THE INTERVIEW HANDBOOK

TanStack Query

React server-state management - from first query to production architecture

v5 / React | JavaScript-first examples | Updated 13 November 2026

A practical interview guide for frontend and full-stack developers

This handbook focuses on the questions interviewers ask about fetching, caching, mutations, synchronization, and production behavior. Examples use React and JavaScript; TypeScript notes are included where useful.

> What this library is for
> TanStack Query manages server state: data that lives outside the browser, is fetched asynchronously, can become stale, and must stay synchronized with a server. It is not a general replacement for Zustand, Redux, or local component state.

Interview coverage

- Core concepts, QueryClient, providers, query keys and query lifecycle
- Freshness, cache lifecycle, retries, refetching and cancellation
- Mutations, invalidation, direct cache updates and optimistic UI
- Pagination, infinite scrolling, dependent and parallel queries
- Prefetching, Suspense, SSR hydration, testing, Devtools and performance
- Version 5 API changes, common mistakes, interview Q&A; and practical exercises

How to use this handbook

For a quick interview revision, read the highlighted concepts, learn the example patterns, then answer the Q&A; without looking. For a practical round, work through the job-board scenario in the final section.

Contents

01	TanStack Query mental model
02	Setup and QueryClientProvider
03	Queries and query states
04	Query keys and key factories
05	Caching: staleTime, gcTime and defaults
06	Query functions, errors and retries
07	Query patterns: dependent and parallel
08	Query options, select and initial data
09	Mutations and cache synchronization
10	Optimistic updates and rollback
11	Pagination and infinite queries
12	Cancellation, polling and network modes
13	QueryClient methods and prefetching
14	Suspense, SSR and hydration
15	Devtools, testing and debugging
16	Performance and production architecture
17	v4 to v5 and current v5 API notes
18	Interview questions and practical scenarios
19	Last-minute revision sheet and glossary

Version scope

The official React docs currently show the v5 line. The npm package release information visible on 13 November 2026 lists @tanstack/react-query 5.107.1. This handbook uses current v5 object syntax and calls out recently documented QueryClient API changes. Check the migration guide when maintaining an older v5 or v4 codebase.

01 / FOUNDATIONS

TanStack Query mental model

Start here: why it exists, what it owns, and what it does not.

What is TanStack Query?

TanStack Query (formerly React Query) is an asynchronous state-management and data-fetching library. In a React app, it helps fetch data from REST, GraphQL, or any Promise-based source, then cache, share, refetch, update, and synchronize that data.

The problem it solves

Without a server-state library, components often repeat the same work: loading flags, error handling, request cancellation, cache reuse, deduplicating identical requests, refetching after changes, and keeping data fresh when the user returns to a tab.

MANUAL FETCHING - REPEATED BOILE JSX

```jsx
useEffect(() => {
  let cancelled = false;

  setLoading(true);
  fetch('/api/jobs')
    .then((response) => response.json())
    .then((jobs) => {
      if (!cancelled) setJobs(jobs);
    })
    .catch(setError)
    .finally(() => setLoading(false));

  return () => { cancelled = true; };
}, []);
```

The code above is not inherently wrong. The difficulty is maintaining all the edge cases consistently across many components. TanStack Query centralizes these concerns around a cache and declarative query definitions.

Server state vs client state

Server state	Client / UI state
Fetched from an API or database	Created and owned by the frontend
Can change outside the current browser tab	Usually changes through local user interaction
Needs caching, freshness, retries and refetching	Usually needs simple updates and subscriptions
Examples: users, jobs, orders, products	Examples: modal open, selected tab, draft UI toggle

Do not confuse the tools
Use TanStack Query for remote data. Use Zustand, Redux, or React state for client-owned UI state. Avoid copying every query response into Zustand: that creates two sources of truth that must be synchronized.

TanStack Query vs common alternatives

Tool	Main responsibility	How it differs
TanStack Query	Server-state cache and synchronization	Fetching, caching, invalidation, retries, mutations
Axios / fetch	HTTP transport	Sends requests; does not provide a query cache by itself
Zustand	Client/global state	Stores UI and client-owned state; no automatic server cache lifecycle
Redux Toolkit	App state + optional RTK Query	Redux store model; RTK Query is its data-fetching/caching solution
useEffect	React side-effect mechanism	Can fetch manually, but cache policy and coordination are your responsibility

Interview-ready definition

> Say it like this
> TanStack Query is a server-state management library. It manages asynchronous data fetching, caching, background refetching, mutations, and cache synchronization, so components do not have to hand-roll those behaviors with effects and local loading state.

02 / SETUP

Install and configure

A stable QueryClient and one provider near the application root.

Install

INSTALL BASH

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

Create one QueryClient

QueryClient coordinates the query and mutation caches and exposes imperative cache operations. Create it once for the browser application; do not create a brand-new client on every render.

QUERY CLIENT CONFIGURATION JAVASCRIPT

```javascript
// src/lib/queryClient.js
import { QueryClient } from '@tanstack/react-query';

export const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 60 * 1000,
      retry: 1,
      refetchOnWindowFocus: false,
    },
  },
});
```

Wrap the app with QueryClientProvider

APP ENTRY JSX

```jsx
// src/main.jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import { QueryClientProvider } from '@tanstack/react-query';
import { ReactQueryDevtools } from '@tanstack/react-query-devtools';
import { queryClient } from './lib/queryClient';
import App from './App';

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

Why one client?
If you instantiate QueryClient inside a component body, each render can create a fresh cache and lose the intended shared cache. Keep the browser client stable, usually in a module or a lazy state initializer. On a server, create a request-scoped client instead of sharing one globally across users.

Typical project structure

ONE POSSIBLE STRUCTURE BASH

```bash
src/
  api/
    jobsApi.js          // HTTP functions
  queries/
    jobQueries.js       // query keys + query options
    jobMutations.js     // optional shared mutation helpers
  lib/
    queryClient.js
  components/
    JobList.jsx
  pages/
    JobsPage.jsx
```

A reusable HTTP function

TanStack Query does not require Axios. It accepts any query function that returns a Promise. With fetch, check response.ok because fetch normally resolves even for HTTP 4xx/5xx responses.

HTTP FUNCTION - FETCH JAVASCRIPT

```jsx
// src/api/jobsApi.js
export async function fetchJobs(filters = {}) {
  const params = new URLSearchParams(filters);
  const response = await fetch(`/api/jobs?${params}`);

  if (!response.ok) {
    throw new Error(`Request failed: ${response.status}`);
  }

  return response.json();
}
```

03 / QUERIES

useQuery and query states

A query is a cached, declarative dependency on asynchronous data.

The basic useQuery shape

BASIC QUERY - V5 OBJECT SYNTAX JSX

