key factory functions, reuse the same functions in server prefetch code and client query hooks. Chapter 3 covers key design in detail; the SSR-specific rule is that key generation must be deterministic across execution environments. Route params, locale values, tenant identifiers, and auth-dependent scopes all need identical normalization on both sides.

Query functions must be semantically aligned Exact function identity does not need to match, but behavior does. The server and client query functions for a given key should resolve the same resource under the same parameters and shape the result in the same way. If the server query function talks to an internal endpoint while the client uses a public endpoint that omits fields, hydration can succeed yet the next

refetch produces a structurally different object. That creates hard-to-debug rendering drift because the first client frame reflects server data while subsequent client updates reflect a different contract.

You can vary transport details by environment when necessary, but keep the effective resource semantics stable. If authentication headers, base URLs, or cookies differ between server and client, make sure those differences do not change which logical record the query returns for the same key.

Freshness policy determines whether hydration avoids duplicate work Hydration reuses data, but freshness controls whether the client considers that reused data immediately stale. A query with a zero freshness window can hydrate successfully and still refetch on mount right away. Sometimes that is intentional. More often, it erases the latency gain you just paid for on the server. TanStack Query guidance for SSR recommends a non-zero `staleTime`, commonly around `60 * 1000`, so the first client render can use hydrated data without instant revalidation.

That number is not a universal default. It is a policy choice. Reference data may justify a longer window; highly dynamic dashboards may need a shorter one. The important operational point is that SSR without an explicit freshness policy is incomplete design. You need to decide whether the browser should trust server-fetched data for seconds, milliseconds, or not at all.

Framework integration points vary, but the pipeline does not React frameworks differ in where server prefetch occurs and how dehydrated state is embedded. Traditional SSR may prefetch in a request handler before rendering the component tree. More recent server-component models can create request-scoped helpers and prefetch closer to the route segment boundary. Experimental streaming integrations also exist, including `ReactQueryStreamedHydration` from

'@tanstack/react-query-next-experimental', but prefetching remains the stronger baseline when you want to flatten initial waterfalls rather than merely stream around them.

The stable pieces do not change across those environments:

- Create a fresh server `QueryClient` for the request.
- Prefetch the initial queries.
- Produce `dehydratedState` with `dehydrate()`.
- Deliver that state to the browser.
- Create a browser `QueryClient`.
- Wrap the interactive tree in `HydrationBoundary`.

If your framework hides some of these steps behind adapters, still validate that each boundary exists. The abstraction may move, but the cache continuity contract does not.

Common failure modes A few mistakes account for most broken SSR hydration paths.

- **Server singleton cache.** Reusing one `QueryClient` across requests is both a correctness and isolation failure.
- **Over-prefetching.** Fetching every nested dependency before response emission can make SSR slower than client fetching for non-critical content.
- **Mismatched keys.** Server and client can both appear correct locally while hydration silently misses because key shapes differ.
- **Hydration too low in the tree.** If hooks mount outside or above the `HydrationBoundary`, duplicate requests are expected behavior.

- **Immediate staleness.** Hydrated data with an unsuitable `staleTime` often triggers an instant client refetch, which looks like hydration “not working” even though the cache transfer succeeded.
- **Dehydrating the wrong data.** Shipping large, ephemeral, or sensitive query results as part of the initial payload increases cost and can widen exposure if response caching or logging is misconfigured.

You can validate the pipeline empirically. Load a page with the network panel open. A correctly hydrated route should not reissue prefetched queries on the first client render unless your freshness settings explicitly demand it. If you see duplicate requests immediately after hydration, inspect key equality, `staleTime`, and whether the component tree mounts under the boundary early enough.

Using hydration as a cache continuity contract The most useful mental model is not “SSR support” but cache continuity across environments. The server cache is temporary and request-scoped. The browser cache is durable for the application session. Dehydration is the handoff artifact between them. Once you frame it that way, implementation decisions become clearer: prefetch only what must cross that boundary, preserve exact cache identity, and make hydration available before hooks observe. When those invariants hold, server-rendered TanStack Query applications stop behaving like two disconnected fetch systems and start behaving like one cache moving through two runtimes.

8.2. Choosing Freshness Windows and Isolation Boundaries for SSR

An SSR deployment can look correct in development and still fail under real concurrency. The common pattern is deceptively simple: pages render, hydration succeeds, and the UI appears fast. Then traffic grows and three faults emerge at once. A process-level cache leaks data across requests, hydrated queries refetch immediately and erase the latency benefit of server rendering, and oversized request caches push heap usage upward until the server spends more time collecting garbage than rendering. These are not rendering bugs. They are policy bugs around ownership, freshness, and retention.

A workable baseline starts with explicit server and browser boundaries. On the server, create a fresh `QueryClient` for each request. In the browser, keep a long-lived `QueryClient` for the session. Configure a small positive `staleTime` for SSR so hydrated data is not treated as instantly stale unless you intend that behavior.

