How the useEffect Hook Works (with Examples)

UPDATED OCTOBER 22, 2020

The useEffect hook is the Swiss Army knife of all the hooks. It's the solution to many problems: how to fetch data when a component mounts, how to run code when state changes or when a prop changes, how to set up timers or intervals, you name it.

Pretty much anything you want to "do" in a React component other than return JSX (any sort of side effect), will go into a useEffect. (and you can have more than one useEffect per component, too)

All this power comes with a tradeoff: useEffect can be confusing until you understand how it works.

In this post, we're going to look at lots of useEffect examples so that you understand the mental model and can use it effectively in your own code.

Here's what we'll cover:

useEffect vs Lifecycle Methods

If you've used class components and lifecycle methods, read this section first.

If you've never touched classes and never intend to, you can disregard the comparison to lifecycles – but this section will still be useful for learning *when* useEffect runs and how it fits into React's render cycle.

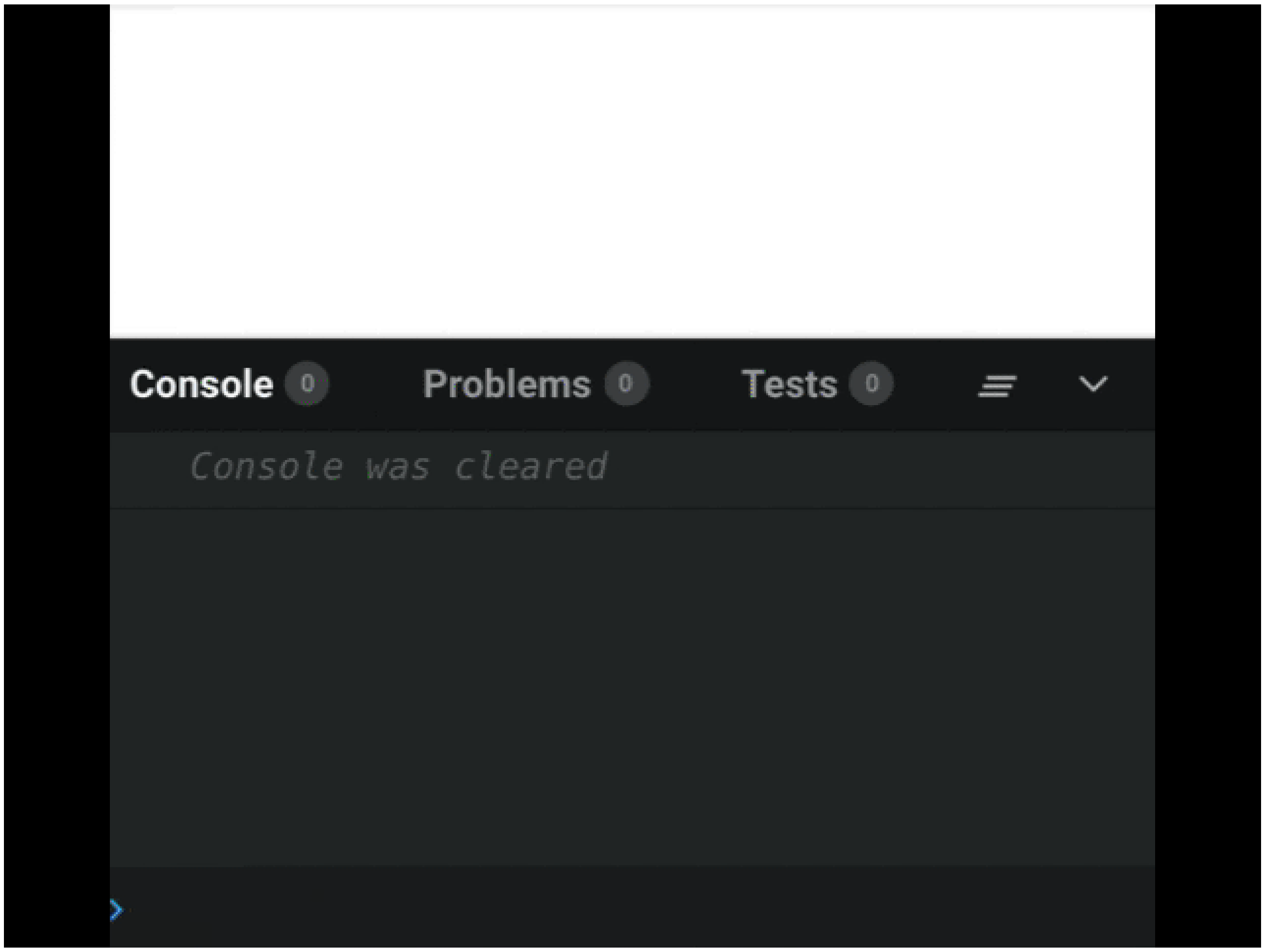
Let's look at how to run code after a component mounts (`componentDidMount`), after it re-renders (`componentDidUpdate`), and before it unmounts (`componentWillUnmount`).

```jsx
import React, { useEffect, useState } from 'react';
import ReactDOM from 'react-dom';
```

```
function LifecycleDemo() {
  // Pass useEffect a function
  useEffect(() => {
    // This gets called after every render, by default
    // (the first one, and every one after that)
    console.log('render!');

    // If you want to implement componentWillUnmount,
    // return a function from here, and React will call
    // it prior to unmounting.
    return () => console.log('unmounting...');
  })

  return "I'm a lifecycle demo";
}

function App() {
  // Set up a piece of state, so that we have
  // a way to trigger a re-render.
  const [random, setRandom] = useState(Math.random());

  // Set up another piece of state to keep track of
  // whether the LifecycleDemo is shown or hidden
  const [mounted, setMounted] = useState(true);

  // This function will change the random number,
  // and trigger a re-render (in the console,
  // you'll see a "render!" from LifecycleDemo)
  const reRender = () => setRandom(Math.random());

  // This function will unmount and re-mount the
  // LifecycleDemo, so you can see its cleanup function
  // being called.
  const toggle = () => setMounted(!mounted);

  return (
    <>
      <button onClick={reRender}>Re-render</button>
      <button onClick={toggle}>Show/Hide LifecycleDemo</button>
      {mounted && <LifecycleDemo/>}
    </>
  );
}

ReactDOM.render(<App/>, document.querySelector('#root'));
```

