Cory Follow
Nov 14, 2021 · 6 min read · Listen

Have we forgotten how to think in React?

It's been a minute or two since "think in React" was something that was talked about. React has changed quite a bit since then. Is it even relevant anymore? Based on the conversations and articles being written, one might think it isn't.

I going to push back hard on that and remind everyone that the way we should think of our React applications **is** and **always has been** as a pure function of props and state. That is the core, fundamental model of React. If React ever departs from it, it will no longer be React.

This is important because thinking this way eliminates a ton of complexity. We seem to wrestle so much with things like useEffect, useMemo, useCallback, unnecessary renders, forms, and so many things it seems. Sometimes we come up with some pretty over the top "solutions" to these problems, but if we take a step back and remember that React expects us to think of our application — and each individual component — as a pure function of props and state, then we can begin to see our problems clearer. We know that if we give different inputs to a pure function we get different outputs. So we know if we give a different object reference as a prop to a component, it will rerender, even if that object has an equivalent shape[1]. Remembering that a change in props will result in a rerender will do more to prevent unnecessary renders than any of the various techniques or libraries out there are able to do on their own.

The same thinking is applicable to hooks, particularly the ones that take a dependency array argument. Think of that dependency array in the same way you would your component props. Your hook will re-run when any element in that array changes, the same way your component will re-render when any of its props changes. In this way, the same mental model applies. Like your components, your hooks are a pure[2] function of arguments[3], and will react similarly to how your components react when new values are "passed".

Perhaps one of the reasons we haven't been accustomed to thinking of hooks in this way is because it is common to write our hooks in line, right in the component. This makes them feel like they behave differently. Since everything we need is already in scope for our hook, it's tempting to think of the dependency array as a "run only when these values change" array.

```
const Welcome = ({ name }) => {
  useEffect(() => {
    logWhoSawThePage(name)
  // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []) // <-- we only want this to run on mount, right?

  return (
    <>
      <h1>Welcome {name}!</h1>
      <p>So glad you could stop by.</p>
    </>
  )
}
```

But that would be a mistake. It's not a list of things to watch, it's a dependency array. This is why I am 100% with Kyle Shevlin when he advocates to never use a bare react hook in a component. Always wrap them in a custom hook. There are many other reasons following this advice is a good idea, but even if this were the only one, I'd still recommend it. Moving all your react-hooks into a custom hook highlights all the things your hook relies on to function. Not just the things you think are the ones that matter to re-run.

```
const useLogWhoSawThePage = () => {
  useEffect(() => {
    logWhoSawThePage(name) //❌ `name` is not defined!
  // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []) // <-- we only want this to run on mount, right?
}
```

```
const Welcome = ({ name }) => {
  useLogWhoSawThePage() // 🧐

  return (
    <>
      <h1>Welcome {name}!</h1>
      <p>So glad you could stop by.</p>
    </>
  )
}
```

If we need to pass it in, we need to add it to our dependency array.

```
const useLogWhoSawThePage = (name) => {
  useEffect(() => {
    logWhoSawThePage(name) //✅ that's better
  }, [name]) // <-- 🤔
}
```

.

```
const Welcome = ({ name }) => {
  useLogWhoSawThePage(name) // 😆

  return (
    <>
      <h1>Welcome {name}!</h1>
      <p>So glad you could stop by.</p>
    </>
  )
}
```

“BUT WAIT!” I hear you saying. “We don’t *want* this to re-run when `name` changes, only on the first render!”

To that, I say, “Oh, really? Tell me why.”

“Go on, I’ll wait…”

Let me help you out if you’re stumped. Your initial thinking was that this is a page, and it would be silly to log that the same person saw this page multiple times, just because props change and the page re-renders.

I imagine most of you actually saw the problem quite early on and felt a little insulted that I thought you or anyone you work with would make this mistake, but such is the benefit and curse of a contrived example. You can clearly see the problem, so I can clearly point you to the proper solution. In the real world, the problem would be more obscured, but I can assure you, I’ve seen this category of fallacy — even almost this exact situation— happen shockingly often by some of the most experienced engineers, myself included. Not because it’s such a hard problem, but it’s just so damn easy to do it incorrectly when we are not thinking in the “React way”. The habit of always wrapping bare react-hook calls in a custom hook just creates an environment where thinking in the React way is much more natural.

The beauty of React is its small and consistent mental model. Over the past several years, in our efforts to get a firm grasp of new features added to React, we’ve taken to focusing on how those features are different from other features, particularly the ones they are meant to replace. This is perfectly fine as far as it goes, but in the process, we’ve forgotten to return to the first principles of React. Purity.

The principle of purity should inform our mental model of how React works across all its features. Purity naturally leads to encapsulation. It becomes easier to identify the lines to draw component boundaries around when we consider that components should be pure functions of state and props. A collection of UI and functionality are a good candidate for a component if we can encapsulate that collection in such a way that it is a pure function of state and props. Pure functions make composition — a central pillar of React — possible. Without it, we would not be able to compose simple pieces into complex applications.

A parting thought. If you are still learning React (and shouldn’t we all consider ourselves as learners?) I would very much recommend you try to error on the side of too many components than too few. Think of it as practice. The more we do it the better we get at it. And it has been my experience that having too few components tends to cause those components to become a tangled mess of multiple concepts and functionality all being intermingled together. Such a thing is very time-consuming to disentangle. Whereas components that are too fine-grained are very easy to unwrap. Anything can be taken too far, but in my experience, having too many components is not our collective problem.

Got a comment, or do you disagree? I’d love to talk with you about it. Comment on this post, or hit me up on Twitter. I’m @uniqname. I’m always happy to engage in good-natured discussions and honest debate.

1. The Records and Tuples proposal will change this sentence somewhat and, I think will radically change the game with programming in Javascript. You can think of Records and Tuples as objects and arrays that are compared by value rather than by reference. Therefore two Records with the same shape (keys and values) are strictly equal (`#{a: 5} === #{a: 5}`). Two Tuples with the same shape (types and values) are strictly equal (`#[‘a’, 5] === #[‘a’, 5]`).
2. Obviously, when it comes to `useEffect` and `useLayoutEffect` we aren’t talking about “pure” functions. These hooks are designed specifically to handle impurity. Impurity is necessary to any application, but ought to be strictly and wisely managed. Effect hooks are a key component of that strict and wise management.
3. Props and arguments are, for all intents and purposes, the same thing. We just use different names for them for different contexts. Strictly speaking, arguments are a more generic term and refer to all the values sent to a function. Props are all the properties (see where we got the term “props”?) on the first argument sent to a function being treated as a React component.

Get unlimited access Open in app