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

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

let browserQueryClient

export function getBrowserQueryClient() {
  if (!browserQueryClient) {
    browserQueryClient = new QueryClient({
      defaultOptions: {
        queries: {
          staleTime: 60 * 1000,
        },
      },
    })
  }
}
```

```
    },  
  })  
}  
  
  return browserQueryClient  
}
```

That split is not a stylistic preference. TanStack Query’s SSR guidance explicitly warns against creating a `QueryClient` at file scope on the server because the cache then becomes shared across requests and users. The failure is both operational and security-related. You pay for data you did not need to retain, and under the wrong conditions you can serve the wrong user’s state. If you work in a Server Components or App Router environment, a request-scoped helper such as `cache(() => new QueryClient())` preserves the same boundary while fitting the framework’s execution model.

Isolation is a correctness guarantee, not an optimization You should think about SSR cache ownership the same way you think about request-local authentication context. A server-side `QueryClient` belongs to one request, one set of cookies or headers, one tenant boundary, and one rendering pass. Once the response is produced and the dehydrated snapshot has been extracted, that `QueryClient` has served its purpose.

If you promote that cache to process scope, every later request inherits its residue. Even when keys appear tenant-specific, the risks remain. A key factory might miss one dimension of identity, a shared query might be less isolated than you thought, or a mutation-triggered invalidation might not fire in the order your concurrent requests require. The safest architecture is to make cross-request reuse impossible by construction.

This is one of the places where SSR differs sharply from client-side rendering. In the browser, a long-lived cache improves navigation performance because one user is reusing their own data. On the server, a long-lived cache often means one user is reusing someone else’s pro-

cess memory. The same abstraction serves different lifetimes because the trust boundary differs.

Freshness after hydration is a policy decision Hydration transfers data and metadata, not a guarantee of freshness. When the browser restores dehydrated queries, it looks at timestamps and your query options to decide whether the restored entry is fresh, stale but usable, or stale enough to refetch right away. That decision determines whether SSR actually reduces work or merely shifts the first fetch to the server and then repeats it on mount.

A zero `staleTime` is the clearest example. The server prefetches the query, renders the page, and ships the dehydrated cache. The client hydrates successfully, but the query is already stale at mount time, so the observer refetches immediately. If your goal is strongest freshness regardless of duplicate work, that may be acceptable. If your goal is to remove initial client duplication and flatten load on your origin, `staleTime: 0` usually defeats the point.

A small positive `staleTime` changes that behavior substantially. TanStack Query's SSR examples use `60 * 1000` for a reason: it gives the browser a short trust window in which hydrated data is considered fresh, so the user sees continuity rather than an instant second request. Since `fetchQuery` and `prefetchQuery` honor `staleTime` when deciding whether to refetch, the same policy also helps avoid redundant work during server-side orchestration when multiple layers request the same data.

A practical policy matrix You need a resource-specific freshness policy rather than a single global intuition about “fresh enough.” The table below is a useful starting point for SSR routes.

Resource class	Typical SSR staleTime	Decision signal
Reference data	minutes to hours	Country lists, feature metadata, catalog facets, and other low-volatility resources rarely justify immediate refetch after hydration.
User dashboard data	30–120 seconds	Metrics and summaries change, but users usually value fast first paint over second-by-second precision on initial load.
Collaborative records	5–30 seconds	Shared documents or task boards can drift quickly; use short windows and rely on background sync after mount.
Operational feeds	0–5 seconds	Incident streams, trading views, and highly dynamic telemetry may need near-immediate revalidation even when SSR is used for initial markup.

These numbers are not framework defaults. They are starting points for classification. Use domain volatility, user tolerance for drift, and the cost of duplicate fetching to adjust them. A resource updated once per day does not need the same hydration behavior as one updated multiple times per second.

RFC 5861 offers a useful analogy here. A stale-while-revalidate strategy only hides latency when requests arrive inside the window where serving slightly stale data is acceptable and background refresh can happen without blocking the user. If the window is too small, or if requests are too sparse, the next request lands outside that interval and pays the full refresh cost anyway. SSR freshness tuning has the same shape. Tiny windows often preserve theoretical truthfulness while giving up most practical latency savings.

Hydration timing and timestamp interpretation Hydration correctness is not just “did the client receive the data.” It also depends on when that data becomes visible to query observers and how its timestamps are interpreted. `HydrationBoundary` merges dehydrated queries into the client cache using update times. That matters if the browser already has cache state from a prior navigation or from nested route transitions. The newer entry wins.

You should use that behavior deliberately. If the client already holds fresher data than the server snapshot, hydration should not overwrite it. That protects the browser from regressing to an older view during navigation. The inverse is also true: if the browser cache is older, the incoming dehydrated data can upgrade the client state without forcing a visible loading pass.

This makes hydration timing part of your operability model. If a query hook mounts before the `HydrationBoundary` has provided the dehydrated state, the hook evaluates against an empty or outdated cache and may start a request you did not need. Once that extra request begins, the rest of the timing logic becomes much harder to reason about because the client is now competing with hydration rather than continuing from it.

Avoid using `staleTime` as a substitute for analysis A longer `staleTime` can suppress duplicate requests and reduce server pressure, but it is not a universal fix for poor route design. Teams sometimes extend freshness windows to hide waterfalls, excessive prefetching, or expensive endpoints. That trades a short-term latency win for a silent correctness loss. Users see old data longer, and operators lose visibility because the system appears calm while drifting farther from origin truth.

Treat `staleTime` as a declaration of acceptable age for a resource in a specific UX context. If a screen cannot tolerate a 60-second-old value, do not set `staleTime` to 60 seconds just to prevent hydration refetch. Instead, revisit whether the query belongs in SSR at all, whether a lighter route shell should render first, or whether the data should revalidate immediately after mount by design.

Server retries directly affect time to first byte Query retries default to 0 on the server. That is an intentional latency optimization for SSR. A server render that retries failed upstream requests can turn a

transient backend issue into a much longer page response time. On the client, retries are often acceptable because the UI is already interactive and the user can continue reading or acting. On the server, every retry usually delays first byte and ties up a worker.

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

export function makeServerQueryClient() {
  return new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 60 * 1000,
        retry: 0,
      },
    },
  })
}
```

If you override that behavior, do it with a concrete failure budget in mind. A single extra retry across many concurrent SSR requests can magnify partial outages by increasing queue depth and saturating limited upstream capacity. For data that is optional to initial render, a better design is often to omit it from SSR and let the client fetch it after hydration under normal retry semantics.

