React Query (TanStack Query v3)

Version - **3.39.3**

https://tanstack.com/query/v3

https://tanstack.com/query/latest (v4)

Overview

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

Motivation

Out of the box, React applications do not come with an opinionated way of fetching or updating data from your components so developers end up building their own ways of fetching data. This usually means cobbling together component-based state and effect using React hooks, or using more general purpose state management libraries to store and provide asynchronous data throughout their apps.

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

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- Can potentially become "out of date" in your applications if you're not careful

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

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- Updating "out of date" data in the background
- Knowing when data is "out of date"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

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

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

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

On a more technical note, React Query will likely:

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

[Enough talk, show me some code already!]

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

const queryClient = new QueryClient()

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

function Example() {
 const { isLoading, error, data } = useQuery('repoData', () =>
  fetch('https://api.github.com/repos/tannerlinsley/react-query').then(res =>
   res.json()
  )
 )

 if (isLoading) return 'Loading...'

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

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

Important Defaults

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

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

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

- Stale queries are refetched automatically in the background when:
  - New instances of the query mount
  - The window is refocused
  - The network is reconnected.
  - The query is optionally configured with a refetch interval.

If you see a refetch that you are not expecting, it is likely because you just focused the window and React Query is doing a `refetchOnWindowFocus`. During development, this will probably be triggered more frequently, especially because focusing between the Browser DevTools and your app will also cause a fetch, so be aware of that.

To change this functionality, you can use options like `refetchOnMount`, `refetchOnWindowFocus`, `refetchOnReconnect` and `refetchInterval`.

- Query results that have no more active instances of `useQuery`, `useInfiniteQuery` or query observers are labeled as "inactive" and remain in the cache in case they are used again at a later time.
- By default, "inactive" queries are garbage collected after **5 minutes**.

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

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

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

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

Structural sharing only works with JSON-compatible values, any other value types will always be considered as changed. If you are seeing performance issues because of large responses for example, you can disable this feature with the ***`config.structuralSharing`*** *flag. If you are dealing with non-JSON compatible values in your query responses and still want to detect if data has changed or not, you can define a data compare function with* ***`config.isDataEqual`***.

Queries

Query Basics

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

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

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

Js

```
import { useQuery } from 'react-query'

function App() {
  const info = useQuery('todos', fetchTodoList)
}
```

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

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

```
const result = useQuery('todos', fetchTodoList)
```

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

- `isLoading` or `status === 'loading'` - The query has no data and is currently fetching
- `isError` or `status === 'error'` - The query encountered an error
- `isSuccess` or `status === 'success'` - The query was successful and data is available
- `isIdle` or `status === 'idle'` - The query is currently disabled (you'll learn more about this in a bit)

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

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

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

```
function Todos() {
  const { isLoading, isError, data, error } = useQuery('todos', fetchTodoList)

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

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

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

If booleans aren't your thing, you can always use the `status` state as well:

js

```
function Todos() {
  const { status, data, error } = useQuery('todos', fetchTodoList)

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

  if (status === 'error') {
```

```
  return <span>Error: {error.message}</span>
 }

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

Query Keys

At its core, React Query manages query caching for you based on query keys. Query keys can be as simple as a string, or as complex as an array of many strings and nested objects. As long as the query key is serializable, and unique to the query's data, you can use it!

String-Only Query Keys

The simplest form of a key is actually not an array, but an individual string. When a string query key is passed, it is converted to an array internally with the string as the only item in the query key. This format is useful for:

Generic List/Index resources

Non-hierarchical resources

js

```
// A list of todos
useQuery('todos', ...) // queryKey === ['todos']

// Something else, whatever!
useQuery('somethingSpecial', ...) // queryKey === ['somethingSpecial']
```

Array Keys

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

Hierarchical or nested resources

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

Queries with additional parameters

It's common to pass an object of additional options

js

```
// An individual todo
useQuery(['todo', 5], ...)
// queryKey === ['todo', 5]

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

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

Query Keys are hashed deterministically!

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

js

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

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

js

```
useQuery(['todos', status, page], ...)
useQuery(['todos', page, status], ...)
useQuery(['todos', undefined, page, status], ...)
```

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

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

js

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

```
}
```

Query Functions

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

All of the following are valid query function configurations:

js

```
useQuery(['todos'], fetchAllTodos)

useQuery(['todos', todoId], () => fetchTodoById(todoId))

useQuery(['todos', todoId], async () => {
  const data = await fetchTodoById(todoId)
  return data
})

useQuery(['todos', todoId], ({ queryKey }) => fetchTodoById(queryKey[1]))
```

Handling and Throwing Errors

For React Query to determine a query has errored, the query function must throw. Any error that is thrown in the query function will be persisted on the error state of the query.

js

```
const { error } = useQuery(['todos', todoId], async () => {
  if (somethingGoesWrong) {
    throw new Error('Oh no!')
  }


  return data})
```

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

While most utilities like axios or graphql-request automatically throw errors for unsuccessful HTTP calls, some utilities like fetch do not throw errors by default. If that's the case, you'll need to throw them on your own. Here is a simple way to do that with the popular fetch API:

js

```
useQuery(['todos', todoId], async () => {
  const response = await fetch('/todos/' + todoId)
```

```
  if (!response.ok) {
    throw new Error('Network response was not ok')
  }
  return response.json()
})
```

Query Function Variables

Query keys are not just for uniquely identifying the data you are fetching, but are also conveniently passed into your query function and while not always necessary, this makes it possible to extract your query functions if needed:

js

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


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

Using a Query Object instead of parameters

Anywhere the [queryKey, queryFn, config] signature is supported throughout React Query's API, you can also use an object to express the same configuration:

js

```
import { useQuery } from 'react-query'


useQuery({
  queryKey: ['todo', 7],
  queryFn: fetchTodo,
  ...config,
```

```
})
```

Parallel Queries

"Parallel" queries are queries that are executed in parallel, or at the same time so as to maximize fetching concurrency.

Manual Parallel Queries

When the number of parallel queries does not change, there is no extra effort to use parallel queries. Just use any number of React Query's useQuery and useInfiniteQuery hooks side-by-side!

js

```
function App () {
  // The following queries will execute in parallel
  const usersQuery = useQuery('users', fetchUsers)
  const teamsQuery = useQuery('teams', fetchTeams)
  const projectsQuery = useQuery('projects', fetchProjects)
  ...
}
```

When using React Query in suspense mode, this pattern of parallelism does not work, since the first query would throw a promise internally and would suspend the component before the other queries run. To get around this, you'll either need to use the useQueries hook (which is suggested) or orchestrate your own parallelism with separate components for each useQuery instance (which is lame).

Dynamic Parallel Queries with useQueries

If the number of queries you need to execute is changing from render to render, you cannot use manual querying since that would violate the rules of hooks. Instead, React Query provides a useQueries hook, which you can use to dynamically execute as many queries in parallel as you'd like.

useQueries accepts an array of query options objects and returns an array of query results:

js

```
function App({ users }) {
  const userQueries = useQueries(
    users.map(user => {
      return {
        queryKey: ['user', user.id],
        queryFn: () => fetchUserById(user.id),
      }
```

```
    })
  )
}
```

Dependent Queries

Dependent (or serial) queries depend on previous ones to finish before they can execute. To achieve this, it's as easy as using the enabled option to tell a query when it is ready to run:

js

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

const userId = user?.id

// Then get the user's projects
const { isIdle, data: projects } = useQuery(
  ['projects', userId],
  getProjectsByUser,
  {
    // The query will not execute until the userId exists
    enabled: !!userId,
  }
)


// isIdle will be `true` until `enabled` is true and the query begins to fetch.
// It will then go to the `isLoading` stage and hopefully the `isSuccess` stage :)
```

Background Fetching Indicators

A query's status === 'loading' state is sufficient enough to show the initial hard-loading state for a query, but sometimes you may want to display an additional indicator that a query is refetching in the background. To do this, queries also supply you with an isFetching boolean that you can use to show that it's in a fetching state, regardless of the state of the status variable:

js

```
function Todos() {
  const { status, data: todos, error, isFetching } = useQuery(
    'todos',
    fetchTodos
  )

  return status === 'loading' ? (
    <span>Loading...</span>
  ) : status === 'error' ? (
    <span>Error: {error.message}</span>
```

```
  ) : (
    <>
      {isFetching ? <div>Refreshing...</div> : null}

      <div>
        {todos.map(todo => (
          <Todo todo={todo} />
        ))}
      </div>
    </>
  )
}
```

Displaying Global Background Fetching Loading State

In addition to individual query loading states, if you would like to show a global loading indicator when any queries are fetching (including in the background), you can use the useIsFetching hook:

js

```
import { useIsFetching } from 'react-query'

function GlobalLoadingIndicator() {
  const isFetching = useIsFetching()

  return isFetching ? (
    <div>Queries are fetching in the background...</div>
  ) : null
}
```

Window Focus Refetching

If a user leaves your application and returns to stale data, React Query automatically requests fresh data for you in the background. You can disable this globally or per-query using the refetchOnWindowFocus option:

Disabling Globally

js

```
//
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      refetchOnWindowFocus: false,
    },
  },
```

```
})

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

Disabling Per-Query

js

```
useQuery('todos', fetchTodos, { refetchOnWindowFocus: false })
```

Custom Window Focus Event

In rare circumstances, you may want to manage your own window focus events that trigger React Query to revalidate. To do this, React Query provides a focusManager.setEventListener function that supplies you the callback that should be fired when the window is focused and allows you to set up your own events. When calling focusManager.setEventListener, the previously set handler is removed (which in most cases will be the default handler) and your new handler is used instead. For example, this is the default handler:

js

```
focusManager.setEventListener(handleFocus => {
  // Listen to visibilitychange and focus
  if (typeof window !== 'undefined' && window.addEventListener) {
    window.addEventListener('visibilitychange', handleFocus, false)
    window.addEventListener('focus', handleFocus, false)
  }

  return () => {
    // Be sure to unsubscribe if a new handler is set
    window.removeEventListener('visibilitychange', handleFocus)
    window.removeEventListener('focus', handleFocus)
  }
})
```

Ignoring Iframe Focus Events

A great use-case for replacing the focus handler is that of iframe events. Iframes present problems with detecting window focus by both double-firing events and also firing false-positive events when focusing or using iframes within your app. If you experience this, you should use an event handler that ignores these events as much as possible. I recommend this one! It can be set up in the following way:

js

```
import { focusManager } from 'react-query'
```

```
import onWindowFocus from './onWindowFocus' // The gist above

focusManager.setEventListener(onWindowFocus) // Boom!
```

Managing Focus in React Native

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

js

```
import { AppState } from 'react-native'
import { focusManager } from 'react-query'

focusManager.setEventListener(handleFocus => {
  const subscription = AppState.addEventListener('change', state => {
    handleFocus(state === 'active')
  })

  return () => {
    subscription.remove()
  }
})
```

Managing focus state

js

```
import { focusManager } from 'react-query'

// Override the default focus state
focusManager.setFocused(true)

// Fallback to the default focus check
focusManager.setFocused(undefined)
```

Pitfalls & Caveats

Some browser internal dialogue windows, such as spawned by alert() or file upload dialogues (as created by <input type="file" />) might also trigger focus refetching after they close. This can result in unwanted side effects, as the refetching might trigger component unmounts or remounts before your file upload handler is executed. See this issue on GitHub for background and possible workarounds.

Disabling/Pausing Queries

If you ever want to disable a query from automatically running, you can use the enabled = false option.

When enabled is false:

- If the query has cached data
  - The query will be initialized in the `status === 'success'` or `isSuccess` state.
- If the query does not have cached data
  - The query will start in the `status === 'idle'` or `isIdle` state.
- The query will not automatically fetch on mount.
- The query will not automatically refetch in the background when new instances mount or new instances appearing
- The query will ignore query client `invalidateQueries` and `refetchQueries` calls that would normally result in the query refetching.
- `refetch` can be used to manually trigger the query to fetch.

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

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



   {isIdle ? (
    'Not ready...'
   ) : isLoading ? (
    <span>Loading...</span>
   ) : isError ? (
    <span>Error: {error.message}</span>
   ) : (
    <>
     <ul>
      {data.map(todo => (
       <li key={todo.id}>{todo.title}</li>
      ))}
     </ul>
     <div>{isFetching ? 'Fetching...' : null}</div>
    </>
   )}
  </>
```

```
  )
}
```

Query Retries

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

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

Setting retry = false will disable retries.

Setting retry = 6 will retry failing requests 6 times before showing the final error thrown by the function.

Setting retry = true will infinitely retry failing requests.

Setting retry = (failureCount, error) => ... allows for custom logic based on why the request failed.

```
import { useQuery } from 'react-query'

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

Retry Delay

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

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

```
// Configure for all queries
import { QueryCache, QueryClient, QueryClientProvider } from 'react-query'

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

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

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

js

```
const result = useQuery('todos', fetchTodoList, {
  retryDelay: 1000, // Will always wait 1000ms to retry, regardless of how many retries
})
```

Paginated / Lagged Queries

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

js

```
const result = useQuery(['projects', page], fetchProjects)
```

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

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

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

Better Paginated Queries with keepPreviousData

Consider the following example where we would ideally want to increment a pageIndex (or cursor) for a query. If we were to use useQuery, it would still technically work fine, but the UI would jump in and out of the success and loading states as different queries are created and destroyed for each page or cursor. By setting keepPreviousData to true we get a few new things:

The data from the last successful fetch available while new data is being requested, even though the query key has changed.

When the new data arrives, the previous data is seamlessly swapped to show the new data.

isPreviousData is made available to know what data the query is currently providing you

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

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

```
  const {
    isLoading,
    isError,
    error,
    data,
    isFetching,
    isPreviousData,
  } = useQuery(['projects', page], () => fetchProjects(page), { keepPreviousData : true })

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

[Lagging Infinite Query results with keepPreviousData]

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

Infinite Queries

Rendering lists that can additively "load more" data onto an existing set of data or "infinite scroll" is also a very common UI pattern. React Query supports a useful version
of useQuery called useInfiniteQuery for querying these types of lists.

When using useInfiniteQuery, you'll notice a few things are different:

- data is now an object containing infinite query data:
- data.pages array containing the fetched pages
- data.pageParams array containing the page params used to fetch the pages
- The fetchNextPage and fetchPreviousPage functions are now available
- The getNextPageParam and getPreviousPageParam options are available for both determining if there is more data to load and the information to fetch it. This information is supplied as an additional parameter in the query function (which can optionally be overridden when calling the fetchNextPage or fetchPreviousPage functions)
- A hasNextPage boolean is now available and is true if getNextPageParam returns a value other than undefined
- A hasPreviousPage boolean is now available and is true if getPreviousPageParam returns a value other than undefined
- The isFetchingNextPage and isFetchingPreviousPage booleans are now available to distinguish between a background refresh state and a loading more state

> Note: When using options like initialData or select in your query, make sure that when you restructure your data that it still includes data.pages and data.pageParams properties, otherwise your changes will be overwritten by the query in its return!

Example

Let's assume we have an API that returns pages of projects 3 at a time based on a cursor index along with a cursor that can be used to fetch the next group of projects:

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

With this information, we can create a "Load More" UI by:

- Waiting for useInfiniteQuery to request the first group of data by default
- Returning the information for the next query in getNextPageParam
- Calling fetchNextPage function

Note: It's very important you do not call fetchNextPage with arguments unless you want them to override the pageParam data returned from the getNextPageParam function. e.g. Do not do this: <button onClick={fetchNextPage} /> as this would send the onClick event to the fetchNextPage function.

```
import { useInfiniteQuery } from 'react-query'

function Projects() {
 const fetchProjects = ({ pageParam = 0 }) =>
  fetch('/api/projects?cursor=' + pageParam)

 const {
  data,
  error,
  fetchNextPage,
  hasNextPage,
  isFetching,
  isFetchingNextPage,
  status,
 } = useInfiniteQuery('projects', fetchProjects, {
  getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
 })

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

```
      </button>
    </div>
    <div>{isFetching && !isFetchingNextPage ? 'Fetching...' : null}</div>
  </>
 )
}
```

What happens when an infinite query needs to be refetched?

When an infinite query becomes stale and needs to be refetched, each group is fetched sequentially, starting from the first one. This ensures that even if the underlying data is mutated, we're not using stale cursors and potentially getting duplicates or skipping records. If an infinite query's results are ever removed from the queryCache, the pagination restarts at the initial state with only the initial group being requested.

refetchPage

If you only want to actively refetch a subset of all pages, you can pass the refetchPage function to refetch returned from useInfiniteQuery.

js

```
const { refetch } = useInfiniteQuery('projects', fetchProjects, {
  getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
})

// only refetch the first page
refetch({ refetchPage: (page, index) => index === 0 })
```

You can also pass this function as part of the 2nd argument (queryFilters) to queryClient.refetchQueries, queryClient.invalidateQueries or queryClient.resetQueries.

