React Hooks versus Classes

Rick L
Oct 3, 2019 · 7 min read ★

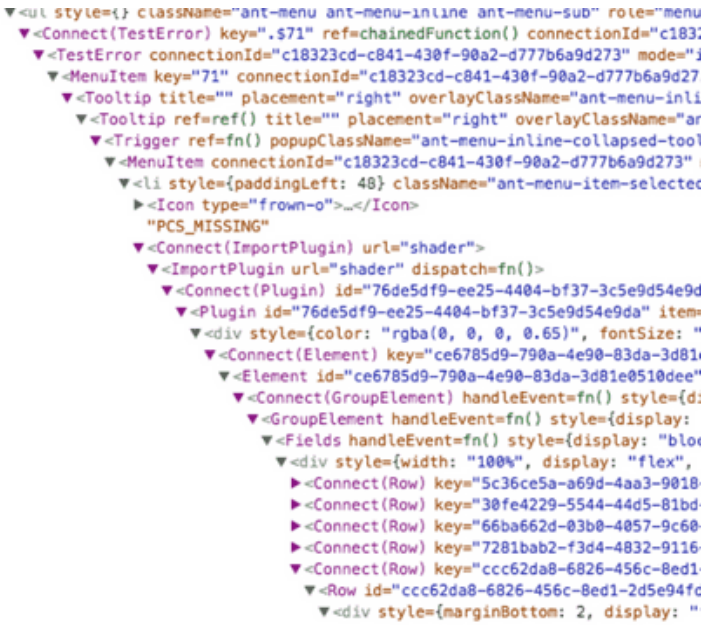
What's the difference, which should you use in your app, and why?

Since React is so popular among developers today, this blog is intended to give you the pros and cons of React hooks vs. classes through React 16.8's release of `useState()` and `useEffect()`'s hooks API.

The problem: React doesn't provide a stateful primitive simpler than a class component—Dan Abramov

First, we will briefly discuss state, then we will go over what *hooks* and *classes* are in React. Finally, we'll see how the release of hooks in **React 16.8** solved the following pain points:

1. **Managing State:** Reusing logic between multiple components can lead to *wrapper hell* or deeply nested components.



2. **Side Effects:** Unrelated mixed in logic in lifecycle methods can get repetitive, and cause unnecessary side effects.

3. **Optimization:** Hooks might reduce your bundle size.

. . .

Managing Local State

What is state in React?

In simple terms, state is simply an object that contains all your key-value pairs. State determines how your components render and behave.

State allows your components to be dynamic and interactive.

State is not to be confused with `props`. State is what is managed within the component, whereas `props` is what gets passed to the component.

How classes manage local state in React

Class components come from ES6 classes and are the default method for managing local state. They also allow side effects to occur through lifecycle methods.

To access and manage state in a class, you have to initialize `this.state` as an object within your `constructor()`, name your local state as a key, and set its initial value as the key's value.

`setState()` is the default method for updating the state in a class, and this is what causes a component to re-render.

Furthermore, it is recommended to call `setState()` every time you want to modify state correctly pre hooks.

Here is a simple example of a counter with an increment button written in a class:

local state is set to counter = 0, handleIncrement() is passed as a callback into button

To set up a class component, you need a fair bit of boilerplate code, which is not limited to your conventional `constructor()` within your class, and the `super()` for extending the component.

For instance, it is necessary to add the `this` context in class components and bind it inside the `constructor()` on line 9.

We would need to bind `this` because of implicit binding in vanilla JavaScript.

When we pass the event handler function reference as a callback like this (`<button type="button" onClick={this.handleIncrement}>+ </button>`), we lose the context of `this` because `handleIncrement()` becomes just a regular function call without an owner object.

In this case, the value of `this` falls back to default binding and points to the global object or `undefined` if the function being invoked is using strict mode.

Finally, we need to wrap our return statement inside the `render()` function.

. . .

How React Hooks Manage Local State

Note: Hooks are completely on an opt-in basis and 100% backward-compatible. This means you don't have to learn or use hooks right away, and there will be no breaking changes when adding or refactoring your classes.

Hooks allow you to use local state and other React features without writing a class.

Hooks are special functions that let you "hook onto" React state and lifecycle features inside function components.

Important: React internally can't keep track of hooks that run out of order. Since hooks return an array, the order that they get called matters.

There are **two rules** for hooks:

1. Only call hooks at the top level—do not nest your hooks in any logic.

Do not do this!

```
if (bool) { const [counter, incrementCounter] = useState(0) }
```

2. Only call hooks from React functions or custom hooks.

Since React components are re-rendered each time the data changes, this means the exact same hooks must be called in the exact same order on every single render. If we wrapped our hooks in a conditional or function, the state would sometimes be created and other times wouldn't.