Memory retention on the server should be deliberately short

A request-scoped server cache is useful while you are rendering. It is rarely useful after the response has been emitted and the dehydrated snapshot has been produced. That means your design goal is usually prompt eligibility for garbage collection, not in-process cache reuse.

Two forces make this important at scale. First, prefetched data tends to be denormalized and response-shaped, which increases per-request memory footprint. Second, concurrent SSR means many request caches can coexist transiently, especially under load or when upstream calls are slow. If you prefetch large collections, then dehydrate them into the response, you briefly hold at least two memory representations: the in-memory query cache and the serialized payload. Over-

prefetching therefore increases both heap pressure and response size.

The operational implication is straightforward: keep server prefetch narrow, and do not retain request `QueryClient` instances beyond the render lifecycle. If you need shared data across requests, use infrastructure designed for shared caching such as a CDN, reverse proxy, or explicit application cache with well-defined invalidation semantics. Do not smuggle that responsibility into the SSR `QueryClient`.

Different runtimes change the pressure profile Node.js servers, edge runtimes, and streaming renderers all support the same basic TanStack Query SSR model, but the cost profile differs. A long-lived Node.js process is vulnerable to heap growth and cross-request contamination if you keep caches around too long. Edge environments often have tighter memory limits and shorter execution windows, which makes broad prefetch especially expensive. Streaming can reduce perceived latency for some content, but it does not remove the need for request isolation or freshness policy; it only changes when parts of the UI become visible.

That means you should validate SSR choices in the runtime you actually deploy. A prefetch set that feels harmless in local development can be too heavy in an edge worker. A stale window that seems conservative in staging may still trigger too many mount-time refetches under real user navigation patterns.

Failure modes worth testing explicitly You can test most SSR policy mistakes before production if you look for the right signals.

- Cross-request contamination: issue two requests with different user or tenant contexts against the same server process and verify that dehydrated payloads contain only request-local data.
- Immediate duplicate fetches: inspect the browser network panel after first paint. If hydrated queries refetch immediately on

mount, inspect `staleTime`, key matching, and hydration placement.

- **Heap growth:** run a representative load test and watch memory over time. If heap usage climbs steadily, check for request `QueryClient` retention, excessive prefetch breadth, or very large dehydrated payloads.
- **Latency under backend partial failure:** introduce upstream slowness and measure time to first byte. If TTFB expands dramatically, verify that server queries are not retrying in ways that make render latency nonlinear.

A disciplined SSR policy The most stable SSR systems use simple boundaries. The server gets a fresh `QueryClient` per request and keeps it only long enough to render and dehydrate. The browser gets a long-lived client that can continue from hydrated state. Each resource gets a freshness window chosen for its business volatility, not for developer convenience. Retries stay conservative on the server because SSR is part of the response path, not a background recovery loop. Once you treat SSR as isolation engineering plus freshness budgeting, the rendering path becomes easier to scale without accidental cache sharing, hidden duplicate work, or unnecessary memory retention.

8.3. Persisting Cache State Across Sessions

A hydrated browser cache gives you continuity across the server-to-client boundary, but a full page reload still destroys that in-memory state. Persistence changes that contract. Instead of treating every reload as a cold start, you can restore selected query data from durable storage and let the application resume from a prior session. That improves startup latency and reduces unnecessary refetching, but it also

creates a harder responsibility: you must decide which restored data is still trustworthy enough to show.

The core integration is compact. You create a stable browser `QueryClient`, configure a persister, and mount `PersistQueryClientProvider` so restoration happens before normal query activity resumes.

```
import React from 'react'
import { QueryClient } from '@tanstack/react-query'
import { PersistQueryClientProvider } from '@tanstack/react-query-
  persist-client'
import { createSyncStoragePersister } from '@tanstack/query-sync-
  storage-persister'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 60 * 1000,
      gcTime: 24 * 60 * 60 * 1000,
    },
  },
})

const persister = createSyncStoragePersister({
  storage: window.localStorage,
  key: 'rq-cache',
  throttleTime: 1000,
})

export function AppProviders({ children }) {
  return (
    <PersistQueryClientProvider
      client={queryClient}
      persistOptions={{
        persister,
        maxAge: 24 * 60 * 60 * 1000,
        buster: 'v1',
      }}
    >
      {children}
    </PersistQueryClientProvider>
  )
}
```

That example is intentionally conservative. It uses `localStorage`

through `createSyncStoragePersister`, aligns `gcTime` with `maxAge`, and assigns a buster string so you can invalidate the whole persisted payload when your data contract changes. The provider is the important part. `PersistQueryClientProvider` avoids the most common startup race by keeping queries idle during restore and by tying subscription cleanup to the React lifecycle.

Persistence changes the meaning of “warm cache”

Without persistence, the browser cache is ephemeral. A tab refresh wipes it, and your freshness decisions operate only within the current session. With persistence, cached state can outlive the tab, the process, and sometimes the user’s memory of having seen the data before. A query result that is structurally valid JSON may still be operationally misleading if it reappears twelve hours later with no indication that the underlying business state has changed.

That is why persistence is not just a performance feature. It is a trust feature. You are deciding whether a restored cache entry is allowed to shape the first rendered view of the application before the network has had a chance to confirm it. If that sounds similar to SSR hydration, the resemblance is real. The difference is the age and uncertainty of the payload. A dehydrated SSR snapshot is usually seconds old. A persisted cache might be hours or days old.

The persisted payload has explicit metadata

TanStack Query persistence serializes a `PersistedClient` object with the shape `{ timestamp, buster, clientState }`. Each field matters.

timestamp tells you when the persisted snapshot was written. This is the baseline for age calculations against `maxAge` and your own domain-specific rules.

buster is the version boundary. Change it when a release makes older

persisted state unsafe or misleading, such as an API shape change, a query-key contract change, or an authentication-scope change.

clientState contains the serialized query and mutation cache state.