```jsx
import { useQuery } from '@tanstack/react-query';
import { fetchJobs } from '../api/jobsApi';

function JobsPage() {
  const jobsQuery = useQuery({
    queryKey: ['jobs'],
    queryFn: () => fetchJobs(),
  });

  if (jobsQuery.isPending) return <p>Loading jobs...</p>;
  if (jobsQuery.isError) return <p>{jobsQuery.error.message}</p>;

  return jobsQuery.data.map((job) => (
    <p key={job.id}>{job.title}</p>
  ));
}
```

At minimum, useQuery needs a unique queryKey and a queryFn that returns a Promise which resolves with data or throws an error. A mutation is used for server-side writes; a query describes data to read.

Three query statuses

Property	Meaning	When to use it
status: pending / isPending	No data has been resolved for this query yet	Initial state / not yet fetched
status: error / isError	The latest query attempt ended in an error state	Show error or retry UI
status: success / isSuccess	Data is available successfully	Render data
isFetching	A request is running now, including background refetch	Small refresh indicator
isLoading	First fetch is actively running; equivalent to isPending && isFetching	Initial loading spinner
isRefetching	A fetch is running after the initial fetch	Background refresh indicator

Important interview distinction
status describes the query result state; fetchStatus describes the transport activity. A disabled query can be pending while fetchStatus is idle. Therefore isPending and isLoading are not interchangeable.

status vs fetchStatus examples

Situation	status	fetchStatus	isLoading
First load in progress	pending	fetching	true
Disabled query waiting for an ID	pending	idle	false
Successful cached data, no request	success	idle	false
Successful data refreshing in background	success	fetching	false
Request failed and no data exists	error	idle	false

Render initial loading and background refresh separately

QUERY UI STATES JSX

```jsx
function JobList() {
  const { data, isPending, isError, error, isFetching } = useQuery({
    queryKey: ['jobs'],
    queryFn: fetchJobs,
  });

  if (isPending) return <p>Loading...</p>;
  if (isError) return <p>{error.message}</p>;

  return (
    <>
      {isFetching && <small>Refreshing...</small>}
      {data.map((job) => <p key={job.id}>{job.title}</p>)}
    </>
  );
}
```

What does useQuery return?

Common fields include data, error, status, fetchStatus, isPending, isError, isSuccess, isFetching, isLoading, isRefetching, isStale, refetch, and dataUpdatedAt. The exact result also includes flags for paused, placeholder, and refetch-error cases.

04 / CACHE IDENTITY

Query keys and key factories

The key identifies the data. It is not just a label.

Why query keys matter

TanStack Query uses query keys to cache, share, refetch, and target data. Query keys are arrays at the top level and must be serializable and unique for the data they describe.

QUERY KEY EXAMPLES JAVASCRIPT

```javascript
// A list
['jobs']

// A single record
['job', 42]

// A filtered list
['jobs', { status: 'open', page: 1 }]

// Include every changing variable used by queryFn
['jobs', { search, status, page }]
```

A key is a dependency for queryFn

KEY MUST REPRESENT REQUEST INPUT JAVASCRIPT

```javascript
function useJobs({ search, status, page }) {
  return useQuery({
    queryKey: ['jobs', { search, status, page }],
    queryFn: () => fetchJobs({ search, status, page }),
  });
}
```

When a variable changes, a different key represents a different logical result. The query can then use or fetch the cache entry associated with that key. Omitting a variable can cause unrelated results to share one cache entry.

Deterministic hashing

Object property order does not change a query key hash, but array item order does. Keys should remain serializable; avoid functions, class instances, DOM nodes, or mutable objects as key parts.

KEY COMPARISON JAVASCRIPT

```javascript
// Equivalent object contents (property order differs)
['jobs', { status: 'open', page: 1 }]
['jobs', { page: 1, status: 'open' }]

// Different keys (array order differs)
['jobs', 'open', 1]
['jobs', 1, 'open']
```

Key factory pattern

QUERY KEY FACTORY JAVASCRIPT

```javascript
// src/queries/jobQueries.js
export const jobKeys = {
  all: ['jobs'],
  lists: () => [...jobKeys.all, 'list'],
  list: (filters) => [...jobKeys.lists(), filters],
  details: () => [...jobKeys.all, 'detail'],
  detail: (id) => [...jobKeys.details(), id],
};

// Usage
useQuery({
  queryKey: jobKeys.list({ status: 'open', page: 1 }),
  queryFn: () => fetchJobs({ status: 'open', page: 1 }),
});
```

Partial matching and exact matching

FILTER BY QUERY KEY JAVASCRIPT

```javascript
// Invalidates all queries beginning with ['jobs']
queryClient.invalidateQueries({ queryKey: ['jobs'] });

// Only the exact ['jobs'] key
queryClient.invalidateQueries({
  queryKey: ['jobs'],
  exact: true,
});
```

Interview answer

Query keys are serializable arrays that uniquely describe server data. I include every changing input used by the query function, organize keys hierarchically, and use prefix matching to invalidate related cache entries.

05 / CACHING

staleTime, gcTime and defaults

Freshness and cache retention solve different problems.

staleTime vs gcTime

Option	Controls	Default in current React docs	Interview explanation
staleTime	How long data is considered fresh	0 ms	Fresh data is not refetched just because an observer mounts/focuses/reconnects during this period.
gcTime	How long inactive cache data remains before garbage collection	5 minutes	Starts mattering after no active observer uses the query.

These settings are not interchangeable. staleTime controls freshness; gcTime controls how long an unused query stays in memory. Active queries are not garbage-collected just because gcTime has elapsed.

The lifecycle in plain language

Step	What happens
1. First request	No cached data; query function runs and result is stored.
2. Fresh window	For staleTime, cached result is considered fresh. A new observer can reuse it without staleness-triggered refetch.
3. Data becomes stale	It can refetch in the background on mount, window focus, or reconnect if those options are enabled.
4. Query becomes inactive	No active observer remains. Data stays in cache for gcTime.
5. Garbage collection	If it remains inactive until gcTime expires, it can be removed from memory.

The three staleTime choices

STALETIME OPTIONS JAVASCRIPT

