Automatic Query Invalidation after Mutations

25.05.2024 — ReactJs, React Query, TypeScript, JavaScript — 6 min read

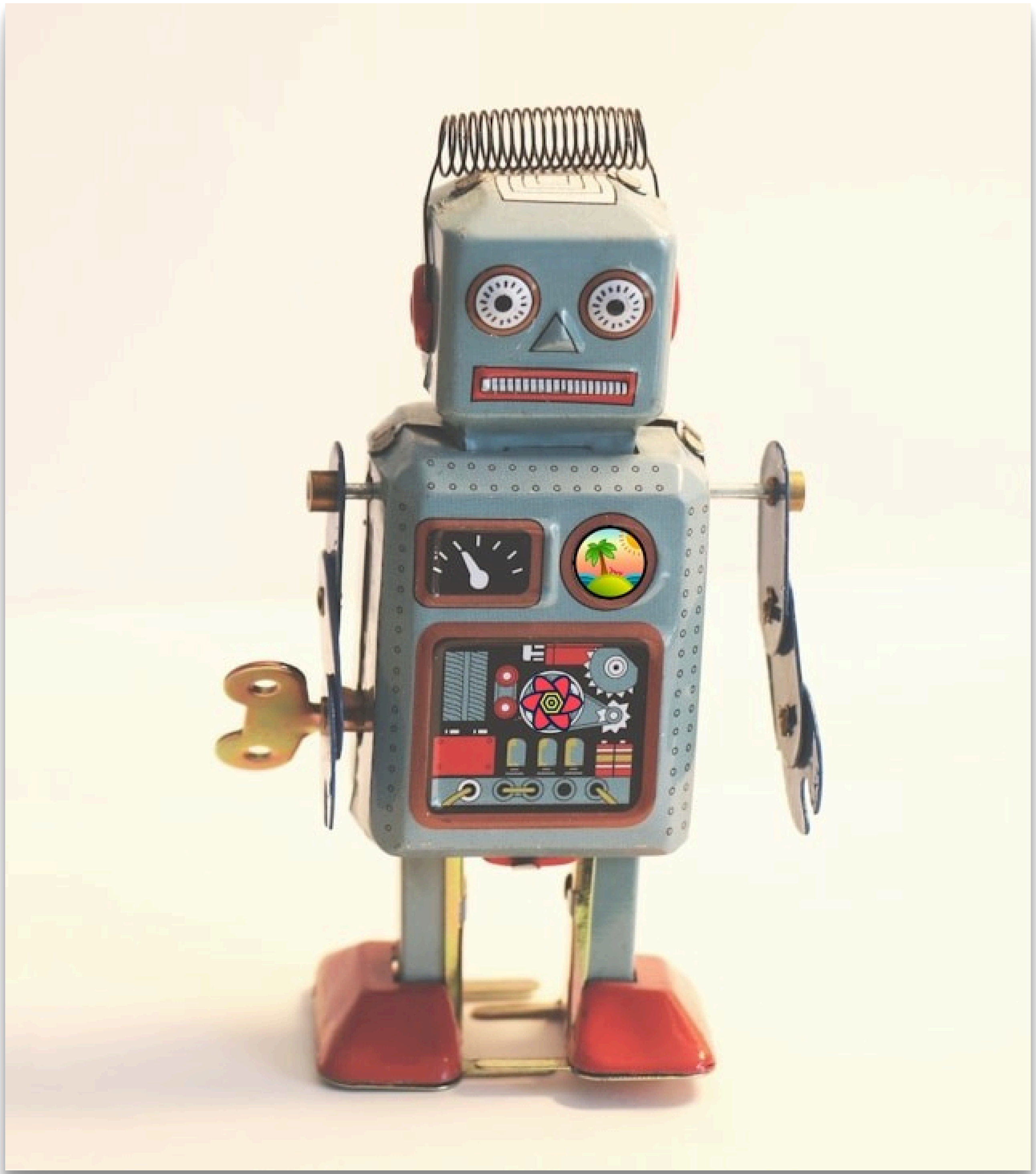

Photo by Rock'n Roll Monkey

Add translation

Queries and Mutations are two sides of the same coin. A **`Query`** defines an asynchronous resource for reading, which often comes from data fetching. A **`Mutation`** on the other hand is an action to update such a resource.

When a Mutation finishes, it very likely affects Queries. For example, updating an `issue` will likely affect the list of `issues`. So it might be a bit surprising that React Query does not link Mutations to Queries at all.

The reason behind this is quite simple: React Query is totally **un-opinionated** about how you manage your resources, and not everyone likes re-fetching after a Mutation. There are cases where the Mutation returns updated data, which we'd want to then put into the cache manually to avoid another network roundtrip.