The presence of this metadata is what lets persistence behave like a policy surface rather than a blind dump of memory. If you ignore the metadata and think only in terms of “save cache” and “load cache,” you will eventually restore data that is old enough to confuse users but new enough to pass syntactic validation.

Restore timing is the first operational hazard

If you restore persisted state too late, mounted query observers may treat the cache as empty and start fetching. By the time restoration completes, you now have duplicate work or conflicting startup paths. This is the same category of race you already avoid in SSR hydration, but persistence makes it easier to trigger because restore can involve asynchronous storage backends and larger payloads.

`PersistQueryClientProvider` exists to solve that startup problem. It keeps queries idle during restore so components do not race against the storage layer. If you instead wire together `persistQueryClientRestore()` and `persistQueryClientSubscribe()` manually, you need to be deliberate about ordering and cleanup.

```
import { QueryClient } from '@tanstack/react-query'
import {
  persistQueryClientRestore,
  persistQueryClientSubscribe,
} from '@tanstack/react-query-persist-client'

async function bootstrap(queryClient, persister) {
  await persistQueryClientRestore({
    queryClient,
    persister,
    maxAge: 24 * 60 * 60 * 1000,
    buster: 'v1',
  })
}
```



```
const unsubscribe = persistQueryClientSubscribe({
  queryClient,
  persister,
  maxAge: 24 * 60 * 60 * 1000,
  buster: 'v1',
})

return unsubscribe
}
```

Manual control is useful when your startup sequence is unusual, but the burden shifts to you. You need to ensure restoration finishes before your app mounts query-heavy routes, and you must dispose the persistence subscription when appropriate. For most React applications, the provider is the safer default.

Align `gcTime` with persistence `maxAge`

A subtle bug appears when in-memory garbage collection and persisted survival use incompatible windows. TanStack Query documentation explicitly warns that query `gcTime` should be greater than or equal to persistence `maxAge`. Otherwise, restored data can disappear earlier than you expect during hydration or after startup because the in-memory cache considers it eligible for collection before the persisted copy has reached its intended expiry.

The default `maxAge` is 24 hours. If you persist for a full day but leave query garbage collection much shorter, you create a confusing system: the storage layer says “this snapshot is still restorable,” while the runtime says “this query should already be gone.” Users experience that mismatch as random startup behavior where restored data sometimes appears and then vanishes or refetches in a way that feels arbitrary.

A simple alignment rule helps:

- Choose `maxAge` based on how long a persisted snapshot remains trustworthy enough to restore.

- Set `gcTime` to at least that duration.
- Set `staleTime` independently based on how long restored data may be shown without revalidation.

Those three timers solve different problems. `staleTime` governs freshness for active queries. `gcTime` governs in-memory retention. `maxAge` governs how long persisted state may survive outside memory. If you collapse them into one mental model, you either discard useful data too early or keep misleading data too long.

Use `buster` as a release boundary

When your query data shape changes, old persisted entries may deserialize cleanly and still be wrong. A renamed property, a changed nesting structure, or a new auth model can make yesterday's cache dangerous even though it is valid JSON. `buster` is the built-in invalidation mechanism for exactly that case.

```
<PersistQueryClientProvider
  client={queryClient}
  persistOptions={{
    persister,
    maxAge: 24 * 60 * 60 * 1000,
    buster: '2026-10-schema-3',
  }}
>
  {children}
</PersistQueryClientProvider>
```

Treat `buster` as part of deployment hygiene. Change it when you ship a release that makes the old cache semantically incompatible, not just when deserialization would fail. Good triggers include key-shape refactors, backend payload redesigns, or switching users or tenants in a way that should sever continuity with prior cached state.

Not every query should persist

Persistence is most valuable for data that is expensive to refetch, rel-

actively stable, and safe to reuse after a restart. That often includes catalog metadata, read-mostly settings, background lists, or content that benefits from instant warm startup. It is much less appropriate for highly sensitive responses, fast-changing operational data, or views whose correctness depends on the current authenticated identity.

A practical filter is to ask three questions for each query family:

- Would showing this result briefly before revalidation ever mislead the user?
- Would restoring this data after logout, account switch, or tenant switch be unsafe?
- Is the startup latency improvement worth the storage and invalidation complexity?

If the answer to the first two questions is “yes,” do not restore it silently. You may choose not to persist it at all, or you may require a stronger boundary such as buster changes on auth transitions. This becomes especially important for personalized dashboards and any data subject to access control changes.

Quota pressure is normal, not exceptional

Browser storage is finite, and persistence failures under quota pressure are expected. The sync storage persister exposes a `retry` option so you can react when writes fail. The documented mitigation is to remove older or less valuable cached entries and retry the `persist` operation. A common strategy is `removeOldestQuery`.

```
import { createSyncStoragePersister } from '@tanstack/query-sync-storage-persister'

const persister = createSyncStoragePersister({
  storage: window.localStorage,
  key: 'rq-cache',
  throttleTime: 1000,
```

```
    retry: removeOldestQuery,  
  })
```

Under quota pressure, treat persistence as an opportunistic optimization rather than a guarantee. You should prioritize queries that produce clear startup value and shed bulky, low-value entries first. Large paginated datasets, search result pages, and ephemeral detail views are frequent candidates for removal. If you persist everything indiscriminately, the storage layer eventually makes the prioritization decision for you, usually at the worst possible moment.