```javascript
useQuery({
  queryKey: ['reference-data'],
  queryFn: fetchReferenceData,
  staleTime: 60_000,       // fresh for 1 minute
});

useQuery({
  queryKey: ['rarely-changing'],
  queryFn: fetchRarelyChanging,
  staleTime: Infinity,    // only invalidation can make it stale
});

useQuery({
  queryKey: ['boot-config'],
  queryFn: fetchBootConfig,
  staleTime: 'static',    // even invalidation does not refetch it
});
```

Infinity vs static
Both avoid time-based staleness. With Infinity, explicit invalidation still works. With static, invalidation does not make the query stale for refetch. Use static only for truly immutable data during the application lifetime.

Important defaults to remember

Default behavior	What it means
Cached data starts stale	staleTime defaults to 0
Background refetch triggers	Stale queries can refetch on mount, window focus and reconnect
Inactive cache retention	gcTime defaults to 5 minutes in the browser
Query retry	Failed queries retry 3 times by default in the browser; server-side default is 0
Structural sharing	JSON-compatible results are structurally shared to preserve unchanged references

Set defaults globally, override locally

GLOBAL AND LOCAL DEFAULTS JAVASCRIPT

```javascript
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 30_000,
      gcTime: 10 * 60 * 1000,
      retry: 1,
      refetchOnWindowFocus: true,
    },
  },
});

// This query overrides the global staleTime
useQuery({
  queryKey: ['live-status'],
  queryFn: fetchLiveStatus,
  staleTime: 5_000,
});
```

06 / QUERY FUNCTIONS

Errors, retries and cancellation

Query functions must resolve data or throw a meaningful error.

The query function contract

A queryFn must return a Promise. Successful data should be returned. Failed HTTP responses must be turned into thrown errors; otherwise TanStack Query may treat an HTTP 404 response as a successful fetch if the function just returns response.json().

QUERY FUNCTION WITH ERROR CHECK JAVASCRIPT

```javascript
async function getJob({ queryKey, signal }) {
  const [, jobId] = queryKey;
  const response = await fetch(`/api/jobs/${jobId}`, { signal });

  if (!response.ok) {
    throw new Error(`Could not load job (${response.status})`);
  }

  return response.json();
}

useQuery({
  queryKey: ['job', jobId],
  queryFn: getJob,
});
```

Retries

A failed query is retried by default up to 3 times in the browser. You can disable retries, set a number, or implement a function. Mutations do not retry by default.

RETRY STRATEGY JAVASCRIPT

```javascript
useQuery({
  queryKey: ['jobs'],
  queryFn: fetchJobs,
  retry: 2,
  retryDelay: (attempt) => Math.min(1000 * 2 ** attempt, 30_000),
});

// Do not retry a not-found response forever
useQuery({
  queryKey: ['job', id],
  queryFn: fetchJob,
  retry: (failureCount, error) => {
    if (error.status === 404) return false;
    return failureCount < 2;
  },
});
```

> Do not blindly retry everything
> Retries help transient network/server failures. They do not fix validation errors, missing resources, or authorization failures. Decide based on error category and avoid retry storms against an unhealthy service.

Cancellation with AbortSignal

TanStack Query gives the query function an AbortSignal. Pass it to fetch or another transport that supports AbortSignal. When the query becomes obsolete and the signal is consumed, the underlying request can be aborted.

CANCELLABLE FETCH JAVASCRIPT

```javascript
useQuery({
  queryKey: ['search', searchTerm],
  queryFn: async ({ signal }) => {
    const response = await fetch(
      `/api/search?q=${encodeURIComponent(searchTerm)}`,
      { signal }
    );

    if (!response.ok) throw new Error('Search failed');
    return response.json();
  },
});
```

Manual cancellation is also possible through queryClient.cancelQueries({ queryKey }). Cancellation is useful for avoiding outdated results and for protecting optimistic cache updates from an in-flight refetch.

07 / QUERY PATTERNS

Dependent and parallel queries

Choose serial execution only when one request truly depends on another.

Parallel queries

If independent data can be fetched at the same time, place multiple useQuery calls in the same component. TanStack Query runs them concurrently.

STATIC PARALLEL QUERIES JSX

```jsx
function Dashboard() {
  const usersQuery = useQuery({
    queryKey: ['users'],
    queryFn: fetchUsers,
  });

  const teamsQuery = useQuery({
    queryKey: ['teams'],
    queryFn: fetchTeams,
  });

  const statsQuery = useQuery({
    queryKey: ['stats'],
    queryFn: fetchStats,
  });

  // Render each state as needed
}
```

Dynamic parallel queries with useQueries

DYNAMIC PARALLEL QUERIES JAVASCRIPT

```javascript
const userIds = [12, 18, 24];

const messageQueries = useQueries({
  queries: userIds.map((userId) => ({
    queryKey: ['messages', userId],
    queryFn: () => fetchMessages(userId),
  })),
});
```

Dependent queries with enabled

A dependent query waits for data from another query. Use enabled to make the second query start only when its required input is available.

DEPENDENT QUERY JAVASCRIPT

```javascript
const userQuery = useQuery({
  queryKey: ['user', email],
  queryFn: () => fetchUserByEmail(email),
});

const userId = userQuery.data?.id;

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

Avoid waterfalls when possible
A dependent query is serial by definition. If both results can be fetched from a single backend endpoint, redesigning that endpoint may remove a request waterfall and improve latency.

Disabled queries and manual refetch

enabled: false disables automatic execution. The query can still be manually refetched using the returned refetch function. For TypeScript, skipToken is a type-safe alternative for conditional query functions, but refetch does not work with skipToken because there is no active queryFn.

LAZY / DISABLED QUERY JSX

```jsx
const [appliedSearch, setAppliedSearch] = useState('');

const searchQuery = useQuery({
  queryKey: ['search', appliedSearch],
  queryFn: () => searchJobs(appliedSearch),
  enabled: appliedSearch.length > 0,
});
```

08 / QUERY OPTIONS

select, initialData and placeholderData

Know where transformed data lives and what gets written to cache.

select transforms data for one observer

select lets a component read a derived value without changing the underlying cached response. It is useful for picking fields or calculating a small view model.

SELECT OPTION JAVASCRIPT

```javascript
const jobCountQuery = useQuery({
  queryKey: ['jobs'],
  queryFn: fetchJobs,
  select: (jobs) => jobs.length,
});

// jobCountQuery.data is a number, not the jobs array
```

initialData vs placeholderData

Option	Meaning	Cache behavior	Typical use
initialData	Data that can be treated as the query's initial result	Written into cache	You have real, authoritative initial data
placeholderData	Temporary data to render while the query's own data is not ready	Not persisted as query data	Keep previous page visible during pagination

Keep previous page data in v5

In v5, the keepPreviousData option was removed. Use the exported keepPreviousData helper as placeholderData.