Signature

refetchPage: (page: TData, index: number, allPages: TData[]) => boolean

The function is executed for each page, and only pages where this function returns true will be refetched.

What if I need to pass custom information to my query function?

By default, the variable returned from getNextPageParam will be supplied to the query function, but in some cases, you may want to override this. You can pass custom variables to the fetchNextPage function which will override the default variable like so:

```
function Projects() {
  const fetchProjects = ({ pageParam = 0 }) =>
    fetch('/api/projects?cursor=' + pageParam)

  const {
    status,
    data,
    isFetching,
    isFetchingNextPage,
    fetchNextPage,
    hasNextPage,
  } = useInfiniteQuery('projects', fetchProjects, {
    getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
  })

  // Pass your own page param
  const skipToCursor50 = () => fetchNextPage({ pageParam: 50 })
}
```

[What if I want to implement a bi-directional infinite list?]

Bi-directional lists can be implemented by using
the getPreviousPageParam, fetchPreviousPage, hasPreviousPage and isFetchingPreviousPage properties and functions.

js

```
useInfiniteQuery('projects', fetchProjects, {

  getNextPageParam: (lastPage, pages) => lastPage.nextCursor,

  getPreviousPageParam: (firstPage, pages) => firstPage.prevCursor,

})
```

[What if I want to show the pages in reversed order?]

Sometimes you may want to show the pages in reversed order. If this is case, you can use
the select option:

js

```
useInfiniteQuery('projects', fetchProjects, {

  select: data => ({

    pages: [...data.pages].reverse(),

    pageParams: [...data.pageParams].reverse(),

  }),
```

```
})
```

What if I want to manually update the infinite query?

Manually removing first page:

js

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

Manually removing a single value from an individual page:

js

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

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

Make sure to keep the same data structure of pages and pageParams!

Initial Query Data

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

- Declaratively:
- Provide initialData to a query to prepopulate its cache if empty
- Imperatively:

- Prefetch the data using queryClient.prefetchQuery
- Manually place the data into the cache using queryClient.setQueryData

Using initialData to prepopulate a query

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

IMPORTANT: initialData is persisted to the cache, so it is not recommended to provide placeholder, partial or incomplete data to this option and instead use placeholderData

js

```
function Todos() {
  const result = useQuery('todos', () => fetch('/todos'), {
    initialData: initialTodos,
  })
}
```

staleTime and initialDataUpdatedAt

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

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

```
function Todos() {

  // Will show initialTodos immediately, but also immediately refetch todos after mount

  const result = useQuery('todos', () => fetch('/todos'), {

    initialData: initialTodos,

  })

}
```

If you configure your query observer with initialData and a staleTime of 1000 ms, the data will be considered fresh for that same amount of time, as if it was just fetched from your query function.

```
function Todos() {
  // Show initialTodos immediately, but won't refetch until another interaction event is encountered after 1000 ms
  const result = useQuery('todos', () => fetch('/todos'), {
```

```
  initialData: initialTodos,
  staleTime: 1000,
 })
}
```

So what if your initialData isn't totally fresh? That leaves us with the last configuration that is actually the most accurate and uses an option called initialDataUpdatedAt. This option allows you to pass a numeric JS timestamp in milliseconds of when the initialData itself was last updated, e.g. what Date.now() provides. Take note that if you have a unix timestamp, you'll need to convert it to a JS timestamp by multiplying it by 1000.

```
function Todos() {
 // Show initialTodos immediately, but won't refetch until another interaction event is encountered after 1000 ms
 const result = useQuery('todos', () => fetch('/todos'), {
  initialData: initialTodos,
  staleTime: 60 * 1000 // 1 minute
  // This could be 10 seconds ago or 10 minutes ago
  initialDataUpdatedAt: initialTodosUpdatedTimestamp // eg. 1608412420052
 })
}
```

This option allows the staleTime to be used for its original purpose, determining how fresh the data needs to be, while also allowing the data to be refetched on mount if the initialData is older than the staleTime. In the example above, our data needs to be fresh within 1 minute, and we can hint to the query when the initialData was last updated so the query can decide for itself whether the data needs to be refetched again or not.

If you would rather treat your data as prefetched data, we recommend that you use the prefetchQuery or fetchQuery APIs to populate the cache beforehand, thus letting you configure your staleTime independently from your initialData

Initial Data Function

If the process for accessing a query's initial data is intensive or just not something you want to perform on every render, you can pass a function as the initialData value. This function will be executed only once when the query is initialized, saving you precious memory and/or CPU:

```
function Todos() {
 const result = useQuery('todos', () => fetch('/todos'), {
  initialData: () => {
   return getExpensiveTodos()
  },
 })
}
```

Initial Data from Cache

In some circumstances, you may be able to provide the initial data for a query from the cached result of another. A good example of this would be searching the cached data from a todos list query for an individual todo item, then using that as the initial data for your individual todo query:

```
function Todo({ todoId }) {
  const result = useQuery(['todo', todoId], () => fetch('/todos'), {
    initialData: () => {
      // Use a todo from the 'todos' query as the initial data for this todo query
      return queryClient.getQueryData('todos')?.find(d => d.id === todoId)
    },
  })
}
```

Initial Data from the cache with initialDataUpdatedAt

Getting initial data from the cache means the source query you're using to look up the initial data from is likely old, but initialData. Instead of using an artificial staleTime to keep your query from refetching immediately, it's suggested that you pass the source query's dataUpdatedAt to initialDataUpdatedAt. This provides the query instance with all the information it needs to determine if and when the query needs to be refetched, regardless of initial data being provided.

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

Conditional Initial Data from Cache

If the source query you're using to look up the initial data from is old, you may not want to use the cached data at all and just fetch from the server. To make this decision easier, you can use the queryClient.getQueryState method instead to get more information about the source query, including a state.dataUpdatedAt timestamp you can use to decide if the query is "fresh" enough for your needs:

```
function Todo({ todoId }) {
  const result = useQuery(['todo', todoId], () => fetch(`/todos/${todoId}`), {
    initialData: () => {
      // Get the query state
      const state = queryClient.getQueryState('todos')

      // If the query exists and has data that is no older than 10 seconds...
      if (state && Date.now() - state.dataUpdatedAt <= 10 * 1000) {
        // return the individual todo
        return state.data.find(d => d.id === todoId)
      }

      // Otherwise, return undefined and let it fetch from a hard loading state!
    },
  })
}
```

Placeholder Query Data

What is placeholder data?

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

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

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

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

Placeholder Data as a Value

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

Placeholder Data as a Function

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

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

Placeholder Data from Cache

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

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

Prefetching

If you're lucky enough, you may know enough about what your users will do to be able to prefetch the data they need before it's needed! If this is the case, you can use the prefetchQuery method to prefetch the results of a query to be placed into the cache:

```
const prefetchTodos = async () => {
  // The results of this query will be cached like a normal query
  await queryClient.prefetchQuery('todos', fetchTodos)
}
```

- If data for this query is already in the cache and not invalidated, the data will not be fetched
- If a staleTime is passed eg. prefetchQuery('todos', fn, { staleTime: 5000 }) and the data is older than the specified staleTime, the query will be fetched

- If no instances of useQuery appear for a prefetched query, it will be deleted and garbage collected after the time specified in cacheTime.

Manually Priming a Query

Alternatively, if you already have the data for your query synchronously available, you don't need to prefetch it. You can just use the Query Client's setQueryData method to directly add or update a query's cached result by key.

```
queryClient.setQueryData('todos', todos)
```

Mutations

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

Here's an example of a mutation that adds a new todo to the server:

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

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

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

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

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

- isIdle or status === 'idle' - The mutation is currently idle or in a fresh/reset state
- isLoading or status === 'loading' - The mutation is currently running
- isError or status === 'error' - The mutation encountered an error
- isSuccess or status === 'success' - The mutation was successful and mutation data is available
- Beyond those primary states, more information is available depending on the state of the mutation:
- error - If the mutation is in an error state, the error is available via the error property.
- data - If the mutation is in a success state, the data is available via the data property.

In the example above, you also saw that you can pass variables to your mutations function by calling the mutate function with a single variable or object.

Even with just variables, mutations aren't all that special, but when used with the onSuccess option, the Query Client's invalidateQueries method and the Query Client's setQueryData method, mutations become a very powerful tool.

IMPORTANT: The mutate function is an asynchronous function, which means you cannot use it directly in an event callback in React 16 and earlier. If you need to access the event in onSubmit you need to wrap mutate in another function. This is due to React event pooling.

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

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

// This will work
const CreateTodo = () => {
  const mutation = useMutation(formData => {
    return fetch('/api', formData)
  })
  const onSubmit = event => {
    event.preventDefault()
    mutation.mutate(new FormData(event.target))
  }

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

Resetting Mutation State

It's sometimes the case that you need to clear the error or data of a mutation request. To do this, you can use the reset function to handle this:

```
const CreateTodo = () => {
```

```
  const [title, setTitle] = useState('')
  const mutation = useMutation(createTodo)

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

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

Mutation Side Effects

useMutation comes with some helper options that allow quick and easy side-effects at any stage during the mutation lifecycle. These come in handy for both invalidating and refetching queries after mutations and even optimistic updates

```
useMutation(addTodo, {
  onMutate: variables => {
    // A mutation is about to happen!

    // Optionally return a context containing data to use when for example rolling back
    return { id: 1 }
  },
  onError: (error, variables, context) => {
    // An error happened!
    console.log(`rolling back optimistic update with id ${context.id}`)
  },
  onSuccess: (data, variables, context) => {
    // Boom baby!
  },
  onSettled: (data, error, variables, context) => {
    // Error or success... doesn't matter!
  },
})
```

When returning a promise in any of the callback functions it will first be awaited before the next callback is called:

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

You might find that you want to trigger additional callbacks than the ones defined on useMutation when calling mutate. This can be used to trigger component-specific side effects. To do that, you can provide any of the same callback options to the mutate function after your mutation variable. Supported overrides include: onSuccess, onError and onSettled. Please keep in mind that those additional callbacks won't run if your component unmounts before the mutation finishes.

```
useMutation(addTodo, {
  onSuccess: (data, variables, context) => {
    // I will fire first
  },
  onError: (error, variables, context) => {
    // I will fire first
  },
  onSettled: (data, error, variables, context) => {
    // I will fire first
  },
})

mutate(todo, {
  onSuccess: (data, variables, context) => {
    // I will fire second!
  },
  onError: (error, variables, context) => {
    // I will fire second!
  },
  onSettled: (data, error, variables, context) => {
    // I will fire second!
  },
})
```

Consecutive mutations

There is a slight difference in handling onSuccess, onError and onSettled callbacks when it comes to consecutive mutations. When passed to the mutate function, they will be fired up only once and only if the component is still mounted. This is due to the fact that mutation observer is removed and resubscribed every time when the mutate function is called. On the contrary, useMutation handlers execute for each mutate call.

Be aware that most likely, mutationFn passed to useMutation is ansynchronous. In that case, the order in which mutations are fulfilled may differ from the order of mutate function calls.

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

['Todo 1', 'Todo 2', 'Todo 3'].forEach((todo) => {
  mutate(todo, {
    onSuccess: (data, error, variables, context) => {
      // Will execute only once, for the last mutation (Todo 3),
      // regardless which mutation resolves first
    },
  })
})
```

Promises

Use mutateAsync instead of mutate to get a promise which will resolve on success or throw on an error. This can for example be used to compose side effects.

```
const mutation = useMutation(addTodo)

try {
  const todo = await mutation.mutateAsync(todo)
  console.log(todo)
} catch (error) {
  console.error(error)
} finally {
  console.log('done')
}
```

Retry

By default React Query will not retry a mutation on error, but it is possible with the retry option:

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

If mutations fail because the device is offline, they will be retried in the same order when the device reconnects.

Persist mutations

Mutations can be persisted to storage if needed and resumed at a later point. This can be done with the hydration functions:

```
const queryClient = new QueryClient()

// Define the "addTodo" mutation
queryClient.setMutationDefaults('addTodo', {
  mutationFn: addTodo,
  onMutate: async (variables) => {
    // Cancel current queries for the todos list
    await queryClient.cancelQueries('todos')

    // Create optimistic todo
    const optimisticTodo = { id: uuid(), title: variables.title }

    // Add optimistic todo to todos list
    queryClient.setQueryData('todos', old => [...old, optimisticTodo])

    // Return context with the optimistic todo
    return { optimisticTodo }
  },
  onSuccess: (result, variables, context) => {
    // Replace optimistic todo in the todos list with the result
    queryClient.setQueryData('todos', old => old.map(todo => todo.id === context.optimisticTodo.id ?
result : todo))
  },
  onError: (error, variables, context) => {
    // Remove optimistic todo from the todos list
    queryClient.setQueryData('todos', old => old.filter(todo => todo.id !== context.optimisticTodo.id))
  },
  retry: 3,
})

// Start mutation in some component:
const mutation = useMutation('addTodo')
mutation.mutate({ title: 'title' })

// If the mutation has been paused because the device is for example offline,
// Then the paused mutation can be dehydrated when the application quits:
const state = dehydrate(queryClient)

// The mutation can then be hydrated again when the application is started:
hydrate(queryClient, state)

// Resume the paused mutations:
queryClient.resumePausedMutations()
```

Query Invalidation

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

```
// Invalidate every query in the cache

queryClient.invalidateQueries()

// Invalidate every query with a key that starts with `todos`

queryClient.invalidateQueries('todos')
```

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

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

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

Query Matching with invalidateQueries

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

In this example, we can use the todos prefix to invalidate any queries that start with todos in their query key:

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

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

queryClient.invalidateQueries('todos')

// Both queries below will be invalidated
const todoListQuery = useQuery('todos', fetchTodoList)
const todoListQuery = useQuery(['todos', { page: 1 }], fetchTodoList)
```

You can even invalidate queries with specific variables by passing a more specific query key to the invalidateQueries method:

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

// The query below will be invalidated
const todoListQuery = useQuery(['todos', { type: 'done' }], fetchTodoList)
```

// However, the following query below will NOT be invalidated

```
const todoListQuery = useQuery('todos', fetchTodoList)
```

The invalidateQueries API is very flexible, so even if you want to only invalidate todos queries that don't have any more variables or subkeys, you can pass an exact: true option to the invalidateQueries method:

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

// The query below will be invalidated
const todoListQuery = useQuery(['todos'], fetchTodoList)

// However, the following query below will NOT be invalidated
const todoListQuery = useQuery(['todos', { type: 'done' }], fetchTodoList)
```

If you find yourself wanting even more granularity, you can pass a predicate function to the invalidateQueries method. This function will receive each Query instance from the query cache and allow you to return true or false for whether you want to invalidate that query:

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

// The query below will be invalidated
const todoListQuery = useQuery(['todos', { version: 20 }], fetchTodoList)

// The query below will be invalidated
const todoListQuery = useQuery(['todos', { version: 10 }], fetchTodoList)

// However, the following query below will NOT be invalidated
const todoListQuery = useQuery(['todos', { version: 5 }], fetchTodoList)
```

Invalidation from Mutations

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

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

js

```
const mutation = useMutation(postTodo)
```

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

js

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


const queryClient = useQueryClient()


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

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

Updates from Mutation Responses

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

```
const queryClient = useQueryClient()

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

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

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

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

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

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

Optimistic Updates

When you optimistically update your state before performing a mutation, there is a chance that the mutation will fail. In most of these failure cases, you can just trigger a refetch for your optimistic queries to revert them to their true server state. In some circumstances though, refetching may not work correctly and the mutation error could represent some type of server issue that won't make it possible to refetch. In this event, you can instead choose to rollback your update.

To do this, useMutation's onMutate handler option allows you to return a value that will later be passed to both onError and onSettled handlers as the last argument. In most cases, it is most useful to pass a rollback function.

Updating a list of todos when adding a new todo

```
const queryClient = useQueryClient()

useMutation(updateTodo, {
  // When mutate is called:
  onMutate: async newTodo => {
    // Cancel any outgoing refetches (so they don't overwrite our optimistic update)
    await queryClient.cancelQueries('todos')
```

```
  // Snapshot the previous value
  const previousTodos = queryClient.getQueryData('todos')

  // Optimistically update to the new value
  queryClient.setQueryData('todos', old => [...old, newTodo])

  // Return a context object with the snapshotted value
  return { previousTodos }
 },
 // If the mutation fails, use the context returned from onMutate to roll back
 onError: (err, newTodo, context) => {
  queryClient.setQueryData('todos', context.previousTodos)
 },
 // Always refetch after error or success:
 onSettled: () => {
  queryClient.invalidateQueries('todos')
 },
})
```

Updating a single todo

```
useMutation(updateTodo, {
 // When mutate is called:
 onMutate: async newTodo => {
  // Cancel any outgoing refetches (so they don't overwrite our optimistic update)
  await queryClient.cancelQueries(['todos', newTodo.id])

  // Snapshot the previous value
  const previousTodo = queryClient.getQueryData(['todos', newTodo.id])