Async storage backends change startup behavior

Synchronous storage such as `localStorage` is the simplest browser option, but it is not the only one. Async backends are common in environments like React Native, and TanStack Query provides `createAsyncStoragePersister` in `'@tanstack/query-async-storage-persister'` for AsyncStorage-like systems.

```
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: 24 * 60 * 60 * 1000,  
    },  
  },  
})  
  
const asyncStoragePersister = createAsyncStoragePersister({  
  storage: asyncStorage,  
})  
  
export function Providers({ children }) {  
  return (  
    <PersistQueryClientProvider  
      client={queryClient}
```

```
    persistOptions={{
      persister: asyncStoragePersister,
    }}
  >
    {children}
  </PersistQueryClientProvider>
)
}
```

Async restore increases the importance of controlled startup sequencing. The provider still prevents fetch-vs-restore races, but you should expect restoration to have visible timing implications if the persisted payload is large or the device storage layer is slow. That is another reason to keep the persisted set selective.

Restoration state can influence rendering choices

The persistence package also exposes `useIsRestoring`. Use it when you need to suppress UI behavior that would be misleading during restoration. For example, you might avoid showing an “empty state” placeholder while the app is still deciding whether a valid persisted cache exists. That keeps your initial render honest: not empty, not loading from the network, but restoring from durable client state.

You do not need to wrap every component in restoration-aware logic. Apply it where startup rendering would otherwise send the wrong signal. Overusing it makes the UI harder to reason about, and the provider already handles the most important part: query execution order.

Cross-tab synchronization is a separate concern

Persisting a cache does not automatically synchronize live memory across tabs. If you need same-origin tab coordination, TanStack Query offers `broadcastQueryClient`, but it is explicitly experimental and may change even in patch releases. That makes it unsuitable as a casual default for production platforms with long maintenance horizons.

If you adopt it, pin versions tightly and treat its consistency model conservatively. It can improve user experience for multi-tab workflows, but it does not remove the need for sound invalidation, auth-boundary handling, or persistence policy. Cross-tab propagation of stale or mis-scoped data only spreads mistakes faster.

Operational checks that catch persistence mistakes early

You can validate persistence with a few targeted drills.

Cold restart test: load the app, warm key routes, reload the page, and confirm that expected queries restore without unnecessary blank states.

Age-out test: set a short `maxAge` in a test build, wait beyond the window, and verify that stale persisted data is discarded rather than silently revived.

Schema-bump test: change `buster`, reload with old storage present, and confirm that incompatible persisted state is ignored.

Quota test: fill storage near capacity and observe how your persister behaves under write failures. Validate that low-value entries are dropped first if you use a retry strategy such as `removeOldestQuery`.

Auth-boundary test: sign out or switch accounts and make sure persisted data from the old identity does not reappear for the new one.

These checks reveal whether your persistence strategy is improving startup continuity or merely making stale state harder to detect.

Persistence as trust calibration

The strongest persistence designs are selective, versioned, and explicit about age. They restore only the data you are willing to show before confirmation, keep `gcTime` and `maxAge` aligned, and use `buster` to break continuity when the data contract changes. Once you treat

persisted cache as a user-visible claim about reality rather than a cheap performance trick, startup behavior becomes both faster and more honest.

8.4. Operating Queries and Mutations Under Offline and Flaky Network Conditions

Network instability changes the meaning of “loading” and “done.” A query that usually completes in 200 ms can now pause for minutes. A mutation that looked accepted in the UI may still be sitting in memory when the tab closes. If you treat those cases as rare exceptions, users experience broken forms, duplicate submissions, and reconnect storms. TanStack Query gives you control over this behavior, but only if you choose explicit synchronization contracts for reads and writes.

Start with a concrete policy instead of a vague promise of “offline support.” For read paths, choose a `networkMode` that matches the actual data source. For write paths, decide whether a paused mutation may live only in memory or must survive reload and browser restart.

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

function SettingsQuery() {
  return useQuery({
    queryKey: ['settings'],
    queryFn: loadSettingsFromNetwork,
    networkMode: 'online',
  })
}

function CachedAvatarQuery() {
  return useQuery({
    queryKey: ['avatar', userId],
    queryFn: fetchAvatarWithHttpCache,
    networkMode: 'offlineFirst',
  })
}