V5 PLACEHOLDER DATA JAVASCRIPT

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

const jobsQuery = useQuery({
  queryKey: ['jobs', { page, status }],
  queryFn: () => fetchJobs({ page, status }),
  placeholderData: keepPreviousData,
});

// Use isPlaceholderData to know whether the visible page
// is still the previous page while the new page is loading.
```

queryOptions helper

queryOptions lets you define queryKey, queryFn, and shared options once, then reuse them with useQuery and imperative QueryClient operations. It is especially helpful in TypeScript because it preserves inference across call sites.

REUSABLE QUERY OPTIONS JAVASCRIPT

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

function jobOptions(jobId) {
  return queryOptions({
    queryKey: ['job', jobId],
    queryFn: () => fetchJob(jobId),
    staleTime: 30_000,
  });
}

function JobDetails({ jobId }) {
  const jobQuery = useQuery(jobOptions(jobId));
  // ...
}
```

Interview distinction
select changes what a component observes; it does not rewrite the shared cache. initialData is real initial cache data. placeholderData is temporary display data, commonly used to avoid a loading flash.

09 / WRITES

Mutations and cache synchronization

Queries read server state; mutations change server state.

What is a mutation?

useMutation is designed for create, update, delete, and other server-side effects. Unlike useQuery, a mutation does not automatically execute on mount; you trigger it with mutate or mutateAsync.

BASIC MUTATION JSX

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

function AddJobButton() {
  const queryClient = useQueryClient();

  const addJobMutation = useMutation({
    mutationFn: (newJob) => createJob(newJob),
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['jobs'] });
    },
  });

  return (
    <button
      disabled={addJobMutation.isPending}
      onClick={() => addJobMutation.mutate({ title: 'Frontend Engineer' })}
    >
      {addJobMutation.isPending ? 'Saving...' : 'Add job'}
    </button>
  );
}
```

mutate vs mutateAsync

Method	Return behavior	Use when
mutate(variables)	Starts mutation; does not return its Promise	You mostly use hook state and callbacks
mutateAsync(variables)	Returns a Promise that resolves with data or rejects with error	You need async/await or to compose multiple writes

MUTATE ASYNC JAVASCRIPT

```javascript
try {
  const savedJob = await addJobMutation.mutateAsync(formValues);
  navigate(`/jobs/${savedJob.id}`);
} catch (error) {
  showToast(error.message);
}
```

Mutation lifecycle callbacks

Callback	When it runs	Common purpose
onMutate	Before mutationFn	Cancel reads and apply optimistic cache update
onSuccess	When mutation resolves	Update cache or invalidate related queries
onError	When mutation rejects	Show error or roll back optimistic changes
onSettled	After success or error	Cleanup, reconciliation, final invalidation

In the current v5 API, hook-level callback signatures include context. The value returned from onMutate is passed to later callbacks as onMutateResult. Callback Promises are awaited before the next lifecycle callback proceeds.

Invalidation after a successful write

INVALIDATE RELATED DATA JAVASCRIPT

```javascript
const updateJob = useMutation({
  mutationFn: ({ id, changes }) => patchJob(id, changes),
  onSuccess: async (updatedJob) => {
    await queryClient.invalidateQueries({
      queryKey: ['jobs'],
    });

    await queryClient.invalidateQueries({
      queryKey: ['job', updatedJob.id],
    });
  },
});
```

Returning or awaiting the invalidation Promise can keep mutation.isPending true until the invalidation/refetch work has settled. This is useful when the UI should not consider the write flow complete until refreshed data is ready.

10 / ADVANCED WRITES

Optimistic updates and rollback

Make the UI feel instant, but plan for server failure.

Two approaches

Approach	How it works	Best when
Optimistic UI using mutation variables	Temporarily render mutation.variables while isPending	Only one component needs to show the temporary item
Optimistic cache update	Write provisional data with setQueryData, snapshot old data, roll back on error	Multiple views should immediately reflect the provisional value

Optimistic update with cache rollback

OPTIMISTIC CACHE UPDATE - V5 CAL JAVASCRIPT

```javascript
const addTodo = useMutation({
  mutationFn: createTodo,

  onMutate: async (newTodo) => {
    await queryClient.cancelQueries({ queryKey: ['todos'] });

    const previousTodos = queryClient.getQueryData(['todos']);

    queryClient.setQueryData(['todos'], (oldTodos = []) => [
      ...oldTodos,
      { ...newTodo, id: `temp-${Date.now()}` },
    ]);

    return { previousTodos };
  },

  onError: (_error, _newTodo, onMutateResult) => {
    queryClient.setQueryData(
      ['todos'],
      onMutateResult?.previousTodos
    );
  },

  onSettled: () => {
    queryClient.invalidateQueries({ queryKey: ['todos'] });
  },
});
```

Why cancel queries first?

An in-flight refetch could finish after your optimistic write and overwrite it with older server data. Cancelling matching queries before applying the optimistic update reduces that race. After success or failure, invalidation reconciles the cache with the server.

Optimistic UI without changing the cache

OPTIMISTIC UI USING VARIABLES JSX

```jsx
const addTodo = useMutation({
  mutationFn: createTodo,
  onSettled: () =>
    queryClient.invalidateQueries({ queryKey: ['todos'] }),
});

return (
  <>
    {todos.map((todo) => <TodoRow key={todo.id} todo={todo} />)}
    {addTodo.isPending && (
      <TodoRow todo={{ id: 'pending', text: addTodo.variables.text }}
               dimmed />
    )}
  </>
);
```

Interview trade-off

Optimistic UI is not always necessary. Use it when perceived latency matters and the result is predictable. For risky or complex updates, a normal pending state followed by invalidation is often simpler and safer.

Concurrent mutations

Multiple mutations can be in flight at the same time. They may complete in a different order from the order they were started. Use stable temporary identifiers, careful rollback logic, and mutation scopes when operations must run serially.

11 / LISTS

Pagination and infinite queries

Two related patterns, two different data shapes.

Paginated useQuery

For numbered pagination, include page/filter values in the query key. Each page is its own query entry. placeholderData: keepPreviousData keeps the previous successful page visible while the new page loads.

PAGE-BASED PAGINATION JSX

```jsx
const [page, setPage] = useState(1);

const jobsQuery = useQuery({
  queryKey: ['jobs', { page }],
  queryFn: () => fetchJobsPage(page),
  placeholderData: keepPreviousData,
});

<button
  disabled={page === 1}
  onClick={() => setPage((old) => old - 1)}
>Previous</button>