There are also many different ways of how you'd want to do invalidation:

- Do you invalidate in `onSuccess` or `onSettled` ?
  The former will only be invoked when the Mutation succeeded, while the latter will also run in case of errors.
- Do you want to `await` invalidations?
  Awaiting an invalidation will result in the Mutation staying in `pending` state until the refetch has finished. This can be a good thing, for example if you'd want your form to stay disabled until then, but it might also be not what you want in case you want to navigate from a detail screen to an overview page as soon as possible.

Since there isn't a one-size-fits-all solution, React Query provides nothing out of the box. However, it's not at all difficult to implement automatic invalidation the way you want them to behave in React Query thanks to the global cache callbacks.

The Global Cache Callbacks

Mutations have callbacks - `onSuccess` , `onError` and `onSettled` , which you have to define on each separate `useMutation` . Additionally, the same callbacks exist on the `MutationCache` . Since there is only one `MutationCache` for our application, those callbacks are "global" - they are invoked for *every* Mutation.

It's not quite obvious how to create a `MutationCache` with callbacks, because in most examples, the `MutationCache` is implicitly created for us when we create the `QueryClient` . However, we can also create the cache itself manually and provide callbacks to it:

create-MutationCache

TS Copy

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

const queryClient = new QueryClient({
  mutationCache: new MutationCache({
    onSuccess,
    onError,
    onSettled,
  }),
})
```

The callbacks get the same arguments as the ones on `useMutation`, except that they will also get the Mutation instance as last parameter. And just like the usual callbacks, returned Promises will be awaited.

So how can the global callback help us with automatic invalidation? Well - we can just call `queryClient.invalidateQueries` inside the global callback:

automatic-invalidation

TS

Copy

```ts
const queryClient = new QueryClient({
  mutationCache: new MutationCache({
    onSuccess: () => {
      queryClient.invalidateQueries()
    },
  }),
})
```

With just 5 lines of code, we get a similar behaviour to what frameworks like Remix (sorry, React-Router) are doing as well: Invalidate everything after every submission. Shout out to Alex for showing me this path:

Alex / KATT 🐱
@alexdotjs

I just invalidate everything on every mutation

https://trpc.io/docs/client/re...

- Jan 8, 2024

But isn't that excessive ?

Maybe, maybe not. It depends. Again, that's why it isn't built in, because there are too many different ways to go about it. One thing we have to clarify here is that an invalidation doesn't always equate to a **refetch**.

Invalidation merely refetches all **active** Queries that it matches, and marks the rest as `stale`, so that they get refetched when they are used the next time.

This is usually a good trade-off. Consider having an Issue List with filters. Since each filter should be part of the QueryKey, we'll get multiple Queries in the cache. However, I'm only ever viewing one of those Queries at the same time. Refetching them all would lead to lots of unnecessary requests, and there's no guarantee that I will ever go back to a list with one of those filters.

So invalidation only refetches what I currently see on the screen (active Queries) to get an up-to-date view, and everything else will be refetched if we ever need them again.

Tying invalidation to specific Queries

Okay, hold on. What about fine-grained revalidation? Why would we invalidate the `profile` data when we add an `issue` to our list? That barely makes sense ...

Again, a trade-off. The code is as simple as it gets, and I would prefer fetching some data more often than strictly necessary over missing a refetch. Fine-grained revalidation is nice if you know exactly what you need to refetch, and that you'll never need to extend those matches.

In the past, we've often done fine-grained revalidation, just to find out that we'd need to add another resource into the mix later which doesn't fit the used invalidation pattern. At that point, we had to go through all mutation callbacks to see if that resource needed to be refetched as well. That's cumbersome and error-prone.

On top of that, we often use a medium-sized `staleTime` of ~2 minutes for most our Queries. So the impact of invalidating after an unrelated user interaction is negligible.

Of course, you can make your logic more involved to make your revalidation smarter. Here are some techniques I've used in the past:

Tie it to the `mutationKey`

MutationKey and QueryKey have nothing in common, and the one for Mutations is also optional. You can tie them together if you want by using the MutationKey to specify which Queries should be invalidated:

mutationKey

TS

Copy

```ts
const queryClient = new QueryClient({
  mutationCache: new MutationCache({
    onSuccess: (_data, _variables, _context, mutation) ⇒ {
      queryClient.invalidateQueries({
        queryKey: mutation.options.mutationKey,
      })
    },
  }),
})
```

Then, you can give your Mutation a `mutationKey: ['issues']` to invalidate everything `issue` related only. And if you have a Mutation without a key, it would still invalidate everything. Nice.

Exclude Queries depending on `staleTime`

I often mark Queries as "static" by giving them `staleTime:Infinity`. If we don't want those Queries to be invalidated, we can look at the `staleTime` setting of a Query and exclude those via the `predicate` filter:

nonStaticQueries

