React 19 and Suspense - A Drama in 3 Acts

16.06.2024 — ReactJs, React Query, Suspense, JavaScript — 7 min read

Photo by Jr Korpa

No translations available. Add translation

That was quite a roller-coaster last week 🎢. Some things unravelled, some things went down, and in the middle of it: React Summit, the biggest React conference in the world.

Let me try to break down what happened, in hopefully the right order, and what we can all learn from it. To do that, we have to go back to April this year:

First Act: React 19 Release Candidate

The 25th of April was a great day: React announced the React 19 RC - a release specifically for collecting feedback and preparing libraries for the next major version of React.

I was really excited - there are so many good things in that release. From the new hooks to the `use` operator, from server actions to the `ref` prop. From better hydration errors to cleanup functions for refs. From better `useRef` types to `useLayoutEffect` finally not warning anymore on the server. And of course: The experimental React Compiler. 🚀

This release is packed with goodies, and I was excited to upgrade React Query to see if there were any problems. I was quite busy at the time with work and finishing the 🔮 query.gg course, but about a month later, we released v5.39.0, which is compatible with React 19:

Dominik 🔮
@TkDodo

React Query 🤝 React 19

github.com/TanStack/query...

Someone please try out the react compiler on an example and report back. I've enabled the eslint-plugin-react-compiler and it didn't report anything suspicious. I hope that means we play by the rules 😂

- May 25, 2024

There weren't really any issues to dig into, so I thought this release was on track to become the best React release since hooks were introduced. That is, until we noticed something weird with suspense.

Second Act: Uncovering Suspense

Full disclosure upfront: I wasn't the first to discover this. Shout out to Gabriel Valfridsson who (to the best of my knowledge) first spotted the new behaviour one day after the RC announcement:

Gabriel Valfridsson
@GabbeV_

A ton of great changes! 🥳

However this change probably deserve a bigger disclaimer.
https://github.com/facebook/react...

Code using suspense with libraries like react-query will get waterfalls where loading previously happened in parallel.
https://codesandbox.io/p/devbox/react...
https://codesandbox.io/p/devbox/react...

- Apr 26, 2024

It's funny because I saw the tweet, and even commented on it, but didn't think too much of it at the time. As I said, I was quite busy and planned to look into React 19 later.

So after the React 19 upgrade in React Query itself, I continued working on the suspense lesson of the course. We have one example in there where we're showing how to reveal content at the same time, but still have all requests fetch in parallel. As shown in the react docs, we can achieve this by putting both components as siblings into the same suspense boundary. The example looks roughly like this:

suspense-with-two-children

JSX Copy

```jsx
export default function App() {
  return (
    <Suspense fallback={<p>...</p>}>
      <RepoData name="tanstack/query" />
      <RepoData name="tanstack/table" />
    </Suspense>
  )
}
```

The way this works is that React sees that the first child will suspend, so it knows that it has to show the `fallback`. However, it still continues to render other siblings in case they will also suspend, so that it can "collect" all promises.

This is a pretty great feature because it means if each sibling triggers anything async, we can compose our components in a way that they will still fetch in parallel while not triggering a 🍿 "popcorn UI" 🍿 where multiple parts of the screen pop in one after the other.

A more complete example might look something like this:

app-with-suspense

JSX Copy

```jsx
export default function App() {
  return (
    <Suspense fallback={<p>...</p>}>
      <Header />
      <Navbar />
      <main>
        <Content />
      </main>
      <Footer />
    </Suspense>
  )
}
```

Some or all of those components can initiate critical data fetching, and we'll get our UI displayed at once when those fetches have resolved.