<button
  disabled={jobsQuery.isPlaceholderData || !jobsQuery.data?.hasMore}
  onClick={() => setPage((old) => old + 1)}
>Next</button>
```

Infinite queries

useInfiniteQuery keeps multiple pages in one query result. The data shape contains pages and pageParams. In v5, initialPageParam is required. getNextPageParam determines whether another page exists.

CURSOR INFINITE QUERY JSX

```jsx
const jobsQuery = useInfiniteQuery({
  queryKey: ['jobs', 'infinite'],
  queryFn: ({ pageParam }) => fetchJobsCursor(pageParam),
  initialPageParam: null,
  getNextPageParam: (lastPage) => lastPage.nextCursor ?? undefined,
  maxPages: 5,
});

const jobs = jobsQuery.data?.pages.flatMap((page) => page.items) ?? [];

<button
  disabled={!jobsQuery.hasNextPage || jobsQuery.isFetchingNextPage}
  onClick={() => jobsQuery.fetchNextPage()}
>
  {jobsQuery.isFetchingNextPage ? 'Loading more...' : 'Load more'}
</button>
```

Pagination vs infinite query

Page-based useQuery	useInfiniteQuery
Each page is a separate query key	One query key stores multiple pages
Good for page numbers and direct navigation	Good for feeds, timelines, load-more and infinite scroll
Use page number/filter in queryKey	Use pageParam for the next/previous page
Can keep old page visible with placeholderData	Returns fetchNextPage, hasNextPage, isFetchingNextPage

maxPages
maxPages limits how many pages remain in an infinite query cache. If a list needs to support both forward and backward fetching with a bounded window, define both getNextPageParam and getPreviousPageParam.

12 / CONTROL

Cancellation, polling and network modes

Know how requests behave when the user navigates or goes offline.

Request cancellation

Query functions receive a signal. Passing it to fetch allows the browser request to abort when TanStack Query cancels it. By default, an unused query may finish and populate the cache if the signal is not consumed. This is sometimes useful because the result can be reused later.

CONSUME ABORTSIGNAL JAVASCRIPT

```javascript
const searchQuery = useQuery({
  queryKey: ['search', term],
  queryFn: async ({ signal }) => {
    const response = await fetch(`/api/search?q=${term}`, { signal });
    if (!response.ok) throw new Error('Search failed');
    return response.json();
  },
});
```

Polling

refetchInterval repeats a query on a timer. It is independent from staleTime. Poll only when the feature needs periodic updates; consider interval frequency, API cost, tab visibility, and whether background polling should continue.

POLLING JAVASCRIPT

```javascript
useQuery({
  queryKey: ['job-status', jobId],
  queryFn: () => fetchJobStatus(jobId),
  refetchInterval: 5_000,
  refetchIntervalInBackground: false,
});
```

Network modes

Mode	High-level behavior
online	Queries/mutations wait for online status before running. This is the default mode.
always	Run even when the app is offline; useful for sources that do not rely on network connectivity.
offlineFirst	Attempt once even offline, then pause retries until connectivity returns; useful for service-worker or cache-first setups.

Focus and reconnect behavior

Stale queries can refetch when the window regains focus or the network reconnects. This keeps data current after the user returns from another tab or a temporary offline period. Configure refetchOnWindowFocus and refetchOnReconnect deliberately rather than disabling them everywhere.

13 / QUERYCLIENT

Cache operations and prefetching

Imperative APIs are used outside components and after mutations.

QueryClient methods you should know

Method	Purpose	Typical use
invalidateQueries	Mark matching queries stale; normally refetch active matches	After create/update/delete
getQueryData	Read data synchronously from cache	Read an existing snapshot
setQueryData	Write/update one cache entry synchronously	Use a mutation response to update detail cache
setQueriesData	Update existing matching entries	Update a group of cached lists
cancelQueries	Cancel matching in-flight queries	Before optimistic updates
removeQueries	Remove matching entries from cache	Remove private data on account change
resetQueries	Reset to initial state; active matches refetch	Restore query to initial state
clear	Clear query and mutation caches	Logout or full client reset when appropriate

Invalidate vs setQueryData

Invalidation says the cached data may be outdated; active matching queries are normally refetched. setQueryData writes a known result directly into the cache. Prefer direct updates when the mutation response contains the complete, authoritative record; invalidate when several derived lists or aggregates may have changed.

DIRECT UPDATE + INVALIDATION JAVASCRIPT

```javascript
// Update one known detail cache entry immediately
queryClient.setQueryData(
  ['job', updatedJob.id],
  updatedJob
);

// Re-fetch lists because ordering/counts/filters may have changed
await queryClient.invalidateQueries({
  queryKey: ['jobs'],
});
```

Current v5 QueryClient API note

The latest v5 documentation and migration guide include newer imperative methods: queryClient.query and queryClient.infiniteQuery. The migration guide marks older imperative methods such as fetchQuery, prefetchQuery, fetchInfiniteQuery, and prefetchInfiniteQuery as deprecated for removal in v6. Older tutorials still show the previous methods, so recognize both forms when reading code.

CURRENT QUERYCLIENT QUERY API JAVASCRIPT

```javascript
// Current v5 docs: populate the query cache and await data
const jobs = await queryClient.query({
  queryKey: ['jobs'],
  queryFn: fetchJobs,
  staleTime: 30_000,
});

// Prefetch without surfacing the result to the caller
import { noop } from '@tanstack/react-query';

void queryClient.query({
  queryKey: ['job', jobId],
  queryFn: () => fetchJob(jobId),
  staleTime: 60_000,
}).catch(noop);
```

Do not mix examples blindly
When interviewing or joining a codebase, ask which TanStack Query version is installed. The current documentation is v5 and has been evolving within the major version. Use the package version and its migration guide as the source of truth for imperative APIs.

Prefetching before a screen needs data

Prefetching starts a query before a component subscribes to it, such as on hover, focus, or during route loading. The result is placed in the same cache entry and can be reused when the component mounts. Current docs also provide usePrefetchQuery and usePrefetchInfiniteQuery for component lifecycle patterns.

14 / RENDERING

Suspense, SSR and hydration

Prefetch on the server, serialize the cache, hydrate on the client.

Suspense hooks

TanStack Query v5 provides useSuspenseQuery, useSuspenseInfiniteQuery, and useSuspenseQueries. With useSuspenseQuery, data is guaranteed to be defined in the success path, and loading is handled through React Suspense boundaries.

SUSPENSE QUERY JSX

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

function JobDetails({ jobId }) {
  const { data: job } = useSuspenseQuery({
    queryKey: ['job', jobId],
    queryFn: () => fetchJob(jobId),
  });

  return <h1>{job.title}</h1>;
}

function Page({ jobId }) {
  return (
    <Suspense fallback={<p>Loading job...</p>}>
      <JobDetails jobId={jobId} />
    </Suspense>
  );
}
```

Suspense caution
Multiple useSuspenseQuery calls in one component can suspend serially and create a request waterfall. Use useSuspenseQueries when multiple suspenseful queries should run in parallel. An error boundary is also required for errors thrown by Suspense queries.