```ts
const queryClient = new QueryClient({
  mutationCache: new MutationCache({
    onSuccess: (_data, _variables, _context, mutation) => {
      const nonStaticQueries = (query: Query) => {
        const defaultStaleTime =
          queryClient.getQueryDefaults(query.queryKey).staleTime ?? 0
        const staleTimes = query.observers
          .map((observer) => observer.options.staleTime)
          .filter((staleTime) => staleTime !== undefined)

        const staleTime =
          query.getObserversCount() > 0
            ? Math.min(...staleTimes)
            : defaultStaleTime

        return staleTime !== Number.POSITIVE_INFINITY
      }

      queryClient.invalidateQueries({
        queryKey: mutation.options.mutationKey,
        predicate: nonStaticQueries,
      })
    },
  }),
})
```

Finding out the actual `staleTime` for a Query is not that trivial, because `staleTime` is an observer level property. But it's doable, and we can also combine the `predicate` filter with other filters like `queryKey`. Neat.

Use the `meta` option

We can use `meta` to store arbitrary, static information about a Mutation. As an example, we can add an `invalidates` field to give "tags" to our mutation. These tags can then be used to fuzzily match Queries we'd want to invalidate:

the-meta-option

JS

Copy

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

const queryClient = new QueryClient({
  mutationCache: new MutationCache({
    onSuccess: (_data, _variables, _context, mutation) => {
      queryClient.invalidateQueries({
        predicate: (query) =>
          // invalidate all matching tags at once
          // or everything if no meta is provided
          mutation.meta?.invalidates?.some((queryKey) =>
            matchQuery({ queryKey }, query)
          ) ?? true,
      })
    },
  }),
})

// usage:
useMutation({
  mutationFn: updateLabel,
  meta: {
    invalidates: [['issues'], ['labels']],
  },
})
```

Here, we still use the `predicate` function to get a single call to `queryClient.invalidateQueries`. But inside of it, we do fuzzy matching with `matchQuery` - a function you can import from React Query. It's the same function that gets used internally when passing a single `queryKey` as a filter, but now, we can do it with multiple keys.

This pattern is probably only slightly better than just having `onSuccess` callbacks on `useMutation` itself, but at least we don't need to bring in the QueryClient with `useQueryClient` every time. Also, if we combine this with invalidating everything per default, this will give us a good way to opt-out of that behaviour.

The meta option in TypeScript

Generally, `meta` is typed as `Record<string, unknown>`, but we can tweak this with module augmentation:

TS

Copy

```ts
declare module '@tanstack/react-query' {
  interface Register {
    mutationMeta: {
      invalidates?: Array<QueryKey>
    }
  }
}
```

You can read more about typing meta in the docs.

To Await or not to Await

In all the examples shown above, we are never `awaiting` an invalidation, and that's fine if you want your mutations to finish as fast as possible. One specific situation that I have come across a lot is wanting to invalidate everything, but have the Mutation stay pending until one important refetch is done. For example, I might want label specific Queries to be awaited after updating a label, but I wouldn't want to wait until everything is done refetching.

We can build this into our `meta` solution by extending how that structure is defined, for example:

meta-awaits

TS

Copy

```ts
useMutation({
  mutationFn: updateLabel,
  meta: {
    invalidates: 'all',
    awaits: ['labels'],
  },
})
```

Or, we can take advantage of the fact that callbacks on the MutationCache run **before** callbacks on `useMutation`. If we have our global callback set-up to invalidate everything, we can still add a local callback that just `awaits` what we want it to:

local-onSuccess

TS

Copy

```ts
const queryClient = new QueryClient({
  mutationCache: new MutationCache({
    onSuccess: () => {
      queryClient.invalidateQueries()
    },
  }),
})

useMutation({
  mutationFn: updateLabel,
  onSuccess: () => {
    // returning the Promise to await it
    return queryClient.invalidateQueries(
      { queryKey: ['labels'] },
      { cancelRefetch: false }
    )
  },
})
```

Here's what's happening:

- First, the global callback runs and invalidates all Queries, but we since we neither `await` nor `return` anything, this is a "fire-and-forget" invalidation.
- Then, our local callback will run immediately after that, where we will create a Promise for invalidating the `['labels']` only. Since we are returning that Promise, the Mutation will stay pending until `['labels']` are refetched.

cancelRefetch

Note that we are passing `cancelRefetch: false` to the manual `invalidateQueries` call. This flag defaults to `true`, because we'd usually want imperative refetch calls to take precedence and cancel currently running ones to guarantee up-to-date data afterwards.

But here, we want the opposite: Since our global callback has already invalidated everything, including the Query we want to `await`, we just use `invalidateQueries` to "pick up" the already in-flight Promise and return it.

If we wouldn't do that, we'd see another request for our `['labels']` Query.

I think this shows that it's not a lot of code to add an abstraction that you're comfortable with for automatic invalidation. Just keep in mind that every abstraction has a cost: It's a new API that needs to be learned, understood and applied properly.

I hope by showing all these possibilities, it's a bit clearer why we have nothing built into React Query. Finding an API that is flexible enough to cover all cases without being bloated is not an easy thing to do. For this, I prefer to give *you* the tools to build this in user-land.

That's it for today. 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