  // Optimistically update to the new value
  queryClient.setQueryData(['todos', newTodo.id], newTodo)

  // Return a context with the previous and new todo
  return { previousTodo, newTodo }
 },
 // If the mutation fails, use the context we returned above
 onError: (err, newTodo, context) => {
  queryClient.setQueryData(
   ['todos', context.newTodo.id],
   context.previousTodo
  )
 },
 // Always refetch after error or success:
 onSettled: newTodo => {
  queryClient.invalidateQueries(['todos', newTodo.id])
 },
})
```

You can also use the onSettled function in place of the separate onError and onSuccess handlers if you wish:

```
js
```

```
useMutation(updateTodo, {
 // ...
 onSettled: (newTodo, error, variables, context) => {
  if (error) {
   // do something
  }
 },
})
```

Query Cancellation

Previous method requiring a cancel function

React Query provides each query function with an AbortSignal instance, if it's available in your runtime environment. When a query becomes out-of-date or inactive, this signal will become aborted. This means that all queries are cancellable, and you can respond to the cancellation inside your query function if desired. The best part about this is that it allows you to continue to use normal async/await syntax while getting all the benefits of automatic cancellation. Additionally, this solution works better with TypeScript than the old solution.

The AbortController API is available in most runtime environments, but if the runtime environment does not support it then the query function will receive undefined in its place. You may choose to polyfill the AbortController API if you wish, there are several available.

NOTE: This feature was introduced at version 3.30.0. If you are using an older version, you will need to either upgrade (recommended) or use the old cancel function.

Default behavior

By default, queries that unmount or become unused before their promises are resolved are not cancelled. This means that after the promise has resolved, the resulting data will be available in the cache. This is helpful if you've started receiving a query, but then unmount the component before it finishes. If you mount the component again and the query has not been garbage collected yet, data will be available.

However, if you consume the AbortSignal or attach a cancel function to your Promise, the Promise will be cancelled (e.g. aborting the fetch) and therefore, also the Query must be cancelled. Cancelling the query will result in its state being reverted to its previous state.

Using fetch

```
const query = useQuery('todos', async ({ signal }) => {
 const todosResponse = await fetch('/todos', {
  // Pass the signal to one fetch
```

```
    signal,
  })
  const todos = await todosResponse.json()

  const todoDetails = todos.map(async ({ details } => {
    const response = await fetch(details, {
      // Or pass it to several
      signal,
    })
    return response.json()
  })

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

Using axios

Using axios v0.22.0+

```
import axios from 'axios'

const query = useQuery('todos', ({ signal }) =>
  axios.get('/todos', {
    // Pass the signal to `axios`
    signal,
  })
)
```

Using an axios version less than v0.22.0

```
import axios from 'axios'

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

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

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

  return promise
```

```
})
```

Using XMLHttpRequest

```
const query = useQuery('todos', ({ signal }) => {
  return new Promise((resolve, reject) => {
    var oReq = new XMLHttpRequest()
    oReq.addEventListener('load', () => {
      resolve(JSON.parse(oReq.responseText))
    })
    signal?.addEventListener('abort', () => {
      oReq.abort()
      reject()
    })
    oReq.open('GET', '/todos')
    oReq.send()
  })
})
```

Using graphql-request

An AbortSignal can be set in the client request method.

```
const client = new GraphQLClient(endpoint)

const query = useQuery('todos', ({ signal }) => {
  client.request({ document: query, signal })
})
```

Using graphql-request version less than v4.0.0

An AbortSignal can be set in the GraphQLClient constructor.

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

Manual Cancellation

You might want to cancel a query manually. For example, if the request takes a long time to finish, you can allow the user to click a cancel button to stop the request. To do this, you just need to

call queryClient.cancelQueries(key), which will cancel the query and revert it back to its previous state. If promise.cancel is available, or you have consumed the signal passed to the query function, React Query will additionally also cancel the Promise.

```
const [queryKey] = useState('todos')

const query = useQuery(queryKey, async ({ signal }) => {
  const resp = await fetch('/todos', { signal })
  return resp.json()
})

const queryClient = useQueryClient()

return (
  <button onClick={(e) => {
    e.preventDefault()
    queryClient.cancelQueries(queryKey)
  }}>Cancel</button>
)
```

Old cancel function

Don't worry! The previous cancellation functionality will continue to work. But we do recommend that you move away from the withdrawn cancelable promise proposal to the new AbortSignal interface which has been standardized as a general purpose construct for aborting ongoing activities in most browsers and in Node. The old cancel function might be removed in a future major version.

To integrate with this feature, attach a cancel function to the promise returned by your query that implements your request cancellation. When a query becomes out-of-date or inactive, this promise.cancel function will be called (if available).

Using axios with cancel function

```
import axios from 'axios'

const query = useQuery('todos', () => {
  // Create a new CancelToken source for this request
  const CancelToken = axios.CancelToken
  const source = CancelToken.source()

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

  // Cancel the request if React Query calls the `promise.cancel` method
  promise.cancel = () => {
    source.cancel('Query was cancelled by React Query')
```

```
  }

  return promise
})
```

Using fetch with cancel function

```
const query = useQuery('todos', () => {
  // Create a new AbortController instance for this request
  const controller = new AbortController()
  // Get the abortController's signal
  const signal = controller.signal

  const promise = fetch('/todos', {
    method: 'get',
    // Pass the signal to your request
    signal,
  })

  // Cancel the request if React Query calls the `promise.cancel` method
  promise.cancel = () => controller.abort()

  return promise
})
```

Scroll Restoration

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

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

Filters

Some methods within React Query accept a QueryFilters or MutationFilters object.

Query Filters

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

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

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

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

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

A query filter object supports the following properties:

- exact?: boolean
  - If you don't want to search queries inclusively by query key, you can pass the exact: true option to return only the query with the exact query key you have passed.
- active?: boolean
  - When set to true it will match active queries.
  - When set to false it will match inactive queries.
- inactive?: boolean
  - When set to true it will match inactive queries.
  - When set to false it will match active queries.
- stale?: boolean
  - When set to true it will match stale queries.
  - When set to false it will match fresh queries.
- fetching?: boolean
  - When set to true it will match queries that are currently fetching.
  - When set to false it will match queries that are not fetching.
- predicate?: (query: Query) => boolean
  - This predicate function will be called for every single query in the cache and be expected to return truthy for queries that are found.
- queryKey?: QueryKey
  - Set this property to define a query key to match on.

Mutation Filters

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

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

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

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

A mutation filter object supports the following properties:

- exact?: boolean
  - If you don't want to search mutations inclusively by mutation key, you can pass the exact: true option to return only the mutation with the exact mutation key you have passed.
- fetching?: boolean
  - When set to true it will match mutations that are currently fetching.
  - When set to false it will match mutations that are not fetching.
- predicate?: (mutation: Mutation) => boolean
  - This predicate function will be called for every single mutation in the cache and be expected to return truthy for mutations that are found.
- mutationKey?: MutationKey
  - Set this property to define a mutation key to match on.

SSR

React Query supports two ways of prefetching data on the server and passing that to the queryClient.

- Prefetch the data yourself and pass it in as initialData
  - Quick to set up for simple cases
  - Has some caveats
- Prefetch the query on the server, dehydrate the cache and rehydrate it on the client
  - Requires slightly more setup up front

Using Next.js

The exact implementation of these mechanisms may vary from platform to platform, but we recommend starting with Next.js which supports 2 forms of pre-rendering:

- Static Generation (SSG)
- Server-side Rendering (SSR)

React Query supports both of these forms of pre-rendering regardless of what platform you may be using

Using initialData

Together with Next.js's getStaticProps or getServerSideProps, you can pass the data you fetch in either method to useQuery's' initialData option. From React Query's perspective, these integrate in the same way, getStaticProps is shown below:

```
export async function getStaticProps() {
  const posts = await getPosts()
  return { props: { posts } }
}

function Posts(props) {
  const { data } = useQuery('posts', getPosts, { initialData: props.posts })

  // ...
}
```

The setup is minimal and this can be a quick solution for some cases, but there are a few tradeoffs to consider when compared to the full approach:

- If you are calling useQuery in a component deeper down in the tree you need to pass the initialData down to that point
- If you are calling useQuery with the same query in multiple locations, you need to pass initialData to all of them
- There is no way to know at what time the query was fetched on the server, so dataUpdatedAt and determining if the query needs refetching is based on when the page loaded instead

Using Hydration

React Query supports prefetching multiple queries on the server in Next.js and then dehydrating those queries to the queryClient. This means the server can prerender markup that is immediately available on page load and as soon as JS is available, React Query can upgrade or hydrate those queries with the full functionality of the library. This includes refetching those queries on the client if they have become stale since the time they were rendered on the server.

To support caching queries on the server and set up hydration:

- Create a new QueryClient instance inside of your app, and on an instance ref (or in React state). This ensures that data is not shared between different users and requests, while still only creating the QueryClient once per component lifecycle.
- Wrap your app component with <QueryClientProvider> and pass it the client instance
- Wrap your app component with <Hydrate> and pass it the dehydratedState prop from pageProps

```
// _app.jsx
import { Hydrate, QueryClient, QueryClientProvider } from 'react-query'

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

  return (
    <QueryClientProvider client={queryClient}>
      <Hydrate state={pageProps.dehydratedState}>
```

```
      <Component {...pageProps} />
    </Hydrate>
  </QueryClientProvider>
 )
}
```

Now you are ready to prefetch some data in your pages with either getStaticProps (for SSG) or getServerSideProps (for SSR). From React Query's perspective, these integrate in the same way, getStaticProps is shown below.

- Create a new QueryClient instance for each page request. This ensures that data is not shared between users and requests.
- Prefetch the data using the clients prefetchQuery method and wait for it to complete
- Use dehydrate to dehydrate the query cache and pass it to the page via the dehydratedState prop. This is the same prop that the cache will be picked up from in your _app.js

```
// pages/posts.jsx
import { dehydrate, QueryClient, useQuery } from 'react-query';

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

 await queryClient.prefetchQuery('posts', getPosts)

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

function Posts() {
 // This useQuery could just as well happen in some deeper child to
 // the "Posts"-page, data will be available immediately either way
 const { data } = useQuery('posts', getPosts)

 // This query was not prefetched on the server and will not start
 // fetching until on the client, both patterns are fine to mix
 const { data: otherData } = useQuery('posts-2', getPosts)

 // ...
}
```

As demonstrated, it's fine to prefetch some queries and let others fetch on the queryClient. This means you can control what content server renders or not by adding or removing prefetchQuery for a specific query.

Caveat for Next.js rewrites

There's a catch if you're using Next.js' rewrites feature together with Automatic Static Optimization or getStaticProps: It will cause a second hydration by React Query. That's because Next.js needs to ensure that they parse the rewrites on the client and collect any params after hydration so that they can be provided in router.query.

The result is missing referential equality for all the hydration data, which for example triggers whereever your data is used as props of components or in the dependency array of useEffects/useMemos.

Using Other Frameworks or Custom SSR Frameworks

This guide is at-best, a high level overview of how SSR with React Query should work. Your mileage may vary since there are many different possible setups for SSR.

> If you can, please contribute your findings back to this page for any framework specific guidance!

On the Server

- Create a new QueryClient instance inside of your request handler. This ensures that data is not shared between different users and requests.
- Using the client, prefetch any data you need
- Dehydrate the client
- Render your app with the client provider and also using the dehydrated state. This is extremely important! You must render both server and client using the same dehydrated state to ensure hydration on the client produces the exact same markup as the server.
- Serialize and embed the dehydrated cache to be sent to the client with the HTML
- Clear the React Query caches after the dehydrated state has been sent by calling queryClient.clear()

> SECURITY NOTE: Serializing data with JSON.stringify can put you at risk for XSS-vulnerabilities, this blog post explains why and how to solve it

```
import { dehydrate, Hydrate, QueryClient, QueryClientProvider } from 'react-query';

function handleRequest (req, res) {
  const queryClient = new QueryClient()
  await queryClient.prefetchQuery('key', fn)
  const dehydratedState = dehydrate(queryClient)
```

```
  const html = ReactDOM.renderToString(
    <QueryClientProvider client={queryClient}>
      <Hydrate state={dehydratedState}>
        <App />
      </Hydrate>
    </QueryClientProvider>
  )

  res.send(`
    <html>
      <body>
        <div id="root">${html}</div>
        <script>
          window.__REACT_QUERY_STATE__ = ${JSON.stringify(dehydratedState)};
        </script>
      </body>
    </html>
  `)

  queryClient.clear()
}
```

[Client]

Parse the dehydrated cache state that was sent to the client with the HTML

Create a new QueryClient instance

Render your app with the client provider and also using the dehydrated state. This is extremely important! You must render both server and client using the same dehydrated state to ensure hydration on the client produces the exact same markup as the server.

```
import { Hydrate, QueryClient, QueryClientProvider } from 'react-query'

const dehydratedState = window.__REACT_QUERY_STATE__

const queryClient = new QueryClient()

ReactDOM.hydrate(
  <QueryClientProvider client={queryClient}>
    <Hydrate state={dehydratedState}>
      <App />
    </Hydrate>
  </QueryClientProvider>,
  document.getElementById('root')
)
```

[Tips, Tricks and Caveats]

Only successful queries are included in dehydration

Any query with an error is automatically excluded from dehydration. This means that the default behavior is to pretend these queries were never loaded on the server, usually showing a loading state instead, and retrying the queries on the queryClient. This happens regardless of error.

Sometimes this behavior is not desirable, maybe you want to render an error page with a correct status code instead on certain errors or queries. In those cases, use fetchQuery and catch any errors to handle those manually.

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

A query is considered stale depending on when it was dataUpdatedAt. A caveat here is that the server needs to have the correct time for this to work properly, but UTC time is used, so timezones do not factor into this.

Because staleTime defaults to 0, queries will be refetched in the background on page load by default. You might want to use a higher staleTime to avoid this double fetching, especially if you don't cache your markup.

This refetching of stale queries is a perfect match when caching markup in a CDN! You can set the cache time of the page itself decently high to avoid having to re-render pages on the server, but configure the staleTime of the queries lower to make sure data is refetched in the background as soon as a user visits the page. Maybe you want to cache the pages for a week, but refetch the data automatically on page load if it's older than a day?

High memory consumption on server

In case you are creating the QueryClient for every request, React Query creates the isolated cache for this client, which is preserved in memory for the cacheTime period (which defaults to 5 minutes). That may lead to high memory consumption on server in case of high number of requests during that period.

To clear the cache after it is not needed and to lower memory consumption, you can add a call to queryClient.clear() after the request is handled and dehydrated state has been sent to the client.

Alternatively, you can set a smaller cacheTime.

Caching Examples

Please thoroughly read the Important Defaults before reading this guide

Basic Example

This caching example illustrates the story and lifecycle of:

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

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

- A new instance of useQuery('todos', fetchTodos) mounts.
    - Since no other queries have been made with this query + variable combination, this query will show a hard loading state and make a network request to fetch the data.
    - It will then cache the data using 'todos' and fetchTodos as the unique identifiers for that cache.
    - The hook will mark itself as stale after the configured staleTime (defaults to 0, or immediately).
- A second instance of useQuery('todos', fetchTodos) mounts elsewhere.
    - Because this exact data exists in the cache from the first instance of this query, that data is immediately returned from the cache.
- A background refetch is triggered for both queries (but only one request), since a new instance appeared on screen.
    - Both instances are updated with the new data if the fetch is successful
- Both instances of the useQuery('todos', fetchTodos) query are unmounted and no longer in use.
    - Since there are no more active instances of this query, a cache timeout is set using cacheTime to delete and garbage collect the query (defaults to 5 minutes).
- Before the cache timeout has completed another instance of useQuery('todos', fetchTodos) mounts. The query immediately returns the available cached value while the fetchTodos function is being run in the background to populate the query with a fresh value.
- The final instance of useQuery('todos', fetchTodos) unmounts.
- No more instances of useQuery('todos', fetchTodos) appear within 5 minutes.
    - This query and its data are deleted and garbage collected.

Default Query Function

If you find yourself wishing for whatever reason that you could just share the same query function for your entire app and just use query keys to identify what it should fetch, you can do that by providing a default query function to React Query:

```
// Define a default query function that will receive the query key
// the queryKey is guaranteed to be an Array here
const defaultQueryFn = async ({ queryKey }) => {
  const { data } = await axios.get(`https://jsonplaceholder.typicode.com${queryKey[0]}`);
  return data;
};

// provide the default query function to your app with defaultOptions
```

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      queryFn: defaultQueryFn,
    },
  },
})

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

// All you have to do now is pass a key!
function Posts() {
  const { status, data, error, isFetching } = useQuery('/posts')

  // ...
}

// You can even leave out the queryFn and just go straight into options
function Post({ postId }) {
  const { status, data, error, isFetching } = useQuery(`/posts/${postId}`, {
    enabled: !!postId,
  })

  // ...
}
```

If you ever want to override the default queryFn, you can just provide your own like you normally would.

Suspense

NOTE: Suspense mode for React Query is experimental, same as Suspense for data fetching itself. These APIs WILL change and should not be used in production unless you lock both your React and React Query versions to patch-level versions that are compatible with each other.

React Query can also be used with React's new Suspense for Data Fetching API's. To enable this mode, you can set either the global or query level config's suspense option to true.

Global configuration:

```
// Configure for all queries
import { QueryClient, QueryClientProvider } from 'react-query'

const queryClient = new QueryClient({
```

```
  defaultOptions: {
    queries: {
      suspense: true,
    },
  },
})

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

Query configuration:

```
import { useQuery } from 'react-query'

// Enable for an individual query
useQuery(queryKey, queryFn, { suspense: true })
```

When using suspense mode, status states and error objects are not needed and are then replaced by usage of the React.Suspense component (including the use of the fallback prop and React error boundaries for catching errors). Please read the Resetting Error Boundaries and look at the Suspense Example for more information on how to set up suspense mode.