Another advantage is that we can add fetches later without having to think about how pending states will be handled. The `<Footer />` component might not fetch data now, but if we add it later, it will just work. And if we deem data as non-critical, we can always wrap our component in it's own suspense boundary:

nested-suspense

JSX Copy

```jsx
export default function App() {
  return (
    <Suspense fallback={<p>...</p>}>
      <Header />
      <Navbar />
      <main>
        <Content />
      </main>
      <Suspense fallback={<p>...</p>}>
        <Footer />
      </Suspense>
    </Suspense>
  )
```

```
14 }
```

Now fetching data in our footer will not block rendering the main content. This is pretty powerful and aligned with how React favors component composition above anything else.

I vaguely remembered seeing something on twitter about suspense having a different behaviour in React 19, so just to be sure, I wanted to try out what we have in the course with the new RC release. And, to my surprise, it behaved completely differently: Instead of fetching data for both siblings in parallel, it now created a **waterfall**. 💦

I was so surprised by this behaviour that I did the only thing I could think of at that moment - I jumped on twitter and tagged some react core team members:

Dominik 🔮
@TkDodo

Am I imagining things or is there a difference between React 18 and 19 in terms of how Suspense handles parallel fetching? In 18, there is a "per component" split, so putting two components into the same Suspense Boundary, where each was doing a fetch, was still firing them in parallel:

This fires two queries, in parallel, waits until both are resolved and then shows the whole sub-tree.

In React 19, as far as I can see, the queries run in a waterfall now. I think I remember @rickhanlonii mentioning something like this but I can't find any evidence now.

/cc @acdlite @dan_abramov2

- Jun 11, 2024

Needless to say, this tweet took off and started a somewhat heated twitter discussion. We soon found out that this was not a bug, but an intentional change, which led to quite some outrage.

Why would they do that?

There are of course reasons why this change was made, and, oddly enough, they are meant as a performance improvement for some situations. Continuing to render siblings of a component that has already suspended is not for free, and it will block showing the fallback. Consider the following example:

expensive-sibling

JSX

Copy

```jsx
export default function App() {
  return (
    <Suspense fallback={<p>...</p>}>
      <SuspendingComponent />
      <ExpensiveComponent />
    </Suspense>
  )
}
```

Let's assume that `<ExpensiveComponent />` takes some time to render, e.g. because it is a huge sub-tree, but does not suspend itself. Now when react renders this tree, it will see that `<SuspendingComponent />` suspends, so the only thing it will need to display eventually is the suspense fallback. However, it can only do that when rendering has finished, so it has to wait until `<ExpensiveComponent />` is done rendering. Even more - the render result of `<ExpensiveComponent />` will be thrown away, because the fallback has to be displayed.

When we think about it this way - pre-rending the siblings of a suspended component is pure overhead, as it will never amount to a meaningful output. So React 19 removed that to get instant loading states.

Of course, if you suspend instantly, you can't see that the siblings will also suspend, so if those siblings were to initiate data fetches (e.g. with `useSuspenseQuery`), they will now waterfall. And that's where the controversy comes in.

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

Having a component initiate a fetch is usually called **fetch-on-render**. It's the approach most of us likely use on a daily basis, but it's not the best thing you can do. Even when siblings inside the same suspense boundary are pre-rendered in parallel, you would not be able to avoid the waterfall if you have two `useSuspenseQuery` calls within the same react component, or if you had a parent-child relationship between components.

That is why the recommended approach by the react team is to initiate fetches earlier, e.g. in route loaders or in server components, and to have suspense only consume the resource rather than initiate the promise itself. This is usually called **render-as-you-fetch**.

For example, with TanStack Router and TanStack Query, the example could look like this:

prefetch-in-route-loader

TSX

Copy

```tsx
export const Route = createFileRoute('/')({
```

```
  loader: ({ context: { queryClient } }) => {
    queryClient.ensureQueryData(repoOptions('tanstack/query'))
    queryClient.ensureQueryData(repoOptions('tanstack/table'))
  },
  component: () => (
    <Suspense fallback={<p>...</p>}>
      <RepoData name="tanstack/query" />
      <RepoData name="tanstack/table" />
    </Suspense>
  ),
})
```

Here, the route loader makes sure that the fetches for both queries are initiated *before* the component is rendered. So when react starts to render the suspense children, it doesn't matter if it renders the second `RepoData` component or not, because it wouldn't trigger a fetch - it would just consume the already running promise. In this situation, React 19 would make our app slightly faster because it would have to do less work without any drawbacks.

