

Using the Effect Hook

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

The *Effect Hook* lets you perform side effects in function components:

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

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

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

This snippet is based on the [counter example](#) from the [previous page](#), but we added a new feature to it: we set the document title to a custom message including the number of clicks.

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you're used to calling these operations "side effects" (or just "effects"), you've likely performed them in your components before.



If you're familiar with React class lifecycle methods, you can think of `useEffect` Hook as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined.

There are two common kinds of side effects in React components: those that don't require cleanup, and those that do. Let's look at this distinction in more detail.

Effects Without Cleanup

Sometimes, we want to **run some additional code after React has updated the DOM**. Network requests, manual DOM mutations, and logging are common examples of effects that don't require a cleanup. We say that because we can run them and immediately forget about them. Let's compare how classes and Hooks let us express such side effects.

Example Using Classes

In React class components, the `render` method itself shouldn't cause side effects. It would be too early — we typically want to perform our effects *after* React has updated the DOM.

This is why in React classes, we put side effects into `componentDidMount` and `componentDidUpdate`. Coming back to our example, here is a React counter class component that updates the document title right after React makes changes to the DOM:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
```



```

    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}

```

Note how **we have to duplicate the code between these two lifecycle methods in class.**

This is because in many cases we want to perform the same side effect regardless of whether the component just mounted, or if it has been updated. Conceptually, we want it to happen after every render — but React class components don't have a method like this. We could extract a separate method but we would still have to call it in two places.

Now let's see how we can do the same with the `useEffect` Hook.

Example Using Hooks

We've already seen this example at the top of this page, but let's take a closer look at it:

```

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

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

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

```



```
    Click me
  </button>
</div>
);
}
```

What does `useEffect` do? By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API.

Why is `useEffect` called inside a component? Placing `useEffect` inside the component lets us access the `count` state variable (or any props) right from the effect. We don't need a special API to read it — it's already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution.

Does `useEffect` run after every render? Yes! By default, it runs both after the first render *and* after every update. (We will later talk about how to customize this.) Instead of thinking in terms of "mounting" and "updating", you might find it easier to think that effects happen "after render". React guarantees the DOM has been updated by the time it runs the effects.

Detailed Explanation

Now that we know more about effects, these lines should make sense:

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

We declare the `count` state variable, and then we tell React we need to use an effect. We pass a function to the `useEffect` Hook. This function we pass *is* our effect. Inside our effect, we set the document title using the `document.title` browser API. We can read the latest `count` inside the effect because it's in the scope of our function. When React renders our component,



it will remember the effect we used, and then run our effect after updating the DOM. This happens for every render, including the first one.

Experienced JavaScript developers might notice that the function passed to `useEffect` is going to be different on every render. This is intentional. In fact, this is what lets us read the `count` value from inside the effect without worrying about it getting stale. Every time we re-render, we schedule a *different* effect, replacing the previous one. In a way, this makes the effects behave more like a part of the render result — each effect “belongs” to a particular render. We will see more clearly why this is useful [later on this page](#).

Tip

Unlike `componentDidMount` or `componentDidUpdate`, effects scheduled with `useEffect` don't block the browser from updating the screen. This makes your app feel more responsive. The majority of effects don't need to happen synchronously. In the uncommon cases where they do (such as measuring the layout), there is a separate `useLayoutEffect` Hook with an API identical to `useEffect`.

Effects with Cleanup

Earlier, we looked at how to express side effects that don't require any cleanup. However, some effects do. For example, **we might want to set up a subscription** to some external data source. In that case, it is important to clean up so that we don't introduce a memory leak! Let's compare how we can do it with classes and with Hooks.

Example Using Classes

In a React class, you would typically set up a subscription in `componentDidMount`, and clean up in `componentWillUnmount`. For example, let's say we have a `ChatAPI` module that lets us subscribe to a friend's online status. Here's how we might subscribe and display that status using a class:



```

class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}

```

Notice how `componentDidMount` and `componentWillUnmount` need to mirror each other. Lifecycle methods force us to split this logic even though conceptually code in both of them is related to the same effect.

Note

Eagle-eyed readers may notice that this example also needs a `componentDidUpdate` method to be fully correct. We'll ignore this for now but will come back to it in a [later section](#) of this page.



Example Using Hooks

Let's see how we could write this component with Hooks.

You might be thinking that we'd need a separate effect to perform the cleanup. But code for adding and removing a subscription is so tightly related that `useEffect` is designed to keep it together. If your effect returns a function, React will run it when it is time to clean up:

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Why did we return a function from our effect? This is the optional cleanup mechanism for effects. Every effect may return a function that cleans up after it. This lets us keep the logic for adding and removing subscriptions close to each other. They're part of the same effect!

When exactly does React clean up an effect? React performs the cleanup when the component unmounts. However, as we learned earlier, effects run for every render and not just once. This is why React also cleans up effects from the previous render before running the



once. This is why React also cleans up effects from the previous render before running the

effects next time. We'll discuss [why this helps avoid bugs](#) and [how to opt out of this behavior](#) in case it creates performance issues later below.

Note

We don't have to return a named function from the effect. We called it `cleanup` here to clarify its purpose, but you could return an arrow function or call it something different.

Recap

We've learned that `useEffect` lets us express different kinds of side effects after a component renders. Some effects might require cleanup so they return a function:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

Other effects might not have a cleanup phase, and don't return anything.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

The Effect Hook unifies both use cases with a single API.



If you feel like you have a decent grasp on how the Effect Hook works, or if you feel overwhelmed, you can jump to the [next page about Rules of Hooks](#) now.

Tips for Using Effects

We'll continue this page with an in-depth look at some aspects of `useEffect` that experienced React users will likely be curious about. Don't feel obligated to dig into them now. You can always come back to this page to learn more details about the Effect Hook.

Tip: Use Multiple Effects to Separate Concerns

One of the problems we outlined in the [Motivation](#) for Hooks is that class lifecycle methods often contain unrelated logic, but related logic gets broken up into several methods. Here is a component that combines the counter and the friend status indicator logic from the previous examples:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
```



```

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

handleStatusChange(status) {
  this.setState({
    isOnline: status.isOnline
  });
}
// ...

```

Note how the logic that sets `document.title` is split between `componentDidMount` and `componentDidUpdate`. The subscription logic is also spread between `componentDidMount` and `componentWillUnmount`. And `componentDidMount` contains code for both tasks.

So, how can Hooks solve this problem? Just like you can use the *State* Hook more than once, you can also use several effects. This lets us separate unrelated logic into different effects:

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  // ...
}

```



Hooks let us split the code based on what it is doing rather than a lifecycle method name.

React will apply *every* effect used by the component, in the order they were specified.

Explanation: Why Effects Run on Each Update

If you're used to classes, you might be wondering why the effect cleanup phase happens after every re-render, and not just once during unmounting. Let's look at a practical example to see why this design helps us create components with fewer bugs.

Earlier on [this page](#), we introduced an example `FriendStatus` component that displays whether a friend is online or not. Our class reads `friend.id` from `this.props`, subscribes to the friend status after the component mounts, and unsubscribes during unmounting:

```
componentDidMount() {  
  ChatAPI.subscribeToFriendStatus(  
    this.props.friend.id,  
    this.handleStatusChange  
  );  
}  
  
componentWillUnmount() {  
  ChatAPI.unsubscribeFromFriendStatus(  
    this.props.friend.id,  
    this.handleStatusChange  
  );  
}
```

But what happens if the friend prop changes while the component is on the screen? Our component would continue displaying the online status of a different friend. This is a bug. We would also cause a memory leak or crash when unmounting since the unsubscribe call would use the wrong friend ID.

In a class component, we would need to add `componentDidUpdate` to handle this case:

```
componentDidMount() {  
  ChatAPI.subscribeToFriendStatus(  
    this.props.friend.id,  
    this.handleStatusChange  
  );  
}  
  
componentDidUpdate(prevProps) {
```



```

    // Unsubscribe from the previous friend.id

    ChatAPI.unsubscribeFromFriendStatus(
      prevProps.friend.id,
      this.handleStatusChange
    );
    // Subscribe to the next friend.id
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
}

```

Forgetting to handle `componentDidUpdate` properly is a common source of bugs in React applications.

Now consider the version of this component that uses Hooks:

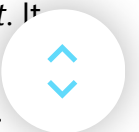
```

function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}

```

It doesn't suffer from this bug. (But we also didn't make any changes to it.)

There is no special code for handling updates because `useEffect` handles them *by default*. It cleans up the previous effects before applying the next effects. To illustrate this, here is a sequence of subscribe and unsubscribe calls that this component could produce over time:



```
// Mount with { friend: { id: 100 } } props
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Run first effect

// Update with { friend: { id: 200 } } props
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Clean up previous
effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Run next effect

// Update with { friend: { id: 300 } } props
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Clean up previous
effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // Run next effect

// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Clean up last effect
```

This behavior ensures consistency by default and prevents bugs that are common in class components due to missing update logic.

Tip: Optimizing Performance by Skipping Effects

In some cases, cleaning up or applying the effect after every render might create a performance problem. In class components, we can solve this by writing an extra comparison with `prevProps` or `prevState` inside `componentDidUpdate`:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

This requirement is common enough that it is built into the `useEffect` Hook API. You can tell React to *skip* applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to `useEffect`:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```



In the example above, we pass `[count]` as the second argument. What does this mean? If the `count` is 5, and then our component re-renders with `count` still equal to 5, React will compare `[5]` from the previous render and `[5]` from the next render. Because all items in the array are the same (`5 === 5`), React would skip the effect. That's our optimization.

When we render with `count` updated to 6, React will compare the items in the `[5]` array from the previous render to items in the `[6]` array from the next render. This time, React will re-apply the effect because `5 !== 6`. If there are multiple items in the array, React will re-run the effect even if just one of them is different.

This also works for effects that have a cleanup phase:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // Only re-subscribe if props.friend.id changes
```

In the future, the second argument might get added automatically by a build-time transformation.

Note

If you use this optimization, make sure the array includes **all values from the component scope (such as props and state) that change over time and that are used by the effect**. Otherwise, your code will reference stale values from previous renders. Learn more about [how to deal with functions](#) and [what to do when the array changes too often](#).

If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array (`[]`) as a second argument. This tells React that your effect doesn't depend on *any* values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.



If you pass an empty array (`[]`), the props and state inside the effect will always have their initial values. While passing `[]` as the second argument is closer to the familiar `componentDidMount` and `componentWillUnmount` mental model, there are usually better solutions to avoid re-running effects too often. Also, don't forget that React defers running `useEffect` until after the browser has painted, so doing extra work is less of a problem.



We recommend using the exhaustive-deps rule as part of our eslint-plugin-react-hooks package. It warns when dependencies are specified incorrectly and suggests a fix.

Next Steps

Congratulations! This was a long page, but hopefully by the end most of your questions about effects were answered. You've learned both the State Hook and the Effect Hook, and there is a *lot* you can do with both of them combined. They cover most of the use cases for classes — and where they don't, you might find the additional Hooks helpful.

We're also starting to see how Hooks solve problems outlined in Motivation. We've seen how effect cleanup avoids duplication in `componentDidUpdate` and `componentWillUnmount`, brings related code closer together, and helps us avoid bugs. We've also seen how we can separate effects by their purpose, which is something we couldn't do in classes at all.

At this point you might be questioning how Hooks work. How can React know which `useState` call corresponds to which state variable between re-renders? How does React "match up" previous and next effects on every update? **On the next page we will learn about the Rules of Hooks — they're essential to making Hooks work.**

Is this page useful?  

[Edit this page](#)



[Previous article](#)

[Using the State Hook](#)

[Next article](#)

[Rules of Hooks](#)