In addition to queries behaving differently in suspense mode, mutations also behave a bit differently. By default, instead of supplying the error variable when a mutation fails, it will be thrown during the next render of the component it's used in and propagate to the nearest error boundary, similar to query errors. If you wish to disable this, you can set the useErrorBoundary option to false. If you wish that errors are not thrown at all, you can set the throwOnError option to false as well!

Resetting Error Boundaries

Whether you are using suspense or useErrorBoundaries in your queries, you will need a way to let queries know that you want to try again when re-rendering after some error occurred.

Query errors can be reset with the QueryErrorResetBoundary component or with the useQueryErrorResetBoundary hook.

When using the component it will reset any query errors within the boundaries of the component:

```
import { QueryErrorResetBoundary } from 'react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App: React.FC = () => (
```

```
  <QueryErrorResetBoundary>
    {({ reset }) => (
      <ErrorBoundary
        onReset={reset}
        fallbackRender={({ resetErrorBoundary }) => (
          <div>
            There was an error!
            <Button onClick={() => resetErrorBoundary()}>Try again</Button>
          </div>
        )}
      >
        <Page />
      </ErrorBoundary>
    )}
  </QueryErrorResetBoundary>
)
```

When using the hook it will reset any query errors within the closest QueryErrorResetBoundary. If there is no boundary defined it will reset them globally:

```
import { useQueryErrorResetBoundary } from 'react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App: React.FC = () => {
  const { reset } = useQueryErrorResetBoundary()
  return (
    <ErrorBoundary
      onReset={reset}
      fallbackRender={({ resetErrorBoundary }) => (
        <div>
          There was an error!
          <Button onClick={() => resetErrorBoundary()}>Try again</Button>
        </div>
      )}
    >
      <Page />
    </ErrorBoundary>
  )
}
```

Fetch-on-render vs Render-as-you-fetch

Out of the box, React Query in suspense mode works really well as a Fetch-on-render solution with no additional configuration. This means that when your components attempt to mount, they will trigger query fetching and suspend, but only once you have imported them and mounted them. If you want to take it to the next level and implement a Render-as-you-fetch model, we recommend implementing Prefetching on routing callbacks and/or user interactions events to start loading queries before they are mounted and hopefully even before you start importing or mounting their parent components.

Testing

React Query works by means of hooks - either the ones we offer or custom ones that wrap around them.

Writing unit tests for these custom hooks can be done by means of the React Hooks Testing Library library.

Install this by running:

bash

```
npm install @testing-library/react-hooks react-test-renderer --save-dev
```

(The react-test-renderer library is needed as a peer dependency of @testing-library/react-hooks, and needs to correspond to the version of React that you are using.)

Our First Test

Once installed, a simple test can be written. Given the following custom hook:

bash

```
export function useCustomHook() {
  return useQuery('customHook', () => 'Hello');
}
```

We can write a test for this as follows:

bash

```
const queryClient = new QueryClient();
const wrapper = ({ children }) => (
  <QueryClientProvider client={queryClient}>
    {children}
  </QueryClientProvider>
);

const { result, waitFor } = renderHook(() => useCustomHook(), { wrapper });

await waitFor(() => result.current.isSuccess);

expect(result.current.data).toEqual("Hello");
```

Note that we provide a custom wrapper that builds the QueryClient and QueryClientProvider. This helps to ensure that our test is completely isolated from any other tests.

It is possible to write this wrapper only once, but if so we need to ensure that the QueryClient gets cleared before every test, and that tests don't run in parallel otherwise one test will influence the results of others.

Turn off retries

The library defaults to three retries with exponential backoff, which means that your tests are likely to timeout if you want to test an erroneous query. The easiest way to turn retries off is via the QueryClientProvider. Let's extend the above example:

bash

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      // ✅ turns retries off
      retry: false,
    },
  },
})
const wrapper = ({ children }) => (
  <QueryClientProvider client={queryClient}>
    {children}
  </QueryClientProvider>
);
```

This will set the defaults for all queries in the component tree to "no retries". It is important to know that this will only work if your actual useQuery has no explicit retries set. If you have a query that wants 5 retries, this will still take precedence, because defaults are only taken as a fallback.

Turn off network error logging

When testing we want to suppress network errors being logged to the console. To do this, we can use react-query's setLogger() function.

ts

```
// as part of your test setup

import { setLogger } from 'react-query'


setLogger({

  log: console.log,
```

```
  warn: console.warn,
  // ✅ no more errors on the console
  error: () => {},
})
```

Set cacheTime to Infinity with Jest

cacheTime is set to 5 minutes by default. It means that the cache garbage collector timer will be triggered every 5 minutes. If you use Jest, you can set the cacheTime to Infinity to prevent "Jest did not exit one second after the test run completed" error message.

Testing Network Calls

The primary use for React Query is to cache network requests, so it's important that we can test our code is making the correct network requests in the first place.

There are plenty of ways that these can be tested, but for this example we are going to use nock.

Given the following custom hook:

bash

```
function useFetchData() {
  return useQuery('fetchData', () => request('/api/data'));
}
We can write a test for this as follows:
bash
const queryClient = new QueryClient();
const wrapper = ({ children }) => (
  <QueryClientProvider client={queryClient}>
    {children}
  </QueryClientProvider>
);

const expectation = nock('http://example.com')
  .get('/api/data')
  .reply(200, {
    answer: 42
  });

const { result, waitFor } = renderHook(() => useFetchData(), { wrapper });
```

```
await waitFor(() => {
  return result.current.isSuccess;
});

expect(result.current.data).toEqual({answer: 42});
```

Here we are making use of waitFor and waiting until the query status indicates that the request has succeeded. This way we know that our hook has finished and should have the correct data.

Testing Load More / Infinite Scroll

First we need to mock our API response

bash

```
function generateMockedResponse(page) {
  return {
    page: page,
    items: [...]
  }
}
```

Then, our nock configuration needs to differentiate responses based on the page, and we'll be using uri to do this. uri's value here will be something like "/?page=1 or /?page=2

bash

```
const expectation = nock('http://example.com')
  .persist()
  .query(true)
  .get('/api/data')
  .reply(200, (uri) => {
    const url = new URL(`http://example.com${uri}`);
    const { page } = Object.fromEntries(url.searchParams);
    return generateMockedResponse(page);
  });
```

(Notice the .persist(), because we'll be calling from this endpoint multiple times)

Now we can safely run our tests, the trick here is to await both isFetching and then !isFetching after calling fetchNextPage():

bash

```
const { result, waitFor } = renderHook(() => useInfiniteQueryCustomHook(), { wrapper });

await waitFor(() => result.current.isSuccess);

expect(result.current.data.pages).toStrictEqual(generateMockedResponse(1));
```

```
result.current.fetchNextPage();

await waitFor(() => result.current.isFetching);
await waitFor(() => !result.current.isFetching);

expect(result.current.data.pages).toStrictEqual([
  ...generateMockedResponse(1),
  ...generateMockedResponse(2),
]);

expectation.done();
```

Does React Query replace Redux, MobX or other global state managers?

Well, let's start with a few important items:

- React Query is a server-state library, responsible for managing asynchronous operations between your server and client
- Redux, MobX, Zustand, etc. are client-state libraries that can be used to store asynchronous data, albeit inefficiently when compared to a tool like React Query

With those points in mind, the short answer is that React Query replaces the boilerplate code and related wiring used to manage cache data in your client-state and replaces it with just a few lines of code.

For a vast majority of applications, the truly globally accessible client state that is left over after migrating all of your async code to React Query is usually very tiny.

There are still some circumstances where an application might indeed have a massive amount of synchronous client-only state (like a visual designer or music production application), in which case, you will probably still want a client state manager. In this situation it's important to note that React Query is not a replacement for local/client state management. However, you can use React Query along side most client state managers with zero issues.

A Contrived Example

Here we have some "global" state being managed by a global state library:

js

```
const globalState = {
  projects,
  teams,
  tasks,
  users,
  themeMode,
  sidebarStatus,
```

```
}
```

Currently, the global state manager is caching 4 types of server-state: projects, teams, tasks, and users. If we were to move these server-state assets to React Query, our remaining global state would look more like this:

js

```
const globalState = {
  themeMode,
  sidebarStatus,
}
```

This also means that with a few hook calls to useQuery and useMutation, we also get to remove any boilerplate code that was used to manage our server state eg.

- Connectors
- Action Creators
- Middlewares
- Reducers
- Loading/Error/Result states
- Contexts

With all of those things removed, you may ask yourself, "Is it worth it to keep using our client state manager for this tiny global state?"

And that's up to you!

But React Query's role is clear. It removes asynchronous wiring and boilerplate from your application and replaces it with just a few lines of code.

What are you waiting for, give it a go already!

Migrating to React Query 3

Previous versions of React Query were awesome and brought some amazing new features, more magic, and an overall better experience to the library. They also brought on massive adoption and likewise a lot of refining fire (issues/contributions) to the library and brought to light a few things that needed more polish to make the library even better. v3 contains that very polish.

Overview

- More scalable and testable cache configuration
- Better SSR support
- Data-lag (previously usePaginatedQuery) anywhere!
- Bi-directional Infinite Queries
- Query data selectors!
- Fully configure defaults for queries and/or mutations before use
- More granularity for optional rendering optimization
- New useQueries hook! (Variable-length parallel query execution)
- Query filter support for the useIsFetching() hook!
- Retry/offline/replay support for mutations
- Observe queries/mutations outside of React
- Use the React Query core logic anywhere you want!
- Bundled/Colocated Devtools via react-query/devtools
- Cache Persistence to web storage (experimental via react-query/persistQueryClient-experimental and react-query/createWebStoragePersistor-experimental)

Breaking Changes

The QueryCache has been split into a QueryClient and lower-level QueryCache and MutationCache instances.

The QueryCache contains all queries, the MutationCache contains all mutations, and the QueryClient can be used to set configuration and to interact with them.

This has some benefits:

- Allows for different types of caches.
- Multiple clients with different configurations can use the same cache.
- Clients can be used to track queries, which can be used for shared caches on SSR.
- The client API is more focused towards general usage.
- Easier to test the individual components.

When creating a new QueryClient(), a QueryCache and MutationCache are automatically created for you if you don't supply them.

js

```
import { QueryClient } from 'react-query'

const queryClient = new QueryClient()
```

ReactQueryConfigProvider and ReactQueryCacheProvider have both been replaced by QueryClientProvider

Default options for queries and mutations can now be specified in QueryClient:

Notice that it's now defaultOptions instead of defaultConfig

js

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      // query options
    },
    mutations: {
      // mutation options
    },
  },
})
```

The QueryClientProvider component is now used to connect a QueryClient to your application:

js

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

const queryClient = new QueryClient()

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

The default QueryCache is gone. For real this time!

As previously noted with a deprecation, there is no longer a default QueryCache that is created or exported from the main package. You must create your own via new QueryClient() or new QueryCache() (which you can then pass to new QueryClient({ queryCache }))

The deprecated makeQueryCache utility has been removed.

It's been a long time coming, but it's finally gone :)

QueryCache.prefetchQuery() has been moved to QueryClient.prefetchQuery()

The new QueryClient.prefetchQuery() function is async, but does not return the data from the query. If you require the data, use the new QueryClient.fetchQuery() function

js

```
// Prefetch a query:
await queryClient.prefetchQuery('posts', fetchPosts)

// Fetch a query:
try {
  const data = await queryClient.fetchQuery('posts', fetchPosts)
} catch (error) {
  // Error handling
}
```

ReactQueryErrorResetBoundary and QueryCache.resetErrorBoundaries() have been replaced by QueryErrorResetBoundary and useQueryErrorResetBoundary().

Together, these provide the same experience as before, but with added control to choose which component trees you want to reset. For more information, see:

- QueryErrorResetBoundary
- useQueryErrorResetBoundary

QueryCache.getQuery() has been replaced by QueryCache.find().

QueryCache.find() should now be used to look up individual queries from a cache

QueryCache.getQueries() has been moved to QueryCache.findAll().

QueryCache.findAll() should now be used to look up multiple queries from a cache

QueryCache.isFetching has been moved to QueryClient.isFetching().

Notice that it's now a function instead of a property

The useQueryCache hook has been replaced by the useQueryClient hook.

It returns the provided queryClient for its component tree and shouldn't need much tweaking beyond a rename.

Query key parts/pieces are no longer automatically spread to the query function.

Inline functions are now the suggested way of passing parameters to your query functions:

js

```
// Old
useQuery(['post', id], (_key, id) => fetchPost(id))

// New
useQuery(['post', id], () => fetchPost(id))
```

If you still insist on not using inline functions, you can use the newly passed QueryFunctionContext:

js

```
useQuery(['post', id], context => fetchPost(context.queryKey[1]))
```

Infinite Query Page params are now passed via QueryFunctionContext.pageParam

They were previously added as the last query key parameter in your query function, but this proved to be difficult for some patterns

js

```
// Old

useInfiniteQuery(['posts'], (_key, pageParam = 0) => fetchPosts(pageParam))


// New

useInfiniteQuery(['posts'], ({ pageParam = 0 }) => fetchPosts(pageParam))
```

usePaginatedQuery() has been deprecated in favor of the keepPreviousData option

The new keepPreviousData options is available for both useQuery and useInfiniteQuery and will have the same "lagging" effect on your data:

Js

```
import { useQuery } from 'react-query'

function Page({ page }) {
  const { data } = useQuery(['page', page], fetchPage, {
    keepPreviousData: true,
  })
}
```

useInfiniteQuery() is now bi-directional

The useInfiniteQuery() interface has changed to fully support bi-directional infinite lists.

- options.getFetchMore has been renamed to options.getNextPageParam
- queryResult.canFetchMore has been renamed to queryResult.hasNextPage
- queryResult.fetchMore has been renamed to queryResult.fetchNextPage
- queryResult.isFetchingMore has been renamed to queryResult.isFetchingNextPage
- Added the options.getPreviousPageParam option
- Added the queryResult.hasPreviousPage property
- Added the queryResult.fetchPreviousPage property
- Added the queryResult.isFetchingPreviousPage
- The data of an infinite query is now an object containing the pages and the pageParams used to fetch the pages: { pages: [data, data, data], pageParams: [...]}

One direction:

js

```
const {
  data,
  fetchNextPage,
  hasNextPage,
  isFetchingNextPage,
} = useInfiniteQuery(
  'projects',
  ({ pageParam = 0 }) => fetchProjects(pageParam),
  {
    getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
  }
)
```

Both directions:

js

```
const {
  data,
  fetchNextPage,
  fetchPreviousPage,
  hasNextPage,
  hasPreviousPage,
  isFetchingNextPage,
  isFetchingPreviousPage,
} = useInfiniteQuery(
  'projects',
  ({ pageParam = 0 }) => fetchProjects(pageParam),
  {
    getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
    getPreviousPageParam: (firstPage, pages) => firstPage.prevCursor,
  }
)
```

One direction reversed:

js

```
const {
  data,
  fetchNextPage,
  hasNextPage,
  isFetchingNextPage,
} = useInfiniteQuery(
  'projects',
  ({ pageParam = 0 }) => fetchProjects(pageParam),
  {
    select: data => ({
      pages: [...data.pages].reverse(),
      pageParams: [...data.pageParams].reverse(),
    }),
    getNextPageParam: (lastPage, pages) => lastPage.nextCursor,
  }
)
```

Infinite Query data now contains the array of pages and pageParams used to fetch those pages.

This allows for easier manipulation of the data and the page params, like, for example, removing the first page of data along with it's params:

js

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

useMutation now returns an object instead of an array

Though the old way gave us warm fuzzy feelings of when we first discovered useState for the first time, they didn't last long. Now the mutation return is a single object.

Js

```
// Old:
const [mutate, { status, reset }] = useMutation()

// New:
const { mutate, status, reset } = useMutation()
```

mutation.mutate no longer return a promise

- The [mutate] variable has been changed to the mutation.mutate function
- Added the mutation.mutateAsync function

We got a lot of questions regarding this behavior as users expected the promise to behave like a regular promise.

