Introducing Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

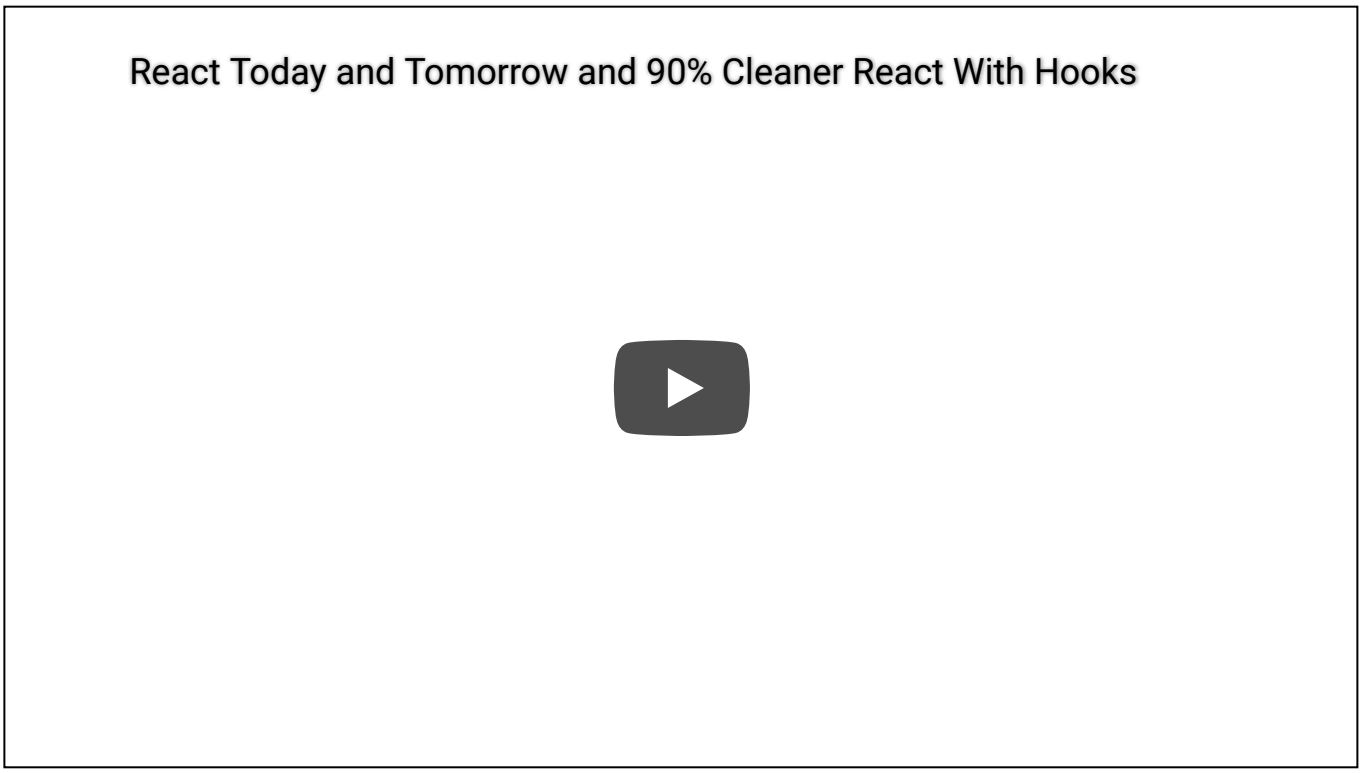
This new function `useState` is the first "Hook" we'll learn about, but this example is just a teaser. Don't worry if it doesn't make sense yet!

You can start learning Hooks on the next page. On this page, we'll continue by explaining why we're adding Hooks to React and how they can help you write great applications.

> **Note**
>
> React 16.8.0 is the first release to support Hooks. When upgrading, don't forget to update all packages, including React DOM. React Native supports Hooks since the 0.59 release of React Native.

Video Introduction

At React Conf 2018, Sophie Alpert and Dan Abramov introduced Hooks, followed by Ryan Florence demonstrating how to refactor an application to use them. Watch the video here:



No Breaking Changes

Before we continue, note that Hooks are:

- **Completely opt-in.** You can try Hooks in a few components without rewriting any existing code. But you don't have to learn or use Hooks right now if you don't want to.
- **100% backwards-compatible.** Hooks don't contain any breaking changes.
- **Available now.** Hooks are now available with the release of v16.8.0.

There are no plans to remove classes from React. You can read more about the gradual adoption strategy for Hooks in the bottom section of this page.

Hooks don't replace your knowledge of React concepts. Instead, Hooks provide a m direct API to the React concepts you already know: props, state, context, refs, and lifecycle. we will show later, Hooks also offer a new powerful way to combine them.

If you just want to start learning Hooks, feel free to jump directly to the next page! You can also keep reading this page to learn more about why we're adding Hooks, and how we're going to start using them without rewriting our applications.

