Status Checks in React Query

27.03.2021 — react, React Query, JavaScript, TypeScript — 3 min read

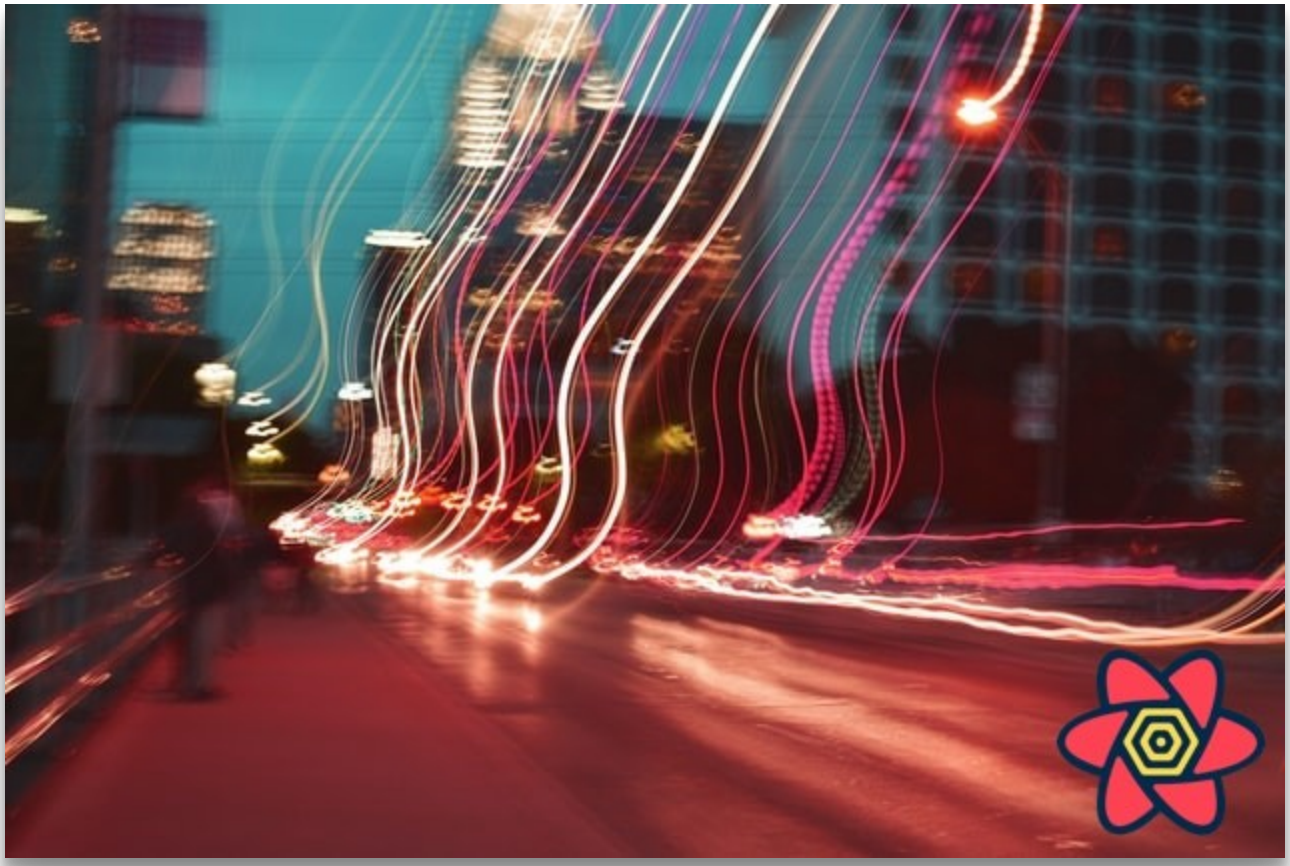

Photo by Kyndall Ramirez

Last Update: 23.04.2022

One advantage of React Query is the easy access to status fields of the query. You instantly know if your query is loading or if it's erroneous. For this, the library exposes a bunch of boolean flags, which are mostly derived from the internal state machine. Looking at the types, your query can be in one of the following states:

- `success` : Your query was successful, and you have `data` for it
- `error` : Your query did not work, and an `error` is set
- `loading` : Your query has no data and is currently `loading` for the first time
- `idle` : Your query has never run because it's not `enabled`

Update: In v4 of React Query, the idle state has been removed. The `loading` state just means "you have no data yet".