function LocalDraftsQuery() {
```

```
return useQuery({
  queryKey: ['drafts'],
  queryFn: loadDraftsFromIndexedDB,
  networkMode: 'always',
})
}
```

Those three modes are the foundation. 'online' is the default and blocks query or mutation execution without network. Use it when success genuinely depends on live connectivity. 'always' ignores online status and is appropriate when the queryFn can succeed without network, such as reads from local storage, IndexedDB, or another client-local source. 'offlineFirst' makes one attempt and pauses retries on failure. That is the documented fit when a service worker, HTTP cache, or another cache layer might satisfy the first request locally even while end-to-end connectivity is unreliable.

Offline capability begins with classification

Most applications do not need one uniform offline story. They need several. A product catalog may need stale read availability. A comment form may need deferred submission. A draft editor may need local continuity even when the origin is unreachable. If you use one policy for every query and mutation, you either overcomplicate low-risk flows or under-engineer critical ones.

Classify read paths first:

Read type	Recommended networkMode	Decision signal
Live server truth	'online'	The request is meaningless without fresh network data.
HTTP cached or service-worker cached	'offlineFirst'	A first attempt may succeed locally even when the network path is degraded.
Pure local data	'always'	The query reads from browser or device storage and does not depend on connectivity.

That classification is more useful than trying to detect every possible connectivity state. It is based on what the `queryFn` can actually do, not what the browser thinks the network might be doing.

Browser online status is only a hint

Do not treat `navigator.onLine` or browser online/offline events as authoritative proof that your API is reachable. Those signals are heuristic. A browser can report “online” while the specific origin is down, a VPN is broken, DNS is failing, or a captive portal is intercepting traffic. The reverse also happens: the browser may look disconnected while a service worker or local cache can still satisfy some reads.

This is why `networkMode` is more useful than wiring your whole strategy to connectivity events. The mode expresses what kind of execution is permissible, and `TanStack Query` handles pause and retry behavior accordingly. Connectivity signals are still useful for UI messaging, but they are not a correctness oracle.

Retries behave differently on client and server

Client queries retry three times by default. Server queries retry zero times by default. That distinction matters because reconnect behavior is largely a client-side concern. Once the application is interactive, a small amount of retry can smooth over transient loss. On the server, retries inflate response latency and are usually the wrong place to absorb outages.

For flaky mobile networks, default client retries can still be too aggressive if many queries become stale at once. A reconnect can trigger a fan-out of pending or stale observers. If each one retries independently, you can create your own thundering herd against a struggling origin. Tune retry counts and refetch triggers conservatively for volatile routes, especially when many queries share the same failure domain.

Paused reads are the easy part

Read recovery is mostly about user expectation. If a query pauses because the network is unavailable, you can often continue showing stale data, a cached result, or a clear “waiting to reconnect” state. The UI remains useful because no irreversible state transition has happened. The main design questions are how much staleness you tolerate and whether the screen should distinguish “last known value” from “currently confirmed value.”

That makes read resilience mostly a matter of freshness policy and presentation. Write resilience is harder because users need a durable answer to a more serious question: did the system accept my action?

Mutation pause and resume need an explicit recovery plan

TanStack Query can pause mutations and later resume them, but you need to wire that behavior intentionally. The recovery entry point is `queryClient.resumePausedMutations()`. If you persist offline mutations across reload, one critical caveat applies: they cannot resume af-

ter reload unless a default mutation function exists. Persisted mutation state alone is not enough to recover execution safely.

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

async function updateTodo({ id, title }) {
  const res = await fetch(`/api/todos/${id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ title }),
  })

  if (!res.ok) {
    throw new Error('Failed to update todo')
  }

  return res.json()
}

const queryClient = new QueryClient()

queryClient.setMutationDefaults(['updateTodo'], {
  mutationFn: updateTodo,
})

async function restoreAndResume() {
  await queryClient.resumePausedMutations()
}
```

The practical meaning is easy to miss. If a user submits a form while offline, the mutation can be paused. If the tab remains open, the app may resume it later after reconnect. If the page reloads before that happens, the persisted paused state still needs a callable mutation function to continue. Without the default `mutationFn`, you have metadata describing unfinished work but no execution path to finish it.

Memory durability and business durability are different

A paused mutation in TanStack Query is not the same as a durable delivery guarantee. By default, the paused work lives inside the application's own state model. Persistence can extend that across reload, but it

still does not automatically give you exactly-once delivery, cross-device ordering, or protection from browser crashes at arbitrary moments.

If a write must survive tab closure, sudden process termination, or long offline periods with stronger delivery guarantees, pair your application logic with infrastructure designed for background replay. Workbox Background Sync is a common complement in browser environments. It can store failed requests in IndexedDB and replay them later, even after service-worker restart. That extends durability beyond TanStack Query's in-memory paused state.

Use the two tools for different layers of the contract:

- Use TanStack Query to represent UI state, pause or resume work, and reconcile query caches after writes settle.
- Use a background replay mechanism when the write itself must survive beyond the page lifecycle.

That separation keeps responsibilities clear. TanStack Query manages application state transitions; a service worker queue manages transport durability.

Ordering and idempotency matter more than queuing

Once writes can be delayed, replayed, or resumed after reconnect, ordering becomes a primary design issue. Consider a user who edits the same record three times while offline. If those writes replay out of order, the final server state may reflect an older intent than the UI currently shows. If the user retries manually and the queued request later replays as well, you can also create duplicate submissions.

The library does not solve these domain conflicts for you. You need application-level rules:

- Prefer idempotent operations where possible.

- Send stable identifiers or idempotency keys for create-like writes.
- Make update endpoints tolerant of retries and duplicate delivery.
- Decide whether later writes supersede earlier queued writes or whether every intermediate transition matters.

This is especially important for optimistic UI. Optimism is safest when the mutation is commutative or idempotent, such as toggling a local preference that the server can accept repeatedly without ambiguity. It is riskier for balance transfers, seat reservations, or conflict-prone collaborative edits where replaying old intent can corrupt state.

Choose a reconciliation strategy before reconnect happens

When paused mutations finally run, the cache can become inconsistent with server truth in several ways. The optimistic state may have drifted. Other users may have changed the same records. The server may canonicalize the result differently from what the UI predicted. You should decide in advance how to repair the client.

Three patterns are common:

- **Invalidate and refetch.** Use this as the default when the server is the authoritative source and the read model is broader than a single record.
- **Targeted repair.** Apply direct cache updates only when the returned payload is narrow, stable, and known to match the query model.
- **Server-issued canonical state.** Prefer this when the server performs merges, assigns final identifiers, or resolves conflicts in ways the client cannot predict.

For reconnect-heavy workloads, invalidation is often the safest baseline. It costs extra requests, but it avoids compounding one stale assumption with another. Direct repair pays off only when the cache surface is small and the mutation result fully describes the authoritative post-write state.

Reconnect storms need backpressure

A reconnect event can cause many stale queries and paused mutations to become eligible at once. If you have persisted caches, long-lived tabs, and a broad query surface, this can turn a single network recovery into a burst of traffic. The risk is not only origin load. The user may see many components flipping through loading states while writes and reads compete for bandwidth.

You can reduce this pressure by narrowing which queries are eligible to refetch immediately, choosing realistic `staleTime` values, and avoiding eager invalidation of broad key prefixes after every resumed mutation. Not all stale data needs to be corrected in the first second after reconnect. Prioritize visible screens and correctness-critical resources. Defer secondary panels or background lists until the interaction path is stable again.