Because of this the mutate function is now split into a mutate and mutateAsync function.

The mutate function can be used when using callbacks:

js

```
const { mutate } = useMutation(addTodo)

mutate('todo', {
  onSuccess: data => {
    console.log(data)
  },
  onError: error => {
    console.error(error)
  },
  onSettled: () => {
    console.log('settled')
  },
})
```

The mutateAsync function can be used when using async/await:

js

```
const { mutateAsync } = useMutation(addTodo)

try {
  const data = await mutateAsync('todo')
  console.log(data)
} catch (error) {
  console.error(error)
} finally {
  console.log('settled')
}
```

The object syntax for useQuery now uses a collapsed config:

```
// Old:
useQuery({
  queryKey: 'posts',
  queryFn: fetchPosts,
  config: { staleTime: Infinity },
})

// New:
```

```
useQuery({
  queryKey: 'posts',
  queryFn: fetchPosts,
  staleTime: Infinity,
})
```

If set, the QueryOptions.enabled option must be a boolean (true/false)

The enabled query option will now only disable a query when the value is false. If needed, values can be casted with !!userId or Boolean(userId) and a handy error will be thrown if a non-boolean value is passed.

The QueryOptions.initialStale option has been removed

The initialStale query option has been removed and initial data is now treated as regular data. Which means that if initialData is provided, the query will refetch on mount by default. If you do not want to refetch immediately, you can define a staleTime.

The QueryOptions.forceFetchOnMount option has been replaced by refetchOnMount: 'always'

Honestly, we were accruing way too many refetchOn____ options, so this should clean things up.

The QueryOptions.refetchOnMount options now only applies to its parent component instead of all query observers

When refetchOnMount was set to false any additional components were prevented from refetching on mount. In version 3 only the component where the option has been set will not refetch on mount.

The QueryOptions.queryFnParamsFilter has been removed in favor of the new QueryFunctionContext object.

The queryFnParamsFilter option has been removed because query functions now get a QueryFunctionContext object instead of the query key.

Parameters can still be filtered within the query function itself as the QueryFunctionContext also contains the query key.

The QueryOptions.notifyOnStatusChange option has been superseded by the new notifyOnChangeProps and notifyOnChangePropsExclusions options.

With these new options it is possible to configure when a component should re-render on a granular level.

Only re-render when the data or error properties change:

js

```
import { useQuery } from 'react-query'

function User() {
  const { data } = useQuery('user', fetchUser, {
    notifyOnChangeProps: ['data', 'error'],
  })
  return <div>Username: {data.username}</div>
}
```

Prevent re-render when the isStale property changes:

js

```
import { useQuery } from 'react-query'

function User() {
  const { data } = useQuery('user', fetchUser, {
    notifyOnChangePropsExclusions: ['isStale'],
  })
  return <div>Username: {data.username}</div>
}
```

The QueryResult.clear() function has been renamed to QueryResult.remove()

Although it was called clear, it really just removed the query from the cache. The name now matches the functionality.

The QueryResult.updatedAt property has been split into QueryResult.dataUpdatedAt and QueryResult.errorUpdatedAt properties

Because data and errors can be present at the same time, the updatedAt property has been split into dataUpdatedAt and errorUpdatedAt.

setConsole() has been replaced by the new setLogger() function

js

```
import { setLogger } from 'react-query'

// Log with Sentry
```

```
setLogger({
  error: error => {
    Sentry.captureException(error)
  },
})

// Log with Winston
setLogger(winston.createLogger())
```

React Native no longer requires overriding the logger

To prevent showing error screens in React Native when a query fails it was necessary to manually change the Console:

js

```
import { setConsole } from 'react-query'

setConsole({
  log: console.log,
  warn: console.warn,
  error: console.warn,
})
```

In version 3 this is done automatically when React Query is used in React Native.

Typescript

QueryStatus has been changed from an enum to a union type

So, if you were checking the status property of a query or mutation against a QueryStatus enum property you will have to check it now against the string literal the enum previously held for each property.

Therefore you have to change the enum properties to their equivalent string literal, like this:

- QueryStatus.Idle -> 'idle'
- QueryStatus.Loading -> 'loading'
- QueryStatus.Error -> 'error'
- QueryStatus.Success -> 'success'
-

Here is an example of the changes you would have to make:

diff

```
- import { useQuery, QueryStatus } from 'react-query';
+ import { useQuery } from 'react-query';
const { data, status } = useQuery(['post', id], () => fetchPost(id))

- if (status === QueryStatus.Loading) {
+ if (status === 'loading') {
```

```
  ...
}

- if (status === QueryStatus.Error) {
+ if (status === 'error') {
  ...
}
```

New features

Query Data Selectors

The useQuery and useInfiniteQuery hooks now have a select option to select or transform parts of the query result.

js

```
import { useQuery } from 'react-query'

function User() {
  const { data } = useQuery('user', fetchUser, {
    select: user => user.username,
  })
  return <div>Username: {data}</div>
}
```

Set the notifyOnChangeProps option to ['data', 'error'] to only re-render when the selected data changes.

The useQueries() hook, for variable-length parallel query execution

Wish you could run useQuery in a loop? The rules of hooks say no, but with the new useQueries() hook, you can!

Js

```
import { useQueries } from 'react-query'

function Overview() {
  const results = useQueries([
    { queryKey: ['post', 1], queryFn: fetchPost },
    { queryKey: ['post', 2], queryFn: fetchPost },
  ])
  return (
```

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

Retry/offline mutations

By default React Query will not retry a mutation on error, but it is possible with the retry option:

js

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

If mutations fail because the device is offline, they will be retried in the same order when the device reconnects.

Persist mutations

Mutations can now be persisted to storage and resumed at a later point. More information can be found in the mutations documentation.

QueryObserver

A QueryObserver can be used to create and/or watch a query:

js

```
const observer = new QueryObserver(queryClient, { queryKey: 'posts' })

const unsubscribe = observer.subscribe(result => {
  console.log(result)
  unsubscribe()
})
```

InfiniteQueryObserver

A InfiniteQueryObserver can be used to create and/or watch an infinite query:

js

```
const observer = new InfiniteQueryObserver(queryClient, {
  queryKey: 'posts',
  queryFn: fetchPosts,
```

```
  getNextPageParam: (lastPage, allPages) => lastPage.nextCursor,
  getPreviousPageParam: (firstPage, allPages) => firstPage.prevCursor,
})

const unsubscribe = observer.subscribe(result => {
  console.log(result)
  unsubscribe()
})
```

QueriesObserver

A QueriesObserver can be used to create and/or watch multiple queries:

js

```
const observer = new QueriesObserver(queryClient, [
  { queryKey: ['post', 1], queryFn: fetchPost },
  { queryKey: ['post', 2], queryFn: fetchPost },
])

const unsubscribe = observer.subscribe(result => {
  console.log(result)
  unsubscribe()
})
```

Set default options for specific queries

The QueryClient.setQueryDefaults() method can be used to set default options for specific queries:

js

```
queryClient.setQueryDefaults('posts', { queryFn: fetchPosts })

function Component() {
  const { data } = useQuery('posts')
}
```

Set default options for specific mutations

The QueryClient.setMutationDefaults() method can be used to set default options for specific mutations:

js

```
queryClient.setMutationDefaults('addPost', { mutationFn: addPost })

function Component() {
  const { mutate } = useMutation('addPost')
}
```

useIsFetching()

The useIsFetching() hook now accepts filters which can be used to for example only show a spinner for certain type of queries:

js

```
const fetches = useIsFetching(['posts'])
```

Core separation

The core of React Query is now fully separated from React, which means it can also be used standalone or in other frameworks. Use the react-query/core entry point to only import the core functionality:

js

```
import { QueryClient } from 'react-query/core'
```

Devtools are now part of the main repo and npm package

The devtools are now included in the react-query package itself under the import react-query/devtools. Simply replace react-query-devtools imports with react-query/devtools

PLUGINS (experimental)

persistQueryClient (Experimental)

VERY IMPORTANT: This utility is currently in an experimental stage. This means that breaking changes will happen in minor AND patch releases. Use at your own risk. If you choose to rely on this in

production in an experimental stage, please lock your version to a patch-level version to avoid unexpected breakages.

persistQueryClient is a utility for persisting the state of your queryClient and its caches for later use. Different persistors can be used to store your client and cache to many different storage layers.

Officially Supported Persistors

- createWebStoragePersistor (Experimental)
- createAsyncStoragePersistor (Experimental)

Installation

This utility comes packaged with react-query and is available under the react-query/persistQueryClient-experimental import.

Usage

Import the persistQueryClient function, and pass it your QueryClient instance (with a cacheTime set), and a Persistor interface (there are multiple persistor types you can use):

ts

```
import { persistQueryClient } from 'react-query/persistQueryClient-experimental'
import { createWebStoragePersistor } from 'react-query/createWebStoragePersistor-experimental'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      cacheTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})

const localStoragePersistor = createWebStoragePersistor({storage: window.localStorage})

persistQueryClient({
  queryClient,
  persistor: localStoragePersistor,
})
```

IMPORTANT - for persist to work properly, you need to pass QueryClient a cacheTime value to override the default during hydration (as shown above).

If it is not set when creating the QueryClient instance, it will default to 300000 (5 minutes) for hydration, and the stored cache will be discarded after 5 minutes of inactivity. This is the default garbage collection behavior.

It should be set as the same value or higher than persistQueryClient's maxAge option. E.g. if maxAge is 24 hours (the default) then cacheTime should be 24 hours or higher. If lower than maxAge, garbage collection will kick in and discard the stored cache earlier than expected.

You can also pass it Infinity to disable garbage collection behavior entirely.

How does it work?

As you use your application:

- When your query/mutation cache is updated, it will be dehydrated and stored by the persistor you provided. By default, this action is throttled to happen at most every 1 second to save on potentially expensive writes to a persistor, but can be customized as you see fit.

When you reload/bootstrap your app:

- Attempts to load a previously persisted dehydrated query/mutation cache from the persistor
- If a cache is found that is older than the maxAge (which by default is 24 hours), it will be discarded. This can be customized as you see fit.

Cache Busting

Sometimes you may make changes to your application or data that immediately invalidate any and all cached data. If and when this happens, you can pass a buster string option to persistQueryClient, and if the cache that is found does not also have that buster string, it will be discarded.

ts

```
persistQueryClient({ queryClient, persistor, buster: buildHash })
```

API persistQueryClient

Pass this function a QueryClient instance and a persistor that will persist your cache. Both are required

ts

```
persistQueryClient({ queryClient, persistor })
```

Options

An object of options:

ts

```
interface PersistQueryClientOptions {
 /** The QueryClient to persist */
 queryClient: QueryClient
 /** The Persistor interface for storing and restoring the cache
  * to/from a persisted location */
```

```
  persistor: Persistor
  /** The max-allowed age of the cache.
   * If a persisted cache is found that is older than this
   * time, it will be discarded */
  maxAge?: number
  /** A unique string that can be used to forcefully
   * invalidate existing caches if they do not share the same buster string */
  buster?: string
  /** The options passed to the hydrate function */
  hydrateOptions?: HydrateOptions
  /** The options passed to the dehydrate function */
  dehydrateOptions?: DehydrateOptions
}
```

The default options are:

ts

```
{
  maxAge = 1000 * 60 * 60 * 24, // 24 hours
  buster = '',
}
```

Building a Persistor

Persistors have the following interface:

ts

```
export interface Persistor {
  persistClient(persistClient: PersistedClient): Promisable<void>
  restoreClient(): Promisable<PersistedClient | undefined>
  removeClient(): Promisable<void>
}
```

Persisted Client entries have the following interface:

ts

```
export interface PersistedClient {
  timestamp: number
  buster: string
  cacheState: any
}
```

Satisfy all of these interfaces and you've got yourself a persistor!

createWebStoragePersistor (Experimental)

VERY IMPORTANT: This utility is currently in an experimental stage. This means that breaking changes will happen in minor AND patch releases. Use at your own risk. If you choose to rely on this in production in an experimental stage, please lock your version to a patch-level version to avoid unexpected breakages.

Installation

This utility comes packaged with react-query and is available under the react-query/createWebStoragePersistor-experimental import.

Usage

- Import the createWebStoragePersistor function
- Create a new webStoragePersistor
- Pass it to the persistQueryClient function

ts

```
import { persistQueryClient } from 'react-query/persistQueryClient-experimental'
import { createWebStoragePersistor } from 'react-query/createWebStoragePersistor-experimental'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      cacheTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})

const localStoragePersistor = createWebStoragePersistor({ storage: window.localStorage })
// const sessionStoragePersistor = createWebStoragePersistor({ storage: window.sessionStorage })

persistQueryClient({
  queryClient,
  persistor: localStoragePersistor,
})
```

API

createWebStoragePersistor

Call this function to create a webStoragePersistor that you can use later with persistQueryClient.

js

```
createWebStoragePersistor(options: CreateWebStoragePersistorOptions)
```

Options

ts

```
interface CreateWebStoragePersistorOptions {
 /** The storage client used for setting an retrieving items from cache (window.localStorage or
window.sessionStorage) */
 storage: Storage
 /** The key to use when storing the cache */
 key?: string
 /** To avoid spamming,
  * pass a time in ms to throttle saving the cache to disk */
 throttleTime?: number
 /** How to serialize the data to storage */
 serialize?: (client: PersistedClient) => string
 /** How to deserialize the data from storage */
 deserialize?: (cachedString: string) => PersistedClient
}
```

The default options are:

js

```
{
 key = `REACT_QUERY_OFFLINE_CACHE`,
 throttleTime = 1000,
 serialize = JSON.stringify,
 deserialize = JSON.parse,
}
```

createAsyncStoragePersistor (Experimental)

VERY IMPORTANT: This utility is currently in an experimental stage. This means that breaking changes will happen in minor AND patch releases. Use at your own risk. If you choose to rely on this in production in an experimental stage, please lock your version to a patch-level version to avoid unexpected breakages.

Installation

This utility comes packaged with react-query and is available under the react-query/createAsyncStoragePersistor-experimental import.

Usage

- Import the createAsyncStoragePersistor function
- Create a new asyncStoragePersistor
  - you can pass any storage to it that adheres to the AsyncStorage interface - the example below uses the async-storage from React Native
- Pass it to the persistQueryClient function

ts

```
import AsyncStorage from '@react-native-async-storage/async-storage'
import { persistQueryClient } from 'react-query/persistQueryClient-experimental'
import { createAsyncStoragePersistor } from 'react-query/createAsyncStoragePersistor-experimental'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      cacheTime: 1000 * 60 * 60 * 24, // 24 hours
    },
  },
})

const asyncStoragePersistor = createAsyncStoragePersistor({
  storage: AsyncStorage
})

persistQueryClient({
  queryClient,
  persistor: asyncStoragePersistor,
})
```

API

createAsyncStoragePersistor

Call this function to create an asyncStoragePersistor that you can use later with persistQueryClient.

js

```
createAsyncStoragePersistor(options: CreateAsyncStoragePersistorOptions)
```

Options

ts

```
interface CreateAsyncStoragePersistorOptions {
  /** The storage client used for setting an retrieving items from cache */
  storage: AsyncStorage
  /** The key to use when storing the cache to localStorage */
  key?: string
  /** To avoid localStorage spamming,
   * pass a time in ms to throttle saving the cache to disk */
  throttleTime?: number
```

```
  /** How to serialize the data to storage */
  serialize?: (client: PersistedClient) => string
  /** How to deserialize the data from storage */
  deserialize?: (cachedString: string) => PersistedClient
}

interface AsyncStorage {
  getItem: (key: string) => Promise<string>
  setItem: (key: string, value: string) => Promise<unknown>
  removeItem: (key: string) => Promise<unknown>
}
```

The default options are:

js

```
{
  key = `REACT_QUERY_OFFLINE_CACHE`,
  throttleTime = 1000,
  serialize = JSON.stringify,
  deserialize = JSON.parse,
}
```

broadcastQueryClient (Experimental)

VERY IMPORTANT: This utility is currently in an experimental stage. This means that breaking changes will happen in minor AND patch releases. Use at your own risk. If you choose to rely on this in production in an experimental stage, please lock your version to a patch-level version to avoid unexpected breakages.

broadcastQueryClient is a utility for broadcasting and syncing the state of your queryClient between browser tabs/windows with the same origin.

Installation

This utility comes packaged with react-query and is available under the react-query/broadcastQueryClient-experimental import.