SSR hydration flow

Server	Client
Create a request-scoped QueryClient	Create or access the browser QueryClient
Prefetch critical queries	Receive the serialized dehydrated cache
Dehydrate the QueryClient cache	Wrap the app/route with HydrationBoundary
Render HTML with the prefetched data	Hydrate the cache and reuse data instead of refetching immediately

SSR + HYDRATION CONCEPT JSX

```jsx
// Server-side sketch (framework-specific integration varies)
const queryClient = new QueryClient();

await queryClient.query({
  queryKey: ['job', jobId],
  queryFn: () => fetchJobOnServer(jobId),
  staleTime: 60_000,
});

const dehydratedState = dehydrate(queryClient);

// Client component tree
<QueryClientProvider client={queryClient}>
  <HydrationBoundary state={dehydratedState}>
    <App />
  </HydrationBoundary>
</QueryClientProvider>
```

Why request-scoped QueryClient on the server?

A module-global server QueryClient can accidentally share cached user-specific data across requests. Create a new QueryClient for each server request or request context. Authorize private reads on the server regardless of query keys; query keys are cache identity, not an access-control boundary.

Why set staleTime for SSR?

The default staleTime is 0, so server-prefetched data can immediately be considered stale and refetch in the browser. A suitable staleTime reduces duplicate requests during hydration while still allowing later revalidation.

15 / DEVELOPER WORKFLOW

Devtools, testing and debugging

Make cache state observable and tests deterministic.

React Query Devtools

DEVTOOLS JSX

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

<QueryClientProvider client={queryClient}>
  <App />
  <ReactQueryDevtools initialIsOpen={false} />
</QueryClientProvider>
```

Devtools help inspect query keys, status, stale/fresh state, data, observers, and cache contents. They are particularly useful when debugging duplicate requests, unexpected refetching, or invalidation that appears not to work.

Testing a custom query hook

Create a fresh QueryClient per test or clear the cache between tests. Disable retries for predictable error tests; default retries can make a failing test slow.

HOOK TESTING PATTERN JSX

```jsx
import { renderHook, waitFor } from '@testing-library/react';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

function createWrapper() {
  const queryClient = new QueryClient({
    defaultOptions: { queries: { retry: false } },
  });

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

test('loads jobs', async () => {
  const { result } = renderHook(() => useJobs(), {
    wrapper: createWrapper(),
  });

  await waitFor(() => expect(result.current.isSuccess).toBe(true));
  expect(result.current.data).toHaveLength(2);
});
```

Testing checklist

- Use a QueryClient isolated to the test to prevent cache leakage.
- Disable query retries for tests that intentionally verify errors.

- Mock the network boundary (for example with MSW) instead of mocking TanStack Query internals.
- Wait for the observable state with waitFor; do not rely on arbitrary sleeps.
- Clean up timers, subscriptions, and persisted cache between tests.
- Test invalidation after a mutation and rollback after an optimistic failure.

Debugging common symptoms

Symptom	Likely cause / first check
Request happens twice	StrictMode development behavior, staleTime 0, multiple QueryClients, duplicate components, or query keys that differ
Query never starts	enabled false, missing provider, query function disabled, or dependent input still undefined
Data looks old	Missing invalidation, wrong key, staleTime choice, mutation updates wrong cache entry
Spinner never stops	QueryFn never settles, unresolved Promise, network request hangs, or retry behavior
A 404 appears as success	fetch response was returned without checking response.ok
Cache disappears every render	QueryClient is recreated during render
Old user data appears after account switch	Private queries not removed/cancelled, user identity missing from key, or server-side cache shared

16 / PRODUCTION DESIGN

Performance and architecture

Use the cache intentionally; do not fetch or invalidate indiscriminately.

Render optimizations

TanStack Query uses structural sharing for JSON-compatible data to retain references for unchanged parts of a response. Components subscribe to the result properties they use. select can reduce the observed data. Avoid disabling structural sharing without a measured reason.

Practical performance checklist

- Choose realistic staleTime values per resource: rapidly changing status data differs from reference data.
- Keep query keys stable, serializable, and complete. Include filters, sort, page, tenant, and entity IDs when they affect the result.
- Use pagination for large collections and maxPages for long-lived infinite lists.
- Use select for derived subsets; avoid expensive calculations on every render.
- Prefetch data when likely to be needed, but avoid fetching every possible route in advance.
- Cancel obsolete requests where the transport supports AbortSignal.
- Use targeted invalidation rather than invalidating the entire cache after every mutation.
- Use a shared cache such as Redis on the server if backend request caching is needed; TanStack Query is a browser/application cache, not a server database cache.

A maintainable feature architecture

SEPARATE API AND QUERY DEFINITIO JAVASCRIPT

```javascript
// api/jobs.js
export async function getJobs(filters, signal) {
  const params = new URLSearchParams(filters);
  const response = await fetch(`/api/jobs?${params}`, { signal });
  if (!response.ok) throw new Error('Unable to load jobs');
  return response.json();
}

// queries/jobQueries.js
import { queryOptions } from '@tanstack/react-query';
import { getJobs } from '../api/jobs';

export const jobKeys = {
  all: ['jobs'],
  list: (filters) => ['jobs', 'list', filters],
};

export const jobsOptions = (filters) => queryOptions({
  queryKey: jobKeys.list(filters),
  queryFn: ({ signal }) => getJobs(filters, signal),
  staleTime: 30_000,
});
```

TanStack Query with Zustand

Put in TanStack Query	Put in Zustand / local state
Fetched list of projects	Current sidebar selection
Server user profile	Open/closed modal state
Product detail from API	Unsaved client-only wizard step
Orders and server pagination	Temporary client filters before user applies them

A useful rule: if a value is owned by the backend and can be refetched, use the query cache. If it represents a local UI decision, use local/global client state. URL search params are often the best home for shareable filters and pagination.

Security and account isolation

- Never treat a query key as authorization; the backend must authorize every request.
- Include account/tenant identity in keys if it changes the returned resource.
- On logout/account switching, cancel and remove private query data so stale user data is not shown.
- Do not publicly cache personalized SSR HTML or private API responses at a CDN.

17 / VERSION AWARENESS

v4 to v5 and current v5 notes

Interviewers often test whether you know the modern object syntax.

Major v5 changes

Older style / concept	v5 approach
Multiple positional overloads such as useQuery(key, fn, options)	Single object form: useQuery({ queryKey, queryFn, ...options })
cacheTime	Renamed gcTime to clarify inactive-cache garbage collection
keepPreviousData: true	placeholderData: keepPreviousData helper
isLoading as first-load state	isPending is status; isLoading means isPending && isFetching
useErrorBoundary option	throwOnError option
onSuccess/onError/onSettled callbacks on useQuery	Removed from query hook options; mutation callbacks remain
initialPageParam could be omitted in older examples	Required for useInfiniteQuery in v5
Removed query callbacks / status shape changes	Revisit UI state checks and custom hook callback assumptions

v5 object syntax

MIGRATION EXAMPLE JAVASCRIPT