Testing offline behavior requires the right failure model

A common testing mistake is to flip the browser's offline toggle and assume you have exercised the full recovery path. That is useful for some scenarios, but it does not simulate everything. Browser-level offline testing can show whether queries pause, whether the UI communicates pending work, and whether reconnect resumes expected fetches. It does not necessarily validate service-worker replay behavior.

If you use Workbox Background Sync, test the queueing and replay flow in an environment where the service worker actually handles failed requests and restarts. Chrome DevTools offline mode is not

a faithful simulation for every background sync path. Validate page-level pause behavior separately from service-worker replay behavior, because they fail in different ways.

A practical test checklist helps:

- Disconnect before a read query starts; verify pause or cached fallback behavior matches `networkMode`.
- Disconnect during a mutation; verify the UI shows pending or paused intent rather than false success.
- Reload while mutations are paused; verify persisted mutations can resume only when a default `mutationFn` is available.
- Reconnect with many stale queries present; observe whether the application refetches in a controlled way.
- If you use a background replay queue, test actual queued request replay rather than assuming page-level recovery covers it.

A disciplined offline contract

The most reliable applications do not claim generic offline support. They define which reads may come from local or cached sources, which writes may pause, which writes must survive reload, and how the system reconciles truth after reconnect. In TanStack Query, that means selecting `networkMode` per workload, preserving paused mutation continuity with `queryClient.resumePausedMutations()`, and treating durable delivery, ordering, and idempotency as explicit architecture decisions rather than side effects of a retry loop.

8.5. Applying Production Heuristics to Real TanStack Query Architectures

Most production failures in TanStack Query applications do not come from missing APIs. They come from reasonable local decisions that collide: one team adds route prefetch, another adds persistence, a third adds optimistic updates, and a fourth suppresses refetching because duplicate requests look expensive. The result is usually not a dramatic crash. It is a system that feels unpredictable: stale data appears after reload, hydration mismatches happen only on some routes, offline writes replay in the wrong context, and nobody can explain which cache is authoritative at a given moment.

A stable architecture starts by making cache ownership explicit at each boundary. Use one `QueryClient` lifetime per environment, choose freshness windows per resource class, and only add persistence or offline replay where the business contract actually requires them.

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

export function makeServerQueryClient() {
  return new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 60 * 1000,
        retry: 0,
      },
    },
  })
}

let browserQueryClient

export function getBrowserQueryClient() {
  if (!browserQueryClient) {
    browserQueryClient = new QueryClient({
      defaultOptions: {
        queries: {
          staleTime: 60 * 1000,
          gcTime: 24 * 60 * 60 * 1000,
        },
      },
    })
  }
}
```