Usage

Import the broadcastQueryClient function, and pass it your QueryClient instance, and optionally, set a broadcastChannel.

ts

```
import { broadcastQueryClient } from 'react-query/broadcastQueryClient-experimental'

const queryClient = new QueryClient()
```

```
broadcastQueryClient({
  queryClient,
  broadcastChannel: 'my-app',
})
```

API

broadcastQueryClient

Pass this function a QueryClient instance and optionally, a broadcastChannel.

ts

```
broadcastQueryClient({ queryClient, broadcastChannel })
```

Options

An object of options:

ts

```
interface broadcastQueryClient {
  /** The QueryClient to sync */
  queryClient: QueryClient
  /** This is the unique channel name that will be used
   * to communicate between tabs and windows */
  broadcastChannel?: string
}
```

The default options are:

```
{
  broadcastChannel = 'react-query',
}
```

API REFERENCE

useQuery

```
const {
  data,
  dataUpdatedAt,
  error,
  errorUpdatedAt,
  failureCount,
  isError,
  isFetched,
  isFetchedAfterMount,
  isFetching,
  isIdle,
  isLoading,
  isLoadingError,
  isPlaceholderData,
  isPreviousData,
  isRefetchError,
  isRefetching,
  isStale,
  isSuccess,
  refetch,
  remove,
  status,
} = useQuery(queryKey, queryFn?, {
  cacheTime,
  enabled,
  initialData,
  initialDataUpdatedAt
  isDataEqual,
  keepPreviousData,
  meta,
  notifyOnChangeProps,
  notifyOnChangePropsExclusions,
  onError,
  onSettled,
  onSuccess,
  placeholderData,
  queryKeyHashFn,
  refetchInterval,
  refetchIntervalInBackground,
  refetchOnMount,
  refetchOnReconnect,
  refetchOnWindowFocus,
  retry,
  retryOnMount,
  retryDelay,
  select,
  staleTime,
  structuralSharing,
  suspense,
  useErrorBoundary,
})

// or using the object syntax

const result = useQuery({
  queryKey,
  queryFn,
  enabled,
})
```

Options

- **queryKey: string | unknown[]**
  - **Required**
  - The query key to use for this query.
  - The query key will be hashed into a stable hash. See Query Keys for more information.
  - The query will automatically update when this key changes (as long as **enabled** is not set to **false**).
- **queryFn: (context: QueryFunctionContext) => Promise<TData>**
  - **Required, but only if no default query function has been defined** See Default Query Function for more information.
  - The function that the query will use to request data.
  - Receives a **QueryFunctionContext** object with the following variables:
    - **queryKey: EnsuredQueryKey**: the queryKey, guaranteed to be an Array
  - Must return a promise that will either resolve data or throw an error.
- **enabled: boolean**
  - Set this to **false** to disable this query from automatically running.
  - Can be used for Dependent Queries.
- **retry: boolean | number | (failureCount: number, error: TError) => boolean**
  - If **false**, failed queries will not retry by default.
  - If **true**, failed queries will retry infinitely.
  - If set to a **number**, e.g. **3**, failed queries will retry until the failed query count meets that number.
- **retryOnMount: boolean**
  - If set to **false**, the query will not be retried on mount if it contains an error. Defaults to **true**.
- **retryDelay: number | (retryAttempt: number, error: TError) => number**
  - This function receives a **retryAttempt** integer and the actual Error and returns the delay to apply before the next attempt in milliseconds.
  - A function like **attempt => Math.min(attempt > 1 ? 2 ** attempt * 1000 : 1000, 30 * 1000)** applies exponential backoff.
  - A function like **attempt => attempt * 1000** applies linear backoff.
- **staleTime: number | Infinity**
  - Optional
  - Defaults to **0**
  - The time in milliseconds after data is considered stale. This value only applies to the hook it is defined on.
  - If set to **Infinity**, the data will never be considered stale
- **cacheTime: number | Infinity**
  - Defaults to **5 * 60 * 1000** (5 minutes)
  - The time in milliseconds that unused/inactive cache data remains in memory. When a query's cache becomes unused or inactive, that cache data will be garbage collected after this duration. When different cache times are specified, the longest one will be used.
  - If set to **Infinity**, will disable garbage collection
- **queryKeyHashFn: (queryKey: QueryKey) => string**
  - Optional
  - If specified, this function is used to hash the **queryKey** to a string.

- **refetchInterval: number | false | ((data: TData | undefined, query: Query) => number | false)**
    - Optional
    - If set to a number, all queries will continuously refetch at this frequency in milliseconds
    - If set to a function, the function will be executed with the latest data and query to compute a frequency
- **refetchIntervalInBackground: boolean**
    - Optional
    - If set to **true**, queries that are set to continuously refetch with a **refetchInterval** will continue to refetch while their tab/window is in the background
- **refetchOnMount: boolean | "always" | ((query: Query) => boolean | "always")**
    - Optional
    - Defaults to **true**
    - If set to **true**, the query will refetch on mount if the data is stale.
    - If set to **false**, the query will not refetch on mount.
    - If set to **"always"**, the query will always refetch on mount.
    - If set to a function, the function will be executed with the query to compute the value
- **refetchOnWindowFocus: boolean | "always" | ((query: Query) => boolean | "always")**
    - Optional
    - Defaults to **true**
    - If set to **true**, the query will refetch on window focus if the data is stale.
    - If set to **false**, the query will not refetch on window focus.
    - If set to **"always"**, the query will always refetch on window focus.
    - If set to a function, the function will be executed with the query to compute the value
- **refetchOnReconnect: boolean | "always" | ((query: Query) => boolean | "always")**
    - Optional
    - Defaults to **true**
    - If set to **true**, the query will refetch on reconnect if the data is stale.
    - If set to **false**, the query will not refetch on reconnect.
    - If set to **"always"**, the query will always refetch on reconnect.
    - If set to a function, the function will be executed with the query to compute the value
- **notifyOnChangeProps: string[] | "tracked"**
    - Optional
    - If set, the component will only re-render if any of the listed properties change.
    - If set to **['data', 'error']** for example, the component will only re-render when the **data** or **error** properties change.
    - If set to **"tracked"**, access to properties will be tracked, and the component will only re-render when one of the tracked properties change.
- **notifyOnChangePropsExclusions: string[]**
    - Optional
    - If set, the component will not re-render if any of the listed properties change.
    - If set to **['isStale']** for example, the component will not re-render when the **isStale** property changes.
- **onSuccess: (data: TData) => void**

    - Optional
    - This function will fire any time the query successfully fetches new data or the cache is updated via **setQueryData**.
- **onError: (error: TError) => void**
    - Optional
    - This function will fire if the query encounters an error and will be passed the error.
- **onSettled: (data?: TData, error?: TError) => void**
    - Optional
    - This function will fire any time the query is either successfully fetched or errors and be passed either the data or error
- **select: (data: TData) => unknown**
    - Optional
    - This option can be used to transform or select a part of the data returned by the query function.
- **suspense: boolean**
    - Optional
    - Set this to **true** to enable suspense mode.
    - When **true**, **useQuery** will suspend when **status === 'loading'**
    - When **true**, **useQuery** will throw runtime errors when **status === 'error'**
- **initialData: TData | () => TData**
    - Optional
    - If set, this value will be used as the initial data for the query cache (as long as the query hasn't been created or cached yet)
    - If set to a function, the function will be called **once** during the shared/root query initialization, and be expected to synchronously return the initialData
    - Initial data is considered stale by default unless a **staleTime** has been set.
    - **initialData is persisted** to the cache
- **initialDataUpdatedAt: number | (() => number | undefined)**
    - Optional
    - If set, this value will be used as the time (in milliseconds) of when the **initialData** itself was last updated.
- **placeholderData: TData | () => TData**
    - Optional
    - If set, this value will be used as the placeholder data for this particular query observer while the query is still in the **loading** data and no initialData has been provided.
    - **placeholderData** is **not persisted** to the cache
- **keepPreviousData: boolean**
    - Optional
    - Defaults to **false**
    - If set, any previous **data** will be kept when fetching new data because the query key changed.
- **structuralSharing: boolean**
    - Optional
    - Defaults to **true**
    - If set to **false**, structural sharing between query results will be disabled.
- **useErrorBoundary: undefined | boolean | (error: TError, query: Query) => boolean**

        - Defaults to the global query config's **useErrorBoundary** value, which is **undefined**
        - Set this to **true** if you want errors to be thrown in the render phase and propagate to the nearest error boundary
        - Set this to **false** to disable **suspense**'s default behavior of throwing errors to the error boundary.
        - If set to a function, it will be passed the error and the query, and it should return a boolean indicating whether to show the error in an error boundary (**true**) or return the error as state (**false**)
- **meta: Record<string, unknown>**
    - Optional
    - If set, stores additional information on the query cache entry that can be used as needed. It will be accessible wherever the **query** is available, and is also part of the **QueryFunctionContext** provided to the **queryFn**.

Returns

- **status: String**
    - Will be:
        - **idle** if the query is idle. This only happens if a query is initialized with **enabled: false** and no initial data is available.
        - **loading** if the query is in a "hard" loading state. This means there is no cached data and the query is currently fetching, eg **isFetching === true**
        - **error** if the query attempt resulted in an error. The corresponding **error** property has the error received from the attempted fetch
        - **success** if the query has received a response with no errors and is ready to display its data. The corresponding **data** property on the query is the data received from the successful fetch or if the query's **enabled** property is set to **false** and has not been fetched yet **data** is the first **initialData** supplied to the query on initialization.
- **isIdle: boolean**
    - A derived boolean from the **status** variable above, provided for convenience.
- **isLoading: boolean**
    - A derived boolean from the **status** variable above, provided for convenience.
- **isSuccess: boolean**
    - A derived boolean from the **status** variable above, provided for convenience.
- **isError: boolean**
    - A derived boolean from the **status** variable above, provided for convenience.
- **isLoadingError: boolean**
    - Will be **true** if the query failed while fetching for the first time.
- **isRefetchError: boolean**
    - Will be **true** if the query failed while refetching.
- **data: TData**
    - Defaults to **undefined**.
    - The last successfully resolved data for the query.
- **dataUpdatedAt: number**
    - The timestamp for when the query most recently returned the **status** as **"success"**.
- **error: null | TError**
    - Defaults to **null**

    - The error object for the query, if an error was thrown.
- **errorUpdatedAt: number**
    - The timestamp for when the query most recently returned the **status** as **"error"**.
- **isStale: boolean**
    - Will be **true** if the data in the cache is invalidated or if the data is older than the given **staleTime**.
- **isPlaceholderData: boolean**
    - Will be **true** if the data shown is the placeholder data.
- **isPreviousData: boolean**
    - Will be **true** when **keepPreviousData** is set and data from the previous query is returned.
- **isFetched: boolean**
    - Will be **true** if the query has been fetched.
- **isFetchedAfterMount: boolean**
    - Will be **true** if the query has been fetched after the component mounted.
    - This property can be used to not show any previously cached data.
- **isFetching: boolean**
    - Is **true** whenever a request is in-flight, which includes initial **loading** as well as background refetches.
    - Will be **true** if the query is currently fetching, including background fetching.
- **isRefetching: boolean**
    - Is **true** whenever a background refetch is in-flight, which *does not* include initial **loading**
    - Is the same as **isFetching && !isLoading**
- **failureCount: number**
    - The failure count for the query.
    - Incremented every time the query fails.
    - Reset to **0** when the query succeeds.
- **errorUpdateCount: number**
    - The sum of all errors.
- **refetch: (options: { throwOnError: boolean, cancelRefetch: boolean }) => Promise<UseQueryResult>**
    - A function to manually refetch the query.
    - If the query errors, the error will only be logged. If you want an error to be thrown, pass the **throwOnError: true** option
    - If **cancelRefetch** is **true**, then the current request will be cancelled before a new request is made
- **remove: () => void**
    - A function to remove the query from the cache.

useQueries

The **useQueries** hook can be used to fetch a variable number of queries:

```
const results = useQueries([
  { queryKey: ['post', 1], queryFn: fetchPost },
  { queryKey: ['post', 2], queryFn: fetchPost },
])
```

Options

The **useQueries** hook accepts an array with query option objects identical to the **useQuery** hook.

Returns

The **useQueries** hook returns an array with all the query results.

useInfiniteQuery

js

```
const {
  fetchNextPage,
  fetchPreviousPage,
  hasNextPage,
  hasPreviousPage,
  isFetchingNextPage,
  isFetchingPreviousPage,
  ...result
} = useInfiniteQuery(queryKey, ({ pageParam = 1 }) => fetchPage(pageParam), {
  ...options,
  getNextPageParam: (lastPage, allPages) => lastPage.nextCursor,
  getPreviousPageParam: (firstPage, allPages) => firstPage.prevCursor,
})
```

Options

The options for **useInfiniteQuery** are identical to the **useQuery** hook with the addition of the following:

- **queryFn: (context: QueryFunctionContext) => Promise<TData>**
  - **Required, but only if no default query function has been defined defaultQueryFn**
  - The function that the query will use to request data.
  - Receives a **QueryFunctionContext** object with the following variables:
    - **queryKey: EnsuredQueryKey**: the queryKey, guaranteed to be an Array
    - **pageParam: unknown | undefined**
  - Must return a promise that will either resolve data or throw an error.
  - Make sure you return the data *and* the **pageParam** if needed for use in the props below.
- **getNextPageParam: (lastPage, allPages) => unknown | undefined**
  - When new data is received for this query, this function receives both the last page of the infinite list of data and the full array of all pages.
  - It should return a **single variable** that will be passed as the last optional parameter to your query function.
  - Return **undefined** to indicate there is no next page available.
- **getPreviousPageParam: (firstPage, allPages) => unknown | undefined**

    - When new data is received for this query, this function receives both the first page of the infinite list of data and the full array of all pages.
    - It should return a **single variable** that will be passed as the last optional parameter to your query function.
    - Return **undefined** to indicate there is no previous page available.

Returns

The returned properties for **useInfiniteQuery** are identical to the **useQuery** hook, with the addition of the following:

- **data.pages: TData[]**
    - Array containing all pages.
- **data.pageParams: unknown[]**
    - Array containing all page params.
- **isFetchingNextPage: boolean**
    - Will be **true** while fetching the next page with **fetchNextPage**.
- **isFetchingPreviousPage: boolean**
    - Will be **true** while fetching the previous page with **fetchPreviousPage**.
- **fetchNextPage: (options?: FetchNextPageOptions) => Promise<UseInfiniteQueryResult>**
    - This function allows you to fetch the next "page" of results.
    - **options.pageParam: unknown** allows you to manually specify a page param instead of using **getNextPageParam**.
    - **options.cancelRefetch: boolean** if set to **true**, calling **fetchNextPage** repeatedly will invoke **fetchPage** every time, whether the previous invocation has resolved or not. Also, the result from previous invocations will be ignored. If set to **false**, calling **fetchNextPage** repeatedly won't have any effect until the first invocation has resolved. Default is **true**.
- **fetchPreviousPage: (options?: FetchPreviousPageOptions) => Promise<UseInfiniteQueryResult>**
    - This function allows you to fetch the previous "page" of results.
    - **options.pageParam: unknown** allows you to manually specify a page param instead of using **getPreviousPageParam**.
    - **options.cancelRefetch: boolean** same as for **fetchNextPage**.
- **hasNextPage: boolean**
    - This will be **true** if there is a next page to be fetched (known via the **getNextPageParam** option).
- **hasPreviousPage: boolean**
    - This will be **true** if there is a previous page to be fetched (known via the **getPreviousPageParam** option).

useMutation

```
const {
  data,
  error,
  isError,
  isIdle,
  isLoading,
  isPaused,
  isSuccess,
  mutate,
  mutateAsync,
  reset,
  status,
} = useMutation(mutationFn, {
  mutationKey,
  onError,
  onMutate,
  onSettled,
  onSuccess,
  retry,
  retryDelay,
  useErrorBoundary,
  meta,
})

mutate(variables, {
  onError,
  onSettled,
  onSuccess,
})
```

Options

- **mutationFn: (variables: TVariables) => Promise<TData>**
  - **Required**
  - A function that performs an asynchronous task and returns a promise.
  - **variables** is an object that **mutate** will pass to your **mutationFn**
- **mutationKey: string**
  - Optional
  - A mutation key can be set to inherit defaults set with **queryClient.setMutationDefaults** or to identify the mutation in the devtools.
- **onMutate: (variables: TVariables) => Promise<TContext | void> | TContext | void**
  - Optional
  - This function will fire before the mutation function is fired and is passed the same variables the mutation function would receive
  - Useful to perform optimistic updates to a resource in hopes that the mutation succeeds
  - The value returned from this function will be passed to both the **onError** and **onSettled** functions in the event of a mutation failure and can be useful for rolling back optimistic updates.