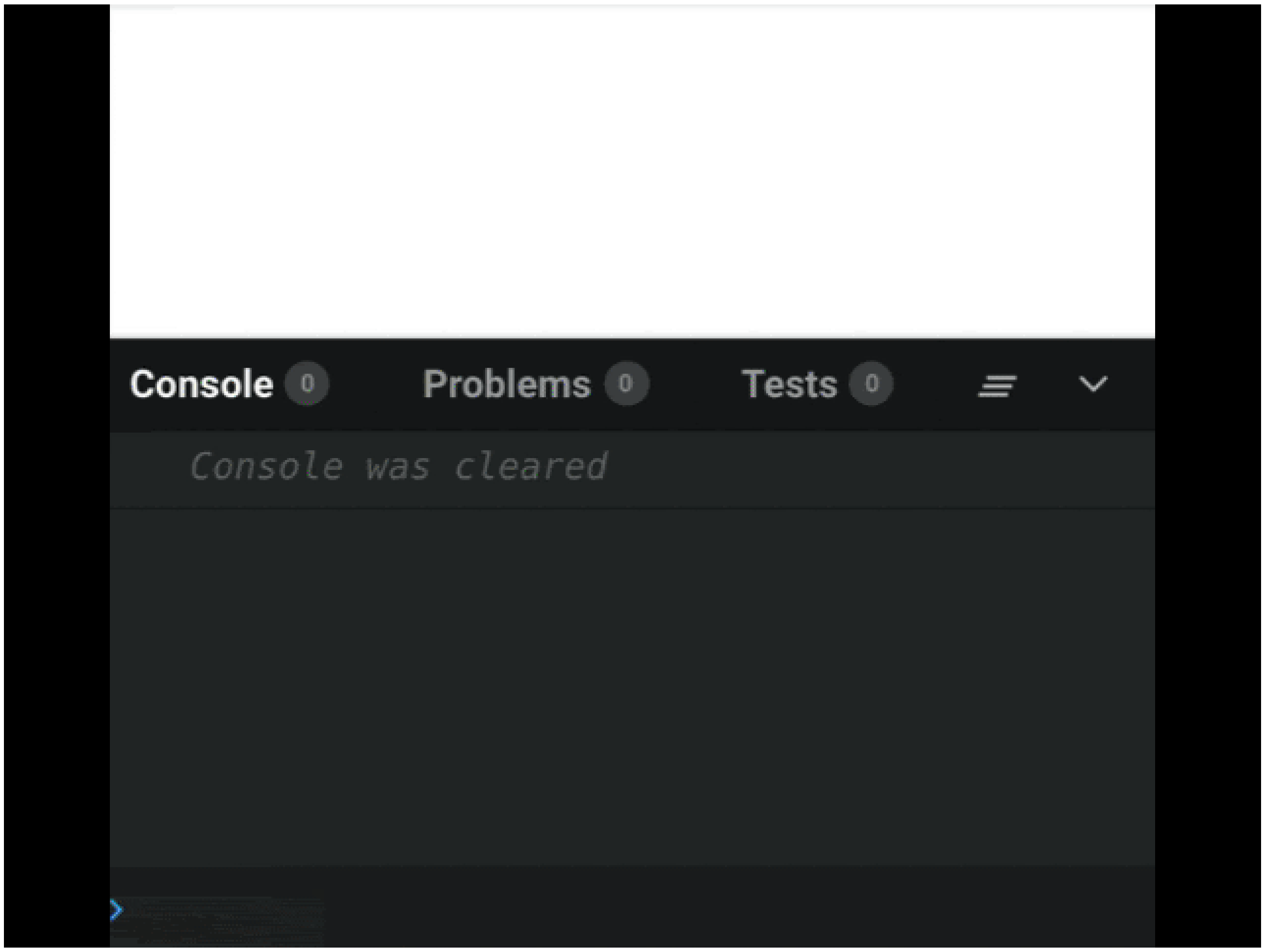
Try it out in CodeSandbox.