```
    },  
  })  
}  
  
  return browserQueryClient  
}
```

That example is not a universal configuration. It is a statement of architectural intent. The server client is request-scoped and conservative about retries because it sits on the response path. The browser client is app-lifetime because it owns navigation continuity. Persistence, if you enable it, becomes a third boundary layered on top of the browser client rather than a replacement for it.

Use boundary-specific lifetimes, not one cache everywhere

The first heuristic is simple and non-negotiable: match `QueryClient` lifetime to trust boundary.

Per-request on the server. Do not create a root-level server `QueryClient`. That risks cross-request data sharing, unnecessary memory retention, and accidental leakage across tenants or users.

App-lifetime in the browser. Reuse one browser `QueryClient` so route transitions and background refetches can build on prior state.

Optional persisted storage across sessions. Persist selected browser cache state only when warm startup is worth the trust and invalidation complexity.

If you collapse these three boundaries into one mental model, every later policy becomes harder. You start asking the wrong questions, such as whether server-prefetched data should survive logout, or whether a persisted payload can safely stand in for route hydration. It cannot. Each layer has a different age profile, isolation requirement, and failure mode.

Treat query keys as a public contract

Query keys are more than hook arguments. They are the naming system that ties together prefetch, hydration, invalidation, persistence, and reconciliation after writes. If those names drift, your architecture becomes brittle no matter how carefully you tune `staleTime` or retries.

Keep key construction centralized and deterministic. Use the same key factories for server prefetch and browser hooks. Include every identity dimension that affects meaning: tenant, locale, filters, pagination cursor, auth scope, or feature variant. If a resource identity is ambiguous, fix the key design before adding more cache logic around it.

This is one of the few places where extra discipline simplifies everything else. Strong keys make invalidation narrower, hydration more reliable, and persistence safer to reason about. Weak keys force you to compensate with broad invalidations, overly short freshness windows, or unnecessary refetching.

Choose freshness windows from domain volatility, not convenience

A non-zero SSR `staleTime` is usually the right default because it prevents immediate client refetch after hydration. That does not mean every query should get the same window. The correct value depends on the resource's volatility, user expectations, and any upstream HTTP caching that already constrains freshness.

A useful heuristic is to classify resources before assigning defaults:

- **Reference and metadata queries:** use longer windows. Country lists, feature flags, and static catalog facets rarely need second-by-second truth.
- **User dashboards and summaries:** use moderate windows. Fast initial render often matters more than instant post-hydration revalidation.

- **Collaborative or operational views:** use short windows or intentional refetch on mount. These screens drift faster.

HTTP cache vocabulary helps here. Concepts like stale-while-revalidate and stale-if-error provide a useful analogy for choosing TanStack Query freshness and failure fallback windows. If you would be comfortable serving a cached response briefly while revalidating in the background at the HTTP layer, a similar application-layer `staleTime` may be reasonable. If a value would be dangerous to display even briefly after disconnect or reload, keep the freshness window short and require faster revalidation.

Do not use `staleTime: 0` everywhere as a reflexive claim of correctness. That often just recreates duplicate work, defeats SSR latency gains, and hides the fact that your freshness policy has not been analyzed.

Prefer invalidation over pervasive manual cache repair

When a mutation changes server state, you have two broad strategies: invalidate related queries and let them refetch, or repair cached data directly. Both are valid. The trap is treating direct cache writes as the default because they feel efficient.

Prefer targeted invalidation unless all of the following are true:

- The read model is narrow and stable.
- The mutation result fully describes the authoritative new state.
- You can update every affected cache entry deterministically.

If even one of those conditions is shaky, broad invalidation is often cheaper in total engineering cost than maintaining bespoke repair logic across many screens. As the application grows, direct writes accumulate hidden coupling. A seemingly harmless response shape change

then breaks several caches at once, and the bugs only appear under specific navigation sequences.

Use direct cache repair where it clearly pays for itself: small detail views, append-only local interactions, or cases where the server returns the exact object needed by a small number of observers. Reserve it for cases you can test thoroughly.

Keep SSR orchestration explicit

Server rendering becomes hard to reason about when prefetch logic is scattered through leaf components, route middleware, and framework adapters at the same time. Keep SSR and hydration boundaries visible in your architecture. Route-level or request-level orchestration should answer three questions explicitly:

- Which queries must be prefetched before the first response?
- Which queries may wait for the browser?
- Which freshness policy prevents immediate duplicate work after hydration without overstating truthfulness?

That clarity matters more as routes gain nested layouts, streaming segments, or client-only branches. Experimental streaming hydration tools and core hydration APIs are not equivalent in stability or operational maturity. Budget risk accordingly. If you adopt experimental tooling such as `ReactQueryStreamedHydration` or cross-tab broadcast features, pin versions tightly and test upgrade paths intentionally.

Use persistence only where restored data is still honest

Persistence can make relaunch feel instant, but it can also revive data that is structurally valid and operationally wrong. The heuristic here is to align persistence policy with truthfulness policy.

Set `maxAge` according to how long restored data remains trustworthy

enough to show. Set `gcTime` to at least that duration so restored cache is not prematurely discarded in memory. Change `buster` when the persisted data schema changes or when an auth boundary changes and prior cached state should no longer survive.

A compact rule set works well:

- Persist low-volatility, high-reuse queries.
- Avoid silently restoring highly sensitive or fast-changing data.
- Invalidate persisted cache on build or schema changes via `buster`.
- Treat logout, tenant switch, and major permission changes as persistence boundaries.

If you cannot explain why a persisted query is safe to show before fresh confirmation, do not persist it by default.

Make offline behavior resource-specific

Offline resilience is not one toggle. Some queries benefit from `networkMode: 'offlineFirst'` because a service worker or HTTP cache may satisfy the first request locally. Others should remain `'online'` because stale or partial data is misleading. Pure local reads can use `'always'` because connectivity is irrelevant.

Apply the same specificity to mutations. Not every write should be replayed after a long disconnect. A local preference update might be safe to queue. A payment attempt, booking action, or conflict-prone collaborative edit needs stronger ordering, idempotency, and server reconciliation design. Use paused mutation recovery where the business workflow tolerates delayed submission, and require stricter server-side safeguards where duplicate or reordered delivery would be harmful.

One practical review question helps: if this mutation replays 30 minutes later after the user has switched context, would the result still be correct? If not, you need a tighter boundary than generic offline replay.

Hydration mismatches are data-boundary bugs

Hydration mismatches are often discussed as a React rendering issue, but in real systems they are frequently cache-boundary mistakes. A page can render different first-frame output on the server and client because restored persisted state changes a branch, because a client-only read runs too early, or because hydration occurs twice with overlapping data of different ages.

Treat mismatches as a production concern. Monitor them. Reproduce them under real reload and restore conditions. Bound client-only logic carefully, especially when local storage or restored cache influences the first render. If unavoidable divergence exists, isolate it explicitly rather than letting it bleed into the route shell.

This is another reason to keep hydration topology simple. One server transfer boundary plus one browser restoration boundary is easier to reason about than multiple overlapping caches affecting the same first paint.

Standardize defaults, but not at the expense of classification

Organization-wide defaults are valuable because they reduce incidental complexity. Define sensible defaults for `staleTime`, retry behavior, persistence strategy, and reconnect behavior. Then let exceptional workloads opt out.

A healthy default set usually includes:

- Non-zero SSR `staleTime`.
- Server retries kept at 0 unless you have a measured reason to change them.

- Persistence disabled by default for sensitive or highly volatile queries.
- Query-key factories shared across prefetch, hooks, and invalidation.

The mistake is forcing every workload to accept the same cache policy because centralization feels simpler. Standardize the decision process, not just the numeric defaults.

Maintain a production runbook

A mature TanStack Query architecture benefits from a short operational checklist. Keep it close to deployment and incident response, not buried in design notes.

- Verify no server `QueryClient` is created at module scope.
- Review routes with `staleTime: 0` and justify each one.
- Ensure persisted `maxAge` does not exceed what users should trust, and that `gcTime` is at least as large.
- Bump `buster` on persisted schema or auth-boundary changes.
- Use `PersistQueryClientProvider` to avoid restore races.
- Test reload-after-offline-mutation flows and confirm default mutation functions exist when paused mutations must resume.
- Pin experimental package versions and isolate them behind clear adoption criteria.
- Monitor hydration mismatch warnings as indicators of data-boundary drift, not mere cosmetic noise.

That runbook is deliberately plain. Most failures come from forgetting one of these basics under release pressure, not from lacking an advanced cache algorithm.

Prefer simple invariants over clever cache choreography

The most maintainable TanStack Query systems are not the ones that exploit every feature. They are the ones that keep a few invariants stable: request-scoped server caches, app-scoped browser caches, selective persistence, resource-specific freshness, and conservative use of direct cache writes and offline replay. Once those rules are in place, TanStack Query stays what it is best at being: a disciplined server-state runtime with explicit contracts, rather than a bag of ad hoc cache tricks.