```javascript
// v4 positional style (legacy)
useQuery(['jobs', page], () => fetchJobs(page), {
  keepPreviousData: true,
});

// v5 object style
useQuery({
  queryKey: ['jobs', page],
  queryFn: () => fetchJobs(page),
  placeholderData: keepPreviousData,
});
```

Current late-v5 documentation change

As of the official docs checked on 13 November 2026, the latest v5 release shown by npm/GitHub is 5.107.1. The current migration guide introduces queryClient.query and queryClient.infiniteQuery and marks older imperative methods such as fetchQuery/prefetchQuery and their infinite variants as deprecated for removal in v6. This is newer than many v5 tutorials. Always check the exact installed package and migration guide.

> Version-aware interview answer
> I use the current v5 object syntax. If I inherit an older codebase, I check its installed version and migration guide because v5 changed query states, pagination placeholders, callback behavior, and imperative QueryClient APIs.

Common v5 mistakes

- Copying a v4 positional useQuery signature into a v5 project.
- Using cacheTime instead of gcTime.
- Using keepPreviousData: true instead of placeholderData: keepPreviousData.
- Assuming isPending always means a network request is currently running.
- Adding onSuccess to useQuery - query callbacks were removed in v5.
- Using a module-global QueryClient on the server.
- Using old prefetchQuery/fetchQuery examples without checking newer v5 migration notes.

18 / INTERVIEW BANK

Questions and answers

Use these to practice concise, confident explanations.

1. What is TanStack Query?

A server-state management library for fetching, caching, synchronizing, and updating asynchronous data in React applications.

2. What is server state?

Data owned by a remote system that the frontend reads or changes. It may be stale, shared, and updated outside the current component.

3. Why not just use useEffect and fetch?

You can, but then you must build caching, deduplication, retries, cancellation, background refetching, race handling, and invalidation yourself.

4. What is QueryClient?

The central object that manages query/mutation caches and exposes cache operations and defaults.

5. Why QueryClientProvider?

It makes a stable QueryClient available to descendants through React context.

6. What are queryKey and queryFn?

queryKey identifies the cached data; queryFn returns a Promise that resolves the data or throws an error.

7. Why should variables be inside the query key?

The key must describe the exact data requested. Changing an input should produce the correct separate cache entry.

8. What is a query cache hit?

The requested query key already has data in cache; it can be reused depending on freshness and refetch policy.

9. staleTime vs gcTime?

staleTime controls how long data is fresh. gcTime controls how long inactive cache data remains before garbage collection.

10. What is the default staleTime?

Zero. Cached data is considered stale immediately unless configured otherwise.

11. What happens when stale data is mounted?

It can render from cache and refetch in the background, depending on refetch options.

12. What triggers background refetches by default?

Stale queries can refetch on mount, window focus, and reconnect.

13. What is the difference between isPending and isFetching?

isPending means no successful data is available yet; isFetching means a request is currently running.

14. What is isLoading in v5?

A first fetch is actively running; it is derived from isPending && isFetching.

15. Why must a fetch queryFn check response.ok?

fetch resolves for HTTP error statuses too; it does not reject solely because the server returned 404 or 500.

16. What is enabled used for?

It controls whether a query should automatically execute. It is useful for dependent or user-triggered queries.

17. What are dependent queries?

Queries whose execution depends on the result of another query, often using enabled.

18. What are parallel queries?

Independent queries running at the same time to reduce total waiting time.

19. useQuery vs useMutation?

useQuery reads/caches asynchronous data. useMutation performs a write or side effect such as create, update, or delete.

20. mutate vs mutateAsync?

mutate triggers the mutation and uses state/callbacks; mutateAsync returns a Promise for async/await composition.

21. What is invalidation?

Marking matching queries stale so active ones can refetch and inactive ones can refresh when used later.

22. setQueryData vs invalidateQueries?

setQueryData directly writes known data to cache. invalidateQueries marks data stale and usually refetches active matches.

23. What is an optimistic update?

Updating the UI/cache as though a server write succeeded before the server confirms it, with a rollback plan if it fails.

24. Why cancel queries in onMutate?

To stop an in-flight read from overwriting the optimistic cache update with an older response.

25. What is placeholderData?

Temporary data displayed while the current query result is not ready; commonly used to preserve the previous page during pagination.

26. initialData vs placeholderData?

initialData seeds the query cache. placeholderData is temporary observer-level data and is not persisted as query data.

27. useInfiniteQuery vs useQuery pagination?

useInfiniteQuery stores multiple pages under one key; page-based useQuery gives each page its own key.

28. What does maxPages do?

Limits how many pages are retained in an infinite query, helping control memory and refetch work.

29. How do you cancel a request?

Consume the AbortSignal passed to queryFn and pass it to fetch; imperative cancellation is available with cancelQueries.

30. What are retries?

Automatic attempts after failed queries. The browser default is 3; configure retry based on the error type.

31. Does a mutation retry by default?

No. Mutation retries are disabled by default unless configured.

32. What does select do?

Transforms the data exposed by a query observer without changing the cached source data.

33. What is structural sharing?

An optimization that preserves references for unchanged parts of JSON-compatible results to reduce unnecessary renders.

34. What is prefetching?

Fetching data before a component needs it and placing it in the query cache to make navigation or rendering faster.

35. What is hydration?

Restoring a serialized query cache, commonly fetched on the server, into the client cache.

36. Why request-scope QueryClient in SSR?

To prevent one server request from seeing another user request's cached private data.

37. How do you test a query hook?

Render it under QueryClientProvider with an isolated client, disable retries as needed, await state with waitFor, and mock the network boundary.

38. What is the role of Devtools?

Inspect cache entries, query keys, status, freshness, observers, and request/refetch behavior.

39. Is TanStack Query a replacement for Zustand?

No. TanStack Query manages remote server data; Zustand is suited to client-owned global/UI state.

40. What should you invalidate after updating a job?

The job detail and any list/count/filter queries whose server results may have changed. Use key hierarchy to target them precisely.

18 / PRACTICAL ROUNDS

Real interview scenarios

Explain the design, not only the hook syntax.

Scenario 1 - Job list refetches on every revisit

Explain staleTime. Decide how long job listings can be considered fresh. Add a realistic staleTime and keep refetch-on-focus if returning users should see changes.

Scenario 2 - Search results show out-of-order responses

Include searchTerm in queryKey. Consume the AbortSignal in the queryFn so obsolete requests can be cancelled. Do not store unrelated searches under one key.

Scenario 3 - Update a job but list still shows old title

After mutation success, update the detail cache if response is authoritative and invalidate affected job lists. Confirm that the mutation and query key factory use the same key shape.

Scenario 4 - Disabled dependent query shows a spinner forever

Check isLoading vs isPending. A disabled query can be pending but idle; display a waiting prompt until the parent ID exists.

Scenario 5 - Pagination flashes a loading page

Include page in queryKey and use placeholderData: keepPreviousData. Use isPlaceholderData to disable unsafe next-page actions until the new response arrives.

Scenario 6 - Infinite scroll grows memory endlessly

Use maxPages, ensure cursor logic is correct, and avoid loading duplicate pages while isFetchingNextPage is true.

Scenario 7 - Optimistic update disappears after a failed write

Snapshot cache in onMutate, return the snapshot, restore it in onError, then invalidate onSettled to reconcile with the server.

Scenario 8 - A user logs out but their private data remains

Cancel and remove private queries, clear sensitive cache as appropriate, and ensure every backend endpoint authorizes requests. Cache keys are not security.

Scenario 9 - Duplicate request immediately after SSR

Check per-request server QueryClient, dehydrate/hydrate wiring, matching query keys, QueryClient stability, and staleTime. Do not disable all refetching as a workaround.

Scenario 10 - Test suite hangs on a failed query

Set retry: false in test QueryClient defaults, use an isolated client, and wait for the expected error state. Clean timers and cache after each test.

Practical coding tasks to rehearse

- Create a useJobs(filters) hook with stable keys, queryOptions, and fetch error handling.
- Create a job detail query that is disabled until jobId is available.
- Build a form mutation that updates a job, invalidates the list, and updates the detail cache.
- Implement a paginated jobs table using keepPreviousData and hasMore metadata.
- Implement cursor-based infinite scrolling with initialPageParam and maxPages.
- Add an optimistic favourite/unfavourite toggle with rollback.
- Write a test for a custom useJob hook using a fresh QueryClient.
- Explain how the same route can be prefetched, dehydrated, and hydrated in SSR.

19 / LAST-MINUTE PREP

Revision sheet and glossary

A compact page to revisit before the interview.

Remember these API patterns

QUICK API REFERENCE JAVASCRIPT