Click the Show/Hide button. Look at the console. It prints “unmounting” before it disappears, and “render!” when it reappears.



```
Console was cleared
```

Now, try the Re-render button. With each click, it prints “unmounting” *and* “render!” every time.



```
Console was cleared
```

Why is it “unmounting” with every render?

Well, the cleanup function you can (optionally) return from `useEffect` isn’t *only* called when the component is unmounted. It’s called every time before that effect runs – to clean up from the last run. This is actually more powerful than the `componentWillUnmount` lifecycle because it lets you run a side effect before and after every render, if you need to.

Not Quite Lifecycles

`useEffect` runs after every render (by default), and can optionally clean up for itself before it runs again.

Rather than thinking of `useEffect` as one function doing the job of 3 separate lifecycles, it might be more helpful to think of it simply as a way to run side effects after render – including the potential cleanup you'd want to do before each one, and before unmounting.

Success! Now check your email.

Prevent useEffect From Running Every Render

If you want your effects to run less often, you can provide a second argument – an array of values. Think of them as the *dependencies* for that effect. If one of the dependencies has changed since the last time, the effect will run again. (It will also still run after the initial render)

```jsx
const [value, setValue] = useState('initial');

useEffect(() => {
  // This effect uses the `value` variable,
  // so it "depends on" `value`.
  console.log(value);
}, [value])  // pass `value` as a dependency
```

Another way to think of this array: it should contain every variable that the effect function uses from the surrounding scope. So if it uses a prop? That goes in the array. If it uses a piece of state? That goes in the array.

useEffect Does Not Actively "Watch"

Some frameworks are *reactive*, meaning they automatically detect changes and update the UI when changes occur.

React does not do this – it will only re-render in response to state changes.

useEffect, too, does not actively "watch" for changes. When you call `useEffect` in your component, this is effectively queuing or scheduling an effect to *maybe* run, *after* the render is done.

After rendering finishes, `useEffect` will check the list of dependency values against the values from the last render, and will call your effect function if any one of them has changed.

> **Without the right mental model, useEffect is super confusing.**
> With the *right* mental model, you'll sidestep the infinite loops and dependency warnings before they happen.
> Get great at useEffect *this afternoon* with Learn useEffect Over Lunch.

Only Run Once, on Mount

You can pass the special value of *empty array* `[]` as a way of saying "only run on mount, and clean up on unmount". So if we changed our component above to call `useEffect` like this:

```jsx
useEffect(() => {
  console.log('mounted');
  return () => console.log('unmounting...');
}, [])  // <-- add this empty array here
```

Then it will print "mounted" after the initial render, remain silent throughout its life, and print "unmounting..." on its way out.

Be careful with the second argument: It's easy to forget to add an item to it if you add a dependency, and if you *miss* a dependency, then that value will be stale the next time `useEffect` runs and it might cause some strange problems.

How To Fix The Warnings (Don't Ignore The Warnings!)

A common mistake people make is to pass the `[]` to useEffect even though the effect *does* depend on some variables. If you're using Create React App or the React ESLint rules, you'll get a warning about this in the browser console. Don't ignore these warnings!

You're passing a function to useEffect. This function creates a closure, which "latches on" to the values of any variables it refers to, at the time the function is created.

The empty array says "never re-create the closure, because this effect doesn't refer to any variables that will change".

With the empty array being passed, useEffect will hang on to the first function you pass it, which in turn will hang on to references to all the (maybe stale) variables that were used inside it.