- **onSuccess: (data: TData, variables: TVariables, context?: TContext) => Promise<unknown> | void**
    - Optional
    - This function will fire when the mutation is successful and will be passed the mutation's result.
    - If a promise is returned, it will be awaited and resolved before proceeding
- **onError: (err: TError, variables: TVariables, context?: TContext) => Promise<unknown> | void**
    - Optional
    - This function will fire if the mutation encounters an error and will be passed the error.
    - If a promise is returned, it will be awaited and resolved before proceeding
- **onSettled: (data: TData, error: TError, variables: TVariables, context?: TContext) => Promise<unknown> | void**
    - Optional
    - This function will fire when the mutation is either successfully fetched or encounters an error and be passed either the data or error
    - If a promise is returned, it will be awaited and resolved before proceeding
- **retry: boolean | number | (failureCount: number, error: TError) => boolean**
    - Defaults to **0**.
    - If **false**, failed mutations will not retry.
    - If **true**, failed mutations will retry infinitely.
    - If set to an **number**, e.g. **3**, failed mutations will retry until the failed mutations count meets that number.
- **retryDelay: number | (retryAttempt: number, error: TError) => number**
    - This function receives a **retryAttempt** integer and the actual Error and returns the delay to apply before the next attempt in milliseconds.
    - A function like **attempt => Math.min(attempt > 1 ? 2 ** attempt * 1000 : 1000, 30 * 1000)** applies exponential backoff.
    - A function like **attempt => attempt * 1000** applies linear backoff.
- **useErrorBoundary: undefined | boolean | (error: TError) => boolean**
    - Defaults to the global query config's **useErrorBoundary** value, which is **undefined**
    - Set this to **true** if you want mutation errors to be thrown in the render phase and propagate to the nearest error boundary
    - Set this to **false** to disable the behavior of throwing errors to the error boundary.
    - If set to a function, it will be passed the error and should return a boolean indicating whether to show the error in an error boundary (**true**) or return the error as state (**false**)
- **meta: Record<string, unknown>**
    - Optional
    - If set, stores additional information on the mutation cache entry that can be used as needed. It will be accessible wherever the **mutation** is available (eg. **onError**, **onSuccess** functions of the **MutationCache**).

Returns

- **mutate: (variables: TVariables, { onSuccess, onSettled, onError }) => void**
    - The mutation function you can call with variables to trigger the mutation and optionally override options passed to **useMutation**.
    - **variables: TVariables**

        - Optional
        - The variables object to pass to the **mutationFn**.
    - Remaining options extend the same options described above in the **useMutation** hook.
    - If you make multiple requests, **onSuccess** will fire only after the latest call you've made.
- **mutateAsync: (variables: TVariables, { onSuccess, onSettled, onError }) => Promise<TData>**
    - Similar to **mutate** but returns a promise which can be awaited.
- **status: string**
    - Will be:
        - **idle** initial status prior to the mutation function executing.
        - **loading** if the mutation is currently executing.
        - **error** if the last mutation attempt resulted in an error.
        - **success** if the last mutation attempt was successful.
- **isIdle**, **isLoading**, **isSuccess**, **isError**: boolean variables derived from **status**
- **data: undefined | unknown**
    - Defaults to **undefined**
    - The last successfully resolved data for the query.
- **error: null | TError**
    - The error object for the query, if an error was encountered.
- **reset: () => void**
    - A function to clean the mutation internal state (i.e., it resets the mutation to its initial state).

useIsFetching

useIsFetching is an optional hook that returns the **number** of the queries that your application is loading or fetching in the background (useful for app-wide loading indicators).
Js

```
import { useIsFetching } from 'react-query'
// How many queries are fetching?
const isFetching = useIsFetching()
// How many queries matching the posts prefix are fetching?
const isFetchingPosts = useIsFetching(['posts'])
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

Returns

- **isFetching: number**
    - Will be the **number** of the queries that your application is currently loading or fetching in the background.

useIsMutating

useIsMutating is an optional hook that returns the **number** of mutations that your application is fetching (useful for app-wide loading indicators).

js

```
import { useIsMutating } from 'react-query'
// How many mutations are fetching?
const isMutating = useIsMutating()
// How many mutations matching the posts prefix are fetching?
const isMutatingPosts = useIsMutating(['posts'])
```

Options

- **mutationKey?: string | unknown[]**
- **filters?: MutationFilters**: Mutation Filters

Returns

- **isMutating: number**
  - Will be the **number** of the mutations that your application is currently fetching.

QueryClient

QueryClient

The **QueryClient** can be used to interact with a cache:

js

```
import { QueryClient } from 'react-query'

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

await queryClient.prefetchQuery('posts', fetchPosts)

Its available methods are:

- **queryClient.fetchQuery**
- **queryClient.fetchInfiniteQuery**
- **queryClient.prefetchQuery**
- **queryClient.prefetchInfiniteQuery**
- **queryClient.getQueryData**
- **queryClient.getQueriesData**
- **queryClient.setQueryData**
- **queryClient.getQueryState**
- **queryClient.setQueriesData**
- **queryClient.invalidateQueries**
- **queryClient.refetchQueries**

- **queryClient.cancelQueries**
- **queryClient.removeQueries**
- **queryClient.resetQueries**
- **queryClient.isFetching**
- **queryClient.isMutating**
- **queryClient.getDefaultOptions**
- **queryClient.setDefaultOptions**
- **queryClient.getQueryDefaults**
- **queryClient.setQueryDefaults**
- **queryClient.getMutationDefaults**
- **queryClient.setMutationDefaults**
- **queryClient.getQueryCache**
- **queryClient.getMutationCache**
- **queryClient.clear**

Options

- **queryCache?: QueryCache**
  - Optional
  - The query cache this client is connected to.
- **mutationCache?: MutationCache**
  - Optional
  - The mutation cache this client is connected to.
- **defaultOptions?: DefaultOptions**
  - Optional
  - Define defaults for all queries and mutations using this queryClient.

queryClient.fetchQuery

fetchQuery is an asynchronous method that can be used to fetch and cache a query. It will either resolve with the data or throw with the error. Use the **prefetchQuery** method if you just want to fetch a query without needing the result.
If the query exists and the data is not invalidated or older than the given **staleTime**, then the data from the cache will be returned. Otherwise it will try to fetch the latest data.
*The difference between using **fetchQuery** and **setQueryData** is that **fetchQuery** is async and will ensure that duplicate requests for this query are not created with **useQuery** instances for the same query are rendered while the data is fetching.*

```
try {
  const data = await queryClient.fetchQuery(queryKey, queryFn)
} catch (error) {
  console.log(error)
}
```

Specify a **staleTime** to only fetch when the data is older than a certain amount of time:

```
try {
  const data = await queryClient.fetchQuery(queryKey, queryFn, {
    staleTime: 10000,
  })
} catch (error) {
  console.log(error)
}
```

Options

The options for **fetchQuery** are exactly the same as those of **useQuery**, except the following: **enabled, refetchInterval, refetchIntervalInBackground, refetchOnWindowFocus, refetchOnReconnect, notifyOnChangeProps, notifyOnChangePropsExclusions, onSuccess, onError, onSettled, useErrorBoundary, select, suspense, keepPreviousData, placeholderData**; which are strictly for useQuery and useInfiniteQuery. You can check the source code for more clarity.

Returns

- **Promise<TData>**

queryClient.fetchInfiniteQuery

fetchInfiniteQuery is similar to **fetchQuery** but can be used to fetch and cache an infinite query.

```
try {
  const data = await queryClient.fetchInfiniteQuery(queryKey, queryFn)
  console.log(data.pages)
} catch (error) {
  console.log(error)
}
```

Options

The options for **fetchInfiniteQuery** are exactly the same as those of **fetchQuery**.

Returns

- **Promise<InfiniteData<TData>>**

queryClient.prefetchQuery

prefetchQuery is an asynchronous method that can be used to prefetch a query before it is needed or rendered with **useQuery** and friends. The method works the same as **fetchQuery** except that it will not throw or return any data.

js

```
await queryClient.prefetchQuery(queryKey, queryFn)
```

You can even use it with a default queryFn in your config!

js

```
await queryClient.prefetchQuery(queryKey)
```

Options

The options for **prefetchQuery** are exactly the same as those of **fetchQuery**.

Returns

- **Promise<void>**
  - A promise is returned that will either immediately resolve if no fetch is needed or after the query has been executed. It will not return any data or throw any errors.

queryClient.prefetchInfiniteQuery

prefetchInfiniteQuery is similar to **prefetchQuery** but can be used to prefetch and cache an infinite query.

```
await queryClient.prefetchInfiniteQuery(queryKey, queryFn)
```

Options

The options for **prefetchInfiniteQuery** are exactly the same as those of **fetchQuery**.

Returns

- **Promise<void>**
  - A promise is returned that will either immediately resolve if no fetch is needed or after the query has been executed. It will not return any data or throw any errors.

queryClient.getQueryData

getQueryData is a synchronous function that can be used to get an existing query's cached data. If the query does not exist, **undefined** will be returned.

js

```
const data = queryClient.getQueryData(queryKey)
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

Returns

- **data: TData | undefined**
  - The data for the cached query, or **undefined** if the query does not exist.

queryClient.getQueriesData

getQueriesData is a synchronous function that can be used to get the cached data of multiple queries. Only queries that match the passed queryKey or queryFilter will be returned. If there are no matching queries, an empty array will be returned.

js

```
const data = queryClient.getQueriesData(queryKey | filters)
```

Options

- **queryKey: QueryKey**: Query Keys | **filters: QueryFilters**: Query Filters
  - if a queryKey is passed as the argument, the data with queryKeys fuzzily matching this param will be returned
  - if a filter is passed, the data with queryKeys matching the filter will be returned

Returns

- **[queryKey:QueryKey, data:TData | unknown][]**
  - An array of tuples for the matched query keys, or **[]** if there are no matches. The tuples are the query key and its associated data.

Caveats

Because the returned data in each tuple can be of varying structures (i.e. using a filter to return "active" queries can return different data types), the **TData** generic defaults to **unknown**. If you provide a more specific type to **TData** it is assumed that you are certain each tuple's data entry is all the same type.

This distinction is more a "convenience" for ts devs that know which structure will be returned.

queryClient.setQueryData

setQueryData is a synchronous function that can be used to immediately update a query's cached data. If the query does not exist, it will be created. **If the query is not utilized by a query**

hook in the default cacheTime of 5 minutes, the query will be garbage collected. To update multiple queries at once and match query keys partially, you need to use **queryClient.setQueriesData** instead.
After successful changing query's cached data via **setQueryData**, it will also trigger **onSuccess** callback from that query.

> *The difference between using **setQueryData** and **fetchQuery** is that **setQueryData** is sync and assumes that you already synchronously have the data available. If you need to fetch the data asynchronously, it's suggested that you either refetch the query key or use **fetchQuery** to handle the asynchronous fetch.*

js

```
queryClient.setQueryData(queryKey, updater)
```

Options

- **queryKey: QueryKey**: Query Keys
- **updater: TData | (oldData: TData | undefined) => TData**
  - If non-function is passed, the data will be updated to this value
  - If a function is passed, it will receive the old data value and be expected to return a new one.

Using an updater value

js

```
setQueryData(queryKey, newData)
```

Using an updater function

For convenience in syntax, you can also pass an updater function which receives the current data value and returns the new one:

js

```
setQueryData(queryKey, oldData => newData)
```

queryClient.getQueryState

getQueryState is a synchronous function that can be used to get an existing query's state. If the query does not exist, **undefined** will be returned.

js

```
const state = queryClient.getQueryState(queryKey)
console.log(state.dataUpdatedAt)
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

queryClient.setQueriesData

setQueriesData is a synchronous function that can be used to immediately update cached data of multiple queries by using filter function or partially matching the query key. Only queries that match the passed queryKey or queryFilter will be updated - no new cache entries will be created. Under the hood, **setQueryData** is called for each query.

Js

```
queryClient.setQueriesData(queryKey | filters, updater)
```

Options

- **queryKey: QueryKey**: Query Keys | **filters: QueryFilters**: Query Filters
  - if a queryKey is passed as first argument, queryKeys partially matching this param will be updated
  - if a filter is passed, queryKeys matching the filter will be updated
- **updater: TData | (oldData: TData | undefined) => TData**
  - the setQueryData updater function or new data, will be called for each matching queryKey

queryClient.invalidateQueries

The **invalidateQueries** method can be used to invalidate and refetch single or multiple queries in the cache based on their query keys or any other functionally accessible property/state of the query. By default, all matching queries are immediately marked as invalid and active queries are refetched in the background.

- If you **do not want active queries to refetch**, and simply be marked as invalid, you can use the **refetchActive: false** option.
- If you **want inactive queries to refetch** as well, use the **refetchInactive: true** option

Js

```
await queryClient.invalidateQueries('posts', {
  exact,
  refetchActive: true,
  refetchInactive: false
}, { throwOnError, cancelRefetch })
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters
  - **refetchActive: Boolean**
    - Defaults to **true**
    - When set to **false**, queries that match the refetch predicate and are actively being rendered via **useQuery** and friends will NOT be refetched in the background, and only marked as invalid.
  - **refetchInactive: Boolean**
    - Defaults to **false**
    - When set to **true**, queries that match the refetch predicate and are not being rendered via **useQuery** and friends will be both marked as invalid and also refetched in the background
  - **refetchPage: (page: TData, index: number, allPages: TData[]) => boolean**
    - Only for Infinite Queries
    - Use this function to specify which pages should be refetched
- **options?: InvalidateOptions**:
  - **throwOnError?: boolean**
    - When set to **true**, this method will throw if any of the query refetch tasks fail.
  - cancelRefetch?: boolean

- When set to **true**, then the current request will be cancelled before a new request is made

queryClient.refetchQueries

The **refetchQueries** method can be used to refetch queries based on certain conditions.
Examples:
js

```
// refetch all queries:
await queryClient.refetchQueries()

// refetch all stale queries:
await queryClient.refetchQueries({ stale: true })

// refetch all active queries partially matching a query key:
await queryClient.refetchQueries(['posts'], { active: true })

// refetch all active queries exactly matching a query key:
await queryClient.refetchQueries(['posts', 1], { active: true, exact: true })
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters
  - **refetchPage: (page: TData, index: number, allPages: TData[]) => boolean**
    - Only for Infinite Queries
    - Use this function to specify which pages should be refetched
- **options?: RefetchOptions**:
  - **throwOnError?: boolean**
    - When set to **true**, this method will throw if any of the query refetch tasks fail.
  - cancelRefetch?: boolean
    - When set to **true**, then the current request will be cancelled before a new request is made

Returns
This function returns a promise that will resolve when all of the queries are done being refetched. By default, it **will not** throw an error if any of those queries refetches fail, but this can be configured by setting the **throwOnError** option to **true**

queryClient.cancelQueries

The **cancelQueries** method can be used to cancel outgoing queries based on their query keys or any other functionally accessible property/state of the query.
This is most useful when performing optimistic updates since you will likely need to cancel any outgoing query refetches so they don't clobber your optimistic update when they resolve.
js

```
await queryClient.cancelQueries('posts', { exact: true })
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters
-

Returns
This method does not return anything

queryClient.removeQueries

The **removeQueries** method can be used to remove queries from the cache based on their query keys or any other functionally accessible property/state of the query.
js
queryClient.removeQueries(queryKey, { exact: true })
Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

Returns
This method does not return anything

queryClient.resetQueries

The **resetQueries** method can be used to reset queries in the cache to their initial state based on their query keys or any other functionally accessible property/state of the query.
This will notify subscribers — unlike **clear**, which removes all subscribers — and reset the query to its pre-loaded state — unlike **invalidateQueries**. If a query has **initialData**, the query's data will be reset to that. If a query is active, it will be refetched.
js

```
queryClient.resetQueries(queryKey, { exact: true })
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters
    - **refetchPage: (page: TData, index: number, allPages: TData[]) => boolean**
        - Only for Infinite Queries
        - Use this function to specify which pages should be refetched
- **options?: ResetOptions**:
    - **throwOnError?: boolean**
        - When set to **true**, this method will throw if any of the query refetch tasks fail.
    - cancelRefetch?: boolean
        - When set to **true**, then the current request will be cancelled before a new request is made

Returns
This method returns a promise that resolves when all active queries have been refetched.

queryClient.isFetching

This **isFetching** method returns an **integer** representing how many queries, if any, in the cache are currently fetching (including background-fetching, loading new pages, or loading more infinite query results)
js

```
if (queryClient.isFetching()) {
  console.log('At least one query is fetching!')
}
```

React Query also exports a handy **useIsFetching** hook that will let you subscribe to this state in your components without creating a manual subscription to the query cache.

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

Returns

This method returns the number of fetching queries.

queryClient.isMutating

This **isMutating** method returns an **integer** representing how many mutations, if any, in the cache are currently fetching.

js

```
if (queryClient.isMutating()) {
  console.log('At least one mutation is fetching!')
}
```

React Query also exports a handy **useIsMutating** hook that will let you subscribe to this state in your components without creating a manual subscription to the mutation cache.