. . .

`useState()` is a hook that lets you add a React state to function components.

This is the same example but written without a class and instead with hooks:

local state is const counter = 0,

By importing and calling `useState()`, it declares a "state variable" `counter` with a value of whatever argument is being passed into

`useState()` . In this case, our state variable `counter` has a value of zero, as set in line 4.

Note: `useState()`'s argument is not limited to an object, it can be a primitive, e.g., numbers, strings, boolean, etc.

`useState()` only takes one argument, the initial state.

`useState()` returns a pair of values in an array, the current state and a function that updates it.

However, unlike `this.setState()` in a class, updating the current state always **replaces** it instead of merging it.

By destructuring our array into two variables, we can use a more declarative approach, since we know the first value returned in the array is the current state, and the second value is the function which updates the state. (Line 4)

This is a concept in programming called *coupling*, and by closely grouping the two values, we know they are closely dependent on one another.

Therefore, our current state is the value of `count` , which is 0, and our `incrementCounter` is the function that updates `count` .

Note: `incrementCounter()` needs to be wrapped in a function and passed as a callback into our button.

Notice how each variable correlates with its respective value, and our functions stay dry and reusable.

In addition, there is no need for the `this` context anymore, saving us some finger strength and time.

. . .

What Is a Side Effect?

A side effect is generally anything that affects something outside the scope of the function being executed or, in the context of React, anything that modifies state outside of its local environment.

Common side effects include data fetching, setting up subscriptions, and manually changing the DOM in React components.

In the case of React, there are two common cases of side effects: those that don't require cleanup and those that do.

Examples of effects that don't need cleanup are network requests, manual DOM mutations, and logging. This is because we run them and immediately forget about them.

If we wanted to clean up after our side effects, we would need to return a function with the unmount logic inside.

Side effects using classes in React

Class example with manual DOM mutation:

componentDidMount() sets the document title = counter on initial mount; componentDidUpdate() updates the document title after every change to counter after initial mount.

This is an example of a side effect being introduced through React's lifecycle methods found in classes. E.g. `componentDidMount()`, `componentDidUpdate()`, `componentWillUnMount()`.

In this example, the `componentDidMount()` "mounts" or sets up the title of the document to be the current count of the local state.

The `componentDidUpdate()` is invoked as soon as the updating happens. The most common use case for `componentDidUpdate()` is updating the DOM in response to a prop or state change.

If we wanted to reset the count, we would also need a `componentWillUnMount()`.

Here is a better read on lifecycle methods: React Lifecycle Methods — A Deep Dive by Mosh Hamedani.

Side effects using React hooks

Function example with the `useEffect()` hook:

useEffect() takes a callback

`useEffect()` lets you use side effects in your function components.

`useEffect()` tells your component to do something after every render.

React will remember the callback being passed in and call it after the DOM updates.

`useEffect()` is placed inside our function component because we want to have access to our local state count.

Additionally, `useEffect()` runs after every render. Therefore, it is like a `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnMount()` all in one.

During our three-week capstone at the Fullstack Academy of Code, we used functional components with hooks using `useEffect()` to fetch NYC open data and remote custom databases in arcGIS.

. . .

Optimizing Performance by Skipping Effects

Class example

Cleaning up and applying the effect after every render is task heavy, and we might right run into issues or bugs.

If for instance, we wanted to limit our `document.title` to a maximum count of 10:

In **class** components, we can combat this by adding an extra conditional into our `componentDidUpdate()` function and passing in `prevProps` and `prevState` as parameters. (Lines 16–17)

Hooks example

With **hooks**, we can simply pass a second argument into `useEffect()` as an array with a counter in it and add the conditional inside our `useEffect()`.

Whatever is being passed into the array can be used to define all variables on which the hook depends. If one of the variables updates, the hook runs again.

Keep in mind, if you pass an empty array, the hook doesn't run when updating the component at all because there is nothing to watch for. This is useful when you are fetching data in a loop and you only want to fetch it on `componentDidMount()`, therefore stopping the loop.

pass [counter] into the second argument of useEffect() — line 10

. . .

Conclusion

While hooks solved many of the pain points that we experienced using classes in React, there are still other use cases for classes, like if you wanted to access specific lifecycle methods.

Again, this guide was **not** meant to convince you to use hooks or completely refactor your classes to hooks.

Just a friendly reminder that there are other options out there to experiment with!

If you are really interested in learning more about hooks, try to apply the concepts to new projects you initiate in the future instead.

Until next time. Happy Coding!—*RL*

If you are interested in learning more about hooks, like accessing context API, etc., see the context API.