Note that the `isFetching` flag is *not* part of the internal state machine - it is an additional flag that will be true whenever a request is in-flight. You can be fetching and success, you can be fetching and error - but you cannot be loading and success at the same time. The state machine makes sure of that.

Update: In v4, the `isFetching` flag is derived from a secondary `fetchStatus` - just like the new `isPaused` flag. You can read more about this in #13: Offline React Query.

The standard example

The `idle` state is mostly left out, because it's an edge case for disabled queries. So most examples look something like this:

standard-example

JSX Copy

```jsx
const todos = useTodos()

if (todos.isLoading) {
  return 'Loading...'
}
if (todos.error) {
  return 'An error has occurred: ' + todos.error.message
}

return <div>{todos.data.map(renderTodo)}</div>
```

Here, we check for loading and error first, and then display our data. This is probably fine for some use-cases, but not for others. Many data fetching solutions, especially hand-crafted ones, have no refetch mechanism, or only refetch on explicit user interactions.

But React Query does.

It refetches quite aggressively per default, and does so without the user actively requesting a refetch. The concepts of `refetchOnMount`, `refetchOnWindowFocus` and `refetchOnReconnect` are great for keeping your data accurate, but they might cause a confusing ux if such an automatic background refetch fails.

Background errors

In many situations, if a background refetch fails, it could be silently ignored. But the code above does not do that. Let's look at two examples:

- The user opens a page, and the initial query loads successfully. They are working on the page for some time, then switch browser tabs to check emails. They come back some minutes later, and React Query will do a background refetch. Now that fetch fails.
- Our user is on page with a list view, and they click on one item to drill down to the detail view. This works fine, so they go back to the list view. Once they go to the detail view again, they will see data from the cache. This is great - except if the background refetch fails.

In both situations, our query will be in the following state:

JSON Copy

```json
{
  "status": "error",
```

```
  "error": { "message": "Something went wrong" },
  "data": [{ ... }]
}
```

As you can see, we will have *both* an error *and* the stale data available. This is what makes React Query great - it embraces the stale-while-revalidate caching mechanism, which means it will always give you data if it exists, even if it's stale.

Now it's up to us to decide what we display. Is it important to show the error? Is it enough to show the stale data only, if we have any? Should we show both, maybe with a little *background error* indicator?

There is no clear answer to this question - it depends on your exact use-case. However, given the two above examples, I think it would be a somewhat confusing user experience if data would be replaced with an error screen.

This is even more relevant when we take into account that React Query will retry failed queries three times per default with exponential backoff, so it might take a couple of seconds until the stale data is replaced with the error screen. If you also have no background fetching indicator, this can be really perplexing.

This is why I usually check for data-availability first:

data-first

JSX Copy

```jsx
const todos = useTodos()

if (todos.data) {
  return <div>{todos.data.map(renderTodo)}</div>
}
if (todos.error) {
  return 'An error has occurred: ' + todos.error.message
}

return 'Loading...'
```

Again, there is no clear principle of what is right, as it is highly dependent on the use-case. Everyone should be aware of the consequences that aggressive refetching has, and we have to structure our code accordingly rather than strictly following the simple todo-examples 😉.

Special thanks go to Niek Bosch who first highlighted to me why this pattern of status checking can be harmful in some situations.

Feel free to reach out to me on twitter if you have any questions, or just leave a comment below ⬇️

Like the monospace font in the code blocks?
Check out monolisa.dev

Theme by LekoArts