Not everything is a fetch

Hoisting your data requirements is a good idea regardless of how suspense works, and I also recommend doing that. However, with the proposed React 19 changes, it becomes almost mandatory to do so.

Further, if learned anything from React Query, it's that not every async operation is a fetch. For example, using `React.lazy` for code-splitting would also mean that bundles are loaded in serial if your App looks like this:

react.lazy

JSX Copy

```
const Header = lazy(() => import('./Header.tsx'))
const Navbar = lazy(() => import('./Navbar.tsx'))
const Content = lazy(() => import('./Content.tsx'))
const Footer = lazy(() => import('./Footer.tsx'))

export default function App() {
  return (
    <Suspense fallback={<p>...</p>}>
      <Header />
      <Navbar />
      <main>
        <Content />
      </main>
      <Footer />
```

```
      </Suspense>
    )
}
```

Yes, you can technically preload dynamic imports as well, but making this required for good performance kind of defeats the purpose of react suspense and component composition, as the `App` component would need to know everything async that goes on in any of its children.

Third Act: Escalation and Delaying the Release

By now, a lot of people on the internet were surprised about and afraid of those changes. Screenshots were shared about how apps that fetched almost everything in parallel in 18 resulted in a total waterfall in 19. The developers behind the Poimandres open source developer collective, which maintains react-three-fiber, were a bit freaked out because a lot of what react-three-fiber is doing is based on async work and leverages how suspense works today. This went so far that even forking react was up for discussion if the change actually made it into v19.

By that time, I was already in Amsterdam for ReactSummit. We were talking about this change at the React Ecosystem Contributors Summit, where everyone was either surprised, concerned or frustrated. The React core team was doubling down, explaining how this change is the better tradeoff and raises the ceiling, how we should hoist data requirements anyways, and that official suspense support on the client was never released (which, even if true, everyone I know misunderstood).

Later that evening, I had the chance to talk to Sathya Gunasekaran, who worked on the react compiler and v19:

Dominik 🔮
@TkDodo

Had a great discussion with @_gsathya about React19, the suspense changes and the react compiler. I felt like my concerns

were truly heard and the feedback appreciated, so thank you for that 🙏

- Jun 13, 2024

He ensured me that the react team cares a lot about the community and that they likely underestimated how the change influences client side suspense interactions.

On the next day, the react team met and decided to hold the release:

sophie alpert
@AvatarSophiebits

good news re Suspense, just met w/ @rickhanlonii @en_JS @acdlite
* we care a lot about SPAs, team misjudged how many people rely on this today
* still recommend preloading but recognize not always practical
* we plan to hold the 19.0 release until we find a good fix

- Jun 14, 2024

It's very reassuring that the react team is open to feedback at this stage. Postponing a release that was already announced and presented at a conference is a big move - one that everyone involved really appreciated. I'll gladly work with the react team as best I can to find good compromise in that matter.

Learnings

There are a couple of things I learned from all of this. For one, trying out early releases before they come out as a final version is a very good idea, especially if the team is ready to take feedback and act on it. Kudos to the react team - I just really wish I would've given that feedback earlier.

The other thing that became obvious to me and other maintainers is that we need a better channel to communicate with the react team. The React 18 Working Group was probably the best thing we had in this regard, and this whole saga shows that having something similar for React 19 (and future react releases) would be great. Something like a permanent working group maybe?

Also, obvious but worth mentioning: shouting at each other on twitter is not helpful. I regret the part I took in any communication that wasn't calm and objective, and I really appreciate Sophie's way of communicating and handling things. 🙏

Like so many others before me have figured out: interactions in person are always so much better, and I'm looking forward to having more great conversations at conferences. 🎉

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. ⬇️