Motivation

Hooks solve a wide variety of seemingly unconnected problems in React that we've encountered over five years of writing and maintaining tens of thousands of components. Whether you're learning React, use it daily, or even prefer a different library with a similar component model, you might recognize some of these problems.

It's hard to reuse stateful logic between components

React doesn't offer a way to "attach" reusable behavior to a component (for example, connecting it to a store). If you've worked with React for a while, you may be familiar with patterns like render props and higher-order components that try to solve this. But these patterns require you to restructure your components when you use them, which can be cumbersome and make code harder to follow. If you look at a typical React application in React DevTools, you will likely find a "wrapper hell" of components surrounded by layers of providers, consumers, higher-order components, render props, and other abstractions. While we could filter them out in DevTools, this points to a deeper underlying problem: React needs a better primitive for sharing stateful logic.

With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. **Hooks allow you to reuse stateful logic without changing your component hierarchy.** This makes it easy to share Hooks among many components or with the community.

We'll discuss this more in Building Your Own Hooks.

Complex components become hard to understand

We've often had to maintain components that started out simple but grew into an unmanageable mess of stateful logic and side effects. Each lifecycle method often contains a mix of unrelated logic. For example, components might perform some data fetching in

mix of unrelated logic. For example, components might perform some data fetching in `componentDidMount` and `componentDidUpdate`. However, the same `componentDidMount` method might also contain some unrelated logic that sets up event listeners, with cleanup performed in `componentWillUnmount`. Mutually related code that changes together gets split apart, but completely unrelated code ends up combined in a single method. This makes it too easy to introduce bugs and inconsistencies.

In many cases it's not possible to break these components into smaller ones because the stateful logic is all over the place. It's also difficult to test them. This is one of the reasons many people prefer to combine React with a separate state management library. However, that often introduces too much abstraction, requires you to jump between different files, and makes reusing components more difficult.

To solve this, **Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data)**, rather than forcing a split based on lifecycle methods. You may also opt into managing the component's local state with a reducer to make it more predictable.

We'll discuss this more in Using the Effect Hook.

Classes confuse both people and machines

In addition to making code reuse and code organization more difficult, we've found that classes can be a large barrier to learning React. You have to understand how `this` works in JavaScript, which is very different from how it works in most languages. You have to remember to bind the event handlers. Without unstable syntax proposals, the code is very verbose. People can understand props, state, and top-down data flow perfectly well but still struggle with classes. The distinction between function and class components in React and when to use each one leads to disagreements even between experienced React developers.

Additionally, React has been out for about five years, and we want to make sure it stays relevant in the next five years. As Svelte, Angular, Glimmer, and others show, ahead-of-time compilation of components has a lot of future potential. Especially if it's not limited to templates. Recently, we've been experimenting with component folding using Prepack, and we've seen promising early results. However, we found that class components can encourage unintentional patterns that make these optimizations fall back to a slower path. Classes r issues for today's tools, too. For example, classes don't minify very well, and they make h reloading flaky and unreliable. We want to present an API that makes it more likely for code to stay on the optimizable path.

To solve these problems, **Hooks let you use more of React's features without classes.** Conceptually, React components have always been closer to functions. Hooks embrace functions, but without sacrificing the practical spirit of React. Hooks provide access to imperative escape hatches and don't require you to learn complex functional or reactive programming techniques.

> **Examples**
> Hooks at a Glance is a good place to start learning Hooks.

Gradual Adoption Strategy

> **TLDR: There are no plans to remove classes from React.**

We know that React developers are focused on shipping products and don't have time to look into every new API that's being released. Hooks are very new, and it might be better to wait for more examples and tutorials before considering learning or adopting them.

We also understand that the bar for adding a new primitive to React is extremely high. For curious readers, we have prepared a detailed RFC that dives into motivation with more details, and provides extra perspective on the specific design decisions and related prior art.

Crucially, Hooks work side-by-side with existing code so you can adopt them gradually. There is no rush to migrate to Hooks. We recommend avoiding any "big rewrites", especially for existing, complex class components. It takes a bit of a mind shift to start "thinking in Hooks". In our experience, it's best to practice using Hooks in new and non-critical components first, and ensure that everybody on your team feels comfortable with them. After you give Hooks a try, please feel free to send us feedback, positive or negative.

We intend for Hooks to cover all existing use cases for classes, but **we will keep suppor class components for the foreseeable future.** At Facebook, we have tens of thousands components written as classes, and we have absolutely no plans to rewrite them. Instead, we are starting to use Hooks in the new code side by side with classes.

are starting to use Hooks in the new code side by side with classes.

Frequently Asked Questions

We've prepared a Hooks FAQ page that answers the most common questions about Hooks.

Next Steps

By the end of this page, you should have a rough idea of what problems Hooks are solving, but many details are probably unclear. Don't worry! **Let's now go to the next page where we start learning about Hooks by example.**

Is this page useful?  Edit this page

Next article

Hooks at a Glance