```javascript
// Read
useQuery({ queryKey: ['jobs'], queryFn: fetchJobs });

// Conditional read
useQuery({ queryKey: ['job', id], queryFn: () => fetchJob(id), enabled: !!id });

// Write
useMutation({ mutationFn: createJob, onSuccess: () => queryClient.invalidateQueries({ queryKey: ['jobs'] }) });

// Manual cache write
queryClient.setQueryData(['job', id], updatedJob);

// Invalidate
queryClient.invalidateQueries({ queryKey: ['jobs'] });

// Infinite
useInfiniteQuery({ queryKey: ['feed'], queryFn: fetchPage, initialPageParam: null, getNextPageParam: (last) => last.nextCursor });

// Current v5 prefetch/query cache population
queryClient.query({ queryKey: ['jobs'], queryFn: fetchJobs });
```

High-priority concepts

Concept	One-line reminder
Query	A declarative cached read
Mutation	A server write / side effect
queryKey	Identity of the remote data
queryFn	Promise that returns data or throws
staleTime	Freshness window
gcTime	Inactive cache retention
isPending	No data has resolved yet
isFetching	A request is currently running
invalidateQueries	Mark matching cache entries stale
setQueryData	Write known data directly
enabled	Gate automatic query execution
placeholderData	Temporary display data
keepPreviousData	Keep prior page while next key loads
AbortSignal	Cooperative request cancellation
maxPages	Bound retained infinite-query pages

Do not say these in an interview

- "TanStack Query is just Axios." - Axios sends requests; Query also manages cache and lifecycle.
- "staleTime and gcTime are the same." - one is freshness; one is inactive cache retention.
- "isPending means the network is loading." - a disabled query may be pending and idle.
- "A mutation automatically updates all lists." - you must update or invalidate relevant cache entries.
- "The query key is an auth check." - the server must authorize requests independently.
- "Just copy all API data into Zustand." - this can create duplicate sources of truth.

Official documentation and references

TanStack Query React documentation (v5)
https://tanstack.com/query/latest/docs/framework/react

Important defaults
https://tanstack.com/query/latest/docs/framework/react/guides/important-defaults

Queries and query states
https://tanstack.com/query/latest/docs/framework/react/guides/queries

Query keys
https://tanstack.com/query/latest/docs/framework/react/guides/query-keys

Query options
https://tanstack.com/query/latest/docs/framework/react/guides/query-options

Mutations
https://tanstack.com/query/latest/docs/framework/react/guides/mutations

Invalidations from mutations
https://tanstack.com/query/latest/docs/framework/react/guides/invalidations-from-mutations

Optimistic updates
https://tanstack.com/query/latest/docs/framework/react/guides/optimistic-updates

Query cancellation
https://tanstack.com/query/latest/docs/framework/react/guides/query-cancellation

Paginated queries
https://tanstack.com/query/latest/docs/framework/react/guides/paginated-queries

Infinite queries
https://tanstack.com/query/latest/docs/framework/react/guides/infinite-queries

Dependent queries
https://tanstack.com/query/latest/docs/framework/react/guides/dependent-queries

Server rendering and hydration
https://tanstack.com/query/latest/docs/framework/react/guides/ssr

Testing
https://tanstack.com/query/latest/docs/framework/react/guides/testing

Migration to v5
https://tanstack.com/query/latest/docs/framework/react/guides/migrating-to-v5

npm package / current release
https://www.npmjs.com/package/@tanstack/react-query

> Final interview strategy
> Be able to explain one complete feature end-to-end: define a stable key, fetch with useQuery, model loading/error/background refresh, update with useMutation, invalidate or patch cache, handle optimistic failure, and explain how you would test it. That is stronger than memorizing every option.