Options

- **filters: MutationFilters**: Mutation Filters

Returns

This method returns the number of fetching mutations.

queryClient.getDefaultOptions

The **getDefaultOptions** method returns the default options which have been set when creating the client or with **setDefaultOptions**.

js

```
const defaultOptions = queryClient.getDefaultOptions()
```

queryClient.setDefaultOptions

The **setDefaultOptions** method can be used to dynamically set the default options for this queryClient. Previously defined default options will be overwritten.

js

```
queryClient.setDefaultOptions({
  queries: {
    staleTime: Infinity,
  },
})
```

queryClient.getQueryDefaults

The **getQueryDefaults** method returns the default options which have been set for specific queries:

js

```
const defaultOptions = queryClient.getQueryDefaults('posts')
```

queryClient.setQueryDefaults

setQueryDefaults can be used to set default options for specific queries:
js

```
queryClient.setQueryDefaults('posts', { queryFn: fetchPosts })

function Component() {
  const { data } = useQuery('posts')
}
```

Options

- **queryKey: QueryKey**: Query Keys
- **options: QueryOptions**

queryClient.getMutationDefaults

The **getMutationDefaults** method returns the default options which have been set for specific mutations:
js

```
const defaultOptions = queryClient.getMutationDefaults('addPost')
```

queryClient.setMutationDefaults

setMutationDefaults can be used to set default options for specific mutations:
js

```
queryClient.setMutationDefaults('addPost', { mutationFn: addPost })

function Component() {
  const { data } = useMutation('addPost')
}
```

Options

- **mutationKey: string | unknown[]**
- **options: MutationOptions**

queryClient.getQueryCache

The **getQueryCache** method returns the query cache this client is connected to.
js

```
const queryCache = queryClient.getQueryCache()
```

queryClient.getMutationCache

The **getMutationCache** method returns the mutation cache this client is connected to.
js

```
const mutationCache = queryClient.getMutationCache()
```

queryClient.clear

The **clear** method clears all connected caches.
js

```
queryClient.clear()
```

QueryClientProvider

Use the **QueryClientProvider** component to connect and provide a **QueryClient** to your application:
js

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

const queryClient = new QueryClient()

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

Options

- **client: QueryClient**
  - **Required**
  - the QueryClient instance to provide
- **contextSharing: boolean**
  - defaults to **false**
  - Set this to **true** to enable context sharing, which will share the first and at least one instance of the context across the window to ensure that if React Query is used across different bundles or microfrontends they will all use the same **instance** of context, regardless of module scoping.

useQueryClient

The **useQueryClient** hook returns the current **QueryClient** instance.

Js

```
import { useQueryClient } from 'react-query'

const queryClient = useQueryClient()
```

QueryCache

The **QueryCache** is the storage mechanism for React Query. It stores all the data, meta information and state of queries it contains.
Normally, you will not interact with the QueryCache directly and instead use the QueryClient for a specific cache.

js

```
import { QueryCache } from 'react-query'

const queryCache = new QueryCache({
  onError: error => {
    console.log(error)
  },
  onSuccess: data => {
    console.log(data)
  }
})

const query = queryCache.find('posts')
```

Its available methods are:

- **find**
- **findAll**
- **subscribe**
- **clear**

Options

- **onError?: (error: unknown, query: Query) => void**
  - Optional
  - This function will be called if some query encounters an error.
- **onSuccess?: (data: unknown, query: Query) => void**
  - Optional
  - This function will be called if some query is successful.

Global callbacks

The **onError** and **onSuccess** callbacks on the QueryCache can be used to handle these events on a global level. They are different to **defaultOptions** provided to the QueryClient because:

- **defaultOptions** can be overridden by each Query - the global callbacks will **always** be called.
- **defaultOptions** callbacks will be called once for each Observer, while the global callbacks will only be called once per Query.

queryCache.find

find is a slightly more advanced synchronous method that can be used to get an existing query instance from the cache. This instance not only contains **all** the state for the query, but all of the instances, and underlying guts of the query as well. If the query does not exist, **undefined** will be returned.
Note: This is not typically needed for most applications, but can come in handy when needing more information about a query in rare scenarios (eg. Looking at the query.state.dataUpdatedAt timestamp to decide whether a query is fresh enough to be used as an initial value)
js

```
const query = queryCache.find(queryKey)
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

Returns

- **Query**
  - The query instance from the cache

queryCache.findAll

findAll is even more advanced synchronous method that can be used to get existing query instances from the cache that partially match query key. If queries do not exist, empty array will be returned.
Note: This is not typically needed for most applications, but can come in handy when needing more information about a query in rare scenarios
js

```
const queries = queryCache.findAll(queryKey)
```

Options

- **queryKey?: QueryKey**: Query Keys
- **filters?: QueryFilters**: Query Filters

Returns

- **Query[]**
  - Query instances from the cache

queryCache.subscribe

The **subscribe** method can be used to subscribe to the query cache as a whole and be informed of safe/known updates to the cache like query states changing or queries being updated, added or removed
js

```
const callback = event => {
  console.log(event.type, event.query)
}

const unsubscribe = queryCache.subscribe(callback)
```

Options

- **callback: (event: QueryCacheNotifyEvent) => void**
  - This function will be called with the query cache any time it is updated via its tracked update mechanisms (eg, **query.setState**, **queryClient.removeQueries**, etc). Out of scope mutations to the cache are not encouraged and will not fire subscription callbacks

Returns

- **unsubscribe: Function => void**
  - This function will unsubscribe the callback from the query cache.

queryCache.clear

The **clear** method can be used to clear the cache entirely and start fresh.
js

```
queryCache.clear()
```

MutationCache

The **MutationCache** is the storage for mutations.
Normally, you will not interact with the MutationCache directly and instead use the QueryClient.

js

```
import { MutationCache } from 'react-query'

const mutationCache = new MutationCache({
  onError: error => {
    console.log(error)
  },
  onSuccess: data => {
    console.log(data)
  },
})
```

Its available methods are:

- **getAll**
- **subscribe**
- **clear**

Options

- **onError?: (error: unknown, variables: unknown, context: unknown, mutation: Mutation) => void**
  - Optional
  - This function will be called if some mutation encounters an error.
- **onSuccess?: (data: unknown, variables: unknown, context: unknown, mutation: Mutation) => void**
  - Optional
  - This function will be called if some mutation is successful.
- **onMutate?: (variables: unknown, mutation: Mutation) => void**
  - Optional
  - This function will be called before some mutation executes.

Global callbacks

The **onError**, **onSuccess** and **onMutate** callbacks on the MutationCache can be used to handle these events on a global level. They are different to **defaultOptions** provided to the QueryClient because:

- **defaultOptions** can be overridden by each Mutation - the global callbacks will **always** be called.
- **onMutate** does not allow returning a context value.

mutationCache.getAll

getAll returns all mutations within the cache.
Note: This is not typically needed for most applications, but can come in handy when needing more information about a mutation in rare scenarios

js

```
const mutations = mutationCache.getAll()
```

Returns

- **Mutation[]**
  - Mutation instances from the cache

mutationCache.subscribe

The **subscribe** method can be used to subscribe to the mutation cache as a whole and be informed of safe/known updates to the cache like mutation states changing or mutations being updated, added or removed.

js

```
const callback = mutation => {
  console.log(mutation)
}

const unsubscribe = mutationCache.subscribe(callback)
```

Options

- **callback: (mutation?: Mutation) => void**
  - This function will be called with the mutation cache any time it is updated.

Returns

- **unsubscribe: Function => void**
  - This function will unsubscribe the callback from the mutation cache.

mutationCache.clear

The **clear** method can be used to clear the cache entirely and start fresh.

js

```
mutationCache.clear()
```

QueryObserver

QueryObserver

The **QueryObserver** can be used to observe and switch between queries.

Js

```
const observer = new QueryObserver(queryClient, { queryKey: 'posts' })

const unsubscribe = observer.subscribe(result => {
  console.log(result)
  unsubscribe()
})
```

Options

The options for the **QueryObserver** are exactly the same as those of **useQuery**.

InfiniteQueryObserver

InfiniteQueryObserver

The **InfiniteQueryObserver** can be used to observe and switch between infinite queries.
js

```
const observer = new InfiniteQueryObserver(queryClient, {
  queryKey: 'posts',
  queryFn: fetchPosts,
  getNextPageParam: (lastPage, allPages) => lastPage.nextCursor,
  getPreviousPageParam: (firstPage, allPages) => firstPage.prevCursor,
})

const unsubscribe = observer.subscribe(result => {
  console.log(result)
  unsubscribe()
})
```

Options
The options for the **InfiniteQueryObserver** are exactly the same as those of **useInfiniteQuery**.

QueriesObserver

QueriesObserver

The **QueriesObserver** can be used to observe multiple queries.
js

```
const observer = new QueriesObserver(queryClient, [
  { queryKey: ['post', 1], queryFn: fetchPost },
  { queryKey: ['post', 2], queryFn: fetchPost },
])

const unsubscribe = observer.subscribe(result => {
  console.log(result)
  unsubscribe()
})
```

Options
The options for the **QueriesObserver** are exactly the same as those of **useQueries**.

QueryErrorResetBoundary

When using suspense or useErrorBoundaries in your queries, you need a way to let queries know that you want to try again when re-rendering after some error occurred. With the QueryErrorResetBoundary component you can reset any query errors within the boundaries of the component.

js

```
import { QueryErrorResetBoundary } from 'react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App: React.FC = () => (
  <QueryErrorResetBoundary>
    {({ reset }) => (
      <ErrorBoundary
        onReset={reset}
        fallbackRender={({ resetErrorBoundary }) => (
          <div>
            There was an error!
            <Button onClick={() => resetErrorBoundary()}>Try again</Button>
          </div>
        )}
      >
        <Page />
      </ErrorBoundary>
    )}
  </QueryErrorResetBoundary>
)
```

useQueryErrorResetBoundary

This hook will reset any query errors within the closest **QueryErrorResetBoundary**. If there is no boundary defined it will reset them globally:

js

```
import { useQueryErrorResetBoundary } from 'react-query'
import { ErrorBoundary } from 'react-error-boundary'

const App: React.FC = () => {
  const { reset } = useQueryErrorResetBoundary()
  return (
    <ErrorBoundary
      onReset={reset}
      fallbackRender={({ resetErrorBoundary }) => (
        <div>
          There was an error!
          <Button onClick={() => resetErrorBoundary()}>Try again</Button>
        </div>
      )}
    >
      <Page />
    </ErrorBoundary>
  )
}
```

FocusManager

The **FocusManager** manages the focus state within React Query.
It can be used to change the default event listeners or to manually change the focus state.
Its available methods are:

- **setEventListener**
- **setFocused**
- **isFocused**

focusManager.setEventListener

setEventListener can be used to set a custom event listener:
js

```
import { focusManager } from 'react-query'

focusManager.setEventListener(handleFocus => {
  // Listen to visibilitychange and focus
  if (typeof window !== 'undefined' && window.addEventListener) {
    window.addEventListener('visibilitychange', handleFocus, false)
    window.addEventListener('focus', handleFocus, false)
  }

  return () => {
    // Be sure to unsubscribe if a new handler is set
    window.removeEventListener('visibilitychange', handleFocus)
    window.removeEventListener('focus', handleFocus)
  }
})
```

focusManager.setFocused

setFocused can be used to manually set the focus state. Set **undefined** to fallback to the default focus check.
js

```
import { focusManager } from 'react-query'

// Set focused
focusManager.setFocused(true)

// Set unfocused
focusManager.setFocused(false)

// Fallback to the default focus check
focusManager.setFocused(undefined)
```

Options

- **focused: boolean | undefined**

focusManager.isFocused

isFocused can be used to get the current focus state.
js

```
const isFocused = focusManager.isFocused()
```

OnlineManager

The **OnlineManager** manages the online state within React Query.
It can be used to change the default event listeners or to manually change the online state.
Its available methods are:

- **setEventListener**
- **setOnline**
- **isOnline**

onlineManager.setEventListener

setEventListener can be used to set a custom event listener:

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

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

onlineManager.setOnline

setOnline can be used to manually set the online state. Set **undefined** to fallback to the default online check.

```
import { onlineManager } from 'react-query'

// Set to online
onlineManager.setOnline(true)

// Set to offline
onlineManager.setOnline(false)

// Fallback to the default online check
onlineManager.setOnline(undefined)
```

Options

- **online: boolean | undefined**

onlineManager.isOnline

isOnline can be used to get the current online state.

```
const isOnline = onlineManager.isOnline()
```

setLogger

setLogger

setLogger is an optional function that allows you to replace the default **logger** used by React Query to log errors. By default, the **window.console** object is used. If no global **console** object is found in the environment, nothing will be logged.
Examples:
js

```
import { setLogger } from 'react-query'
import { printLog, printWarn, printError } from 'custom-logger'

// Custom logger
setLogger({
  log: printLog,
  warn: printWarn,
  error: printError,
})

// Sentry logger
setLogger({
  log: message => {
    Sentry.captureMessage(message)
  },
  warn: message => {
    Sentry.captureMessage(message)
  },
  error: error => {
    Sentry.captureException(error)
  },
})

// Winston logger
setLogger(winston.createLogger())
```

Options

- **logger: Logger**
  - Must implement the **log**, **warn**, and **error** methods.

hydration

dehydrate

dehydrate creates a frozen representation of a **cache** that can later be hydrated with **Hydrate**, **useHydrate**, or **hydrate**. This is useful for passing prefetched queries from server to client or persisting queries to localStorage or other persistent locations. It only includes currently successful queries by default.

```
import { dehydrate } from 'react-query'

const dehydratedState = dehydrate(queryClient, {
  shouldDehydrateQuery,
})
```

*Note: Since version **3.22.0** hydration utilities moved into to core. If you using lower version your should import **dehydrate** from **react-query/hydration***

Options

- **client: QueryClient**
  - **Required**
  - The **queryClient** that should be dehydrated
- **options: DehydrateOptions**
  - Optional
  - **dehydrateMutations: boolean**
    - Optional
    - Whether or not to dehydrate mutations.
  - **dehydrateQueries: boolean**
    - Optional
    - Whether or not to dehydrate queries.
  - **shouldDehydrateMutation: (mutation: Mutation) => boolean**
    - Optional
    - This function is called for each mutation in the cache
    - Return **true** to include this mutation in dehydration, or **false** otherwise
    - The default version only includes paused mutations
  - **shouldDehydrateQuery: (query: Query) => boolean**
    - Optional
    - This function is called for each query in the cache
    - Return **true** to include this query in dehydration, or **false** otherwise
    - The default version only includes successful queries, do **shouldDehydrateQuery: () => true** to include all queries

Returns

- **dehydratedState: DehydratedState**
  - This includes everything that is needed to hydrate the **queryClient** at a later point
  - You **should not** rely on the exact format of this response, it is not part of the public API and can change at any time
  - This result is not in serialized form, you need to do that yourself if desired

limitations

The hydration API requires values to be JSON serializable. If you need to dehydrate values that are not automatically serializable to JSON (like **Error** or **undefined**), you have to serialize them for yourself. Since only successful queries are included per default, to also include **Errors**, you have to provide **shouldDehydrateQuery**, e.g.:

```
// server
const state = dehydrate(client, { shouldDehydrateQuery: () => true }) // to also include Errors
const serializedState = mySerialize(state) // transform Error instances to objects
```

```
// client
const state = myDeserialize(serializedState)  // transform objects back to Error instances
hydrate(client, state)
```

hydrate

hydrate adds a previously dehydrated state into a **cache**. If the queries included in dehydration already exist in the queryCache, **hydrate** does not overwrite them.

js

```
import { hydrate } from 'react-query'

hydrate(queryClient, dehydratedState, options)
```

*Note: Since version **3.22.0** hydration utilities moved into to core. If you using lower version your should import **hydrate** from **react-query/hydration***

Options

- **client: QueryClient**
  - **Required**
  - The **queryClient** to hydrate the state into
- **dehydratedState: DehydratedState**
  - **Required**
  - The state to hydrate into the client
- **options: HydrateOptions**
  - Optional
  - **defaultOptions: DefaultOptions**
    - Optional
    - **mutations: MutationOptions** The default mutation options to use for the hydrated mutations.
    - **queries: QueryOptions** The default query options to use for the hydrated queries.

useHydrate

useHydrate adds a previously dehydrated state into the **queryClient** that would be returned by **useQueryClient()**. If the client already contains data, the new queries will be intelligently merged based on update timestamp.

jsx

```
import { useHydrate } from 'react-query'

useHydrate(dehydratedState, options)
```

*Note: Since version **3.22.0** hydration utilities moved into to core. If you using lower version your should import **useHydrate** from **react-query/hydration***

Options

- **dehydratedState: DehydratedState**
  - **Required**
  - The state to hydrate
- **options: HydrateOptions**
  - Optional
  - **defaultOptions: QueryOptions**
    - The default query options to use for the hydrated queries.

Hydrate

Hydrate wraps **useHydrate** into component. Can be useful when you need hydrate in class component or need hydrate on same level where **QueryClientProvider** rendered.
js

```
import { Hydrate } from 'react-query'

function App() {
  return <Hydrate state={dehydratedState}>...</Hydrate>
}
```

*Note: Since version **3.22.0** hydration utilities moved into to core. If you using lower version your should import **Hydrate** from **react-query/hydration***

Options

- **state: DehydratedState**
  - The state to hydrate
- **options: HydrateOptions**
  - Optional
  - **defaultOptions: QueryOptions**
    - The default query options to use for the hydrated queries.