The entire point of the dependency array is to give you a way to tell React "Hey, one of my variables changed! Re-create that closure!"

If you get a warning, you need to fix it. Take all the variables it's complaining about, and put them in the array. *Usually* that works fine.

Sometimes, you can get into an infinite loop, or a situation where the effect runs on every render even though it seems like it shouldn't. In many cases, the easiest fix for this is to use the functional form of setState. I also cover a few situations where this can go wrong and how to fix them in Learn useEffect Over Lunch.

When Does useEffect Run?

By default, `useEffect` runs *after* each render of the component where it's called. This timing is easiest to see with an example. Look over the code below, and try the interactive example on CodeSandbox, making sure to open up the console so you can see the timing.

In this example, there are 3 nested components: Top contains Middle, and Middle contains Bottom. The timing of useEffect depends on when each component renders, and initially, all 3 will be rendered. You'll see 3 messages printed to the console.

Notice, though, that React renders from the bottom up! In this case: Bottom, then Middle, then Top. It's recursive – the parent is not "done" until all of its children have rendered, and the `useEffect` will only run *after* the render of a component is complete.

From then on, nothing will happen until you click on one of the elements to increment its count. When you do, the only components that will re-render are the one you clicked, and the ones below it. (Notice that if you click on `Bottom`, you won't see "rendered" messages from Top or Middle)

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

  useEffect(() => {
    console.log("Top rendered");
  });

  return (
    <div>
      <div onClick={() => setCount(count + 1)}>Top Level {count}</div>
      <Middle />
    </div>
  );
}

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

  useEffect(() => {
    console.log("Middle rendered");
  });
  return (
    <div>
      <div onClick={() => setCount(count + 1)}>Middle Level {count}</di
      <Bottom />
    </div>
  );
}

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

  useEffect(() => {
    console.log("Bottom rendered");
  });

  return <div onClick={() => setCount(count + 1)}>Bottom Level {count}<
}
```

Run useEffect on State Change

By default, `useEffect` runs after every render, but it's also perfect for running some code in response to a state change. You can limit when the effect runs by passing the second argument to useEffect.

Think of the second argument as an array of "dependencies" – variables that, if changed, the effect should rerun. These can be any kind of variable: props, state, or anything else.

In this example, there are 3 state variables, and 3 buttons. The effect will only run when `count2` changes, and will stay quiet otherwise. Try the interactive example.

```jsx
function ThreeCounts() {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);
  const [count3, setCount3] = useState(0);

  useEffect(() => {
    console.log("count2 changed!");
  }, [count2]);

  return (
    <div>
      {count1} {count2} {count3}
      <br />
      <button onClick={() => setCount1(count1 + 1)}>Increment count1</b
      <button onClick={() => setCount2(count2 + 1)}>Increment count2</b
      <button onClick={() => setCount3(count3 + 1)}>Increment count3</b
    </div>
  );
}
```

Run useEffect When a Prop Changes

Just as we were able to set up useEffect to run when a state variable changed, the same can be done with props. Remember they're all regular variables! `useEffect` can trigger on any of them.

In this example, the `PropChangeWatch` component is receiving 2 props (`a` and `b`), and its effect will only run when the value of `a` changes (because we're passing an array containing `[a]` as the second argument).

Try out the interactive example.:

```jsx
function PropChangeWatch({ a, b }) {
  useEffect(() => {
    console.log("value of 'a' changed to", a);
  }, [a]);

  return (
    <div>
      I've got 2 props: a={a} and b={b}
    </div>
  );
}

function Demo() {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);

  return (
    <div>
      <PropChangeWatch a={count1} b={count2} />
      <button onClick={() => setCount1(count1 + 1)}>Increment count1</b
      <button onClick={() => setCount2(count2 + 1)}>Increment count2</b
    </div>
  );
}
```

Focus On Mount

Sometimes you just want to do *one tiny thing* at mount time, and doing that one little thing requires rewriting a function as a class.

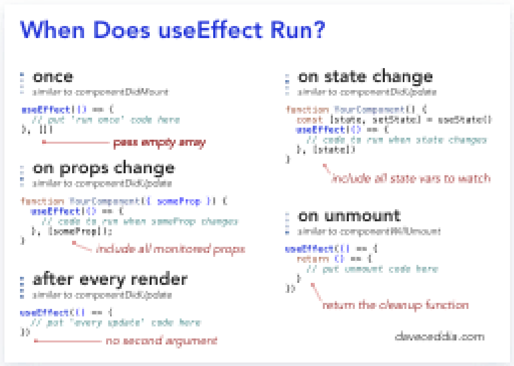
In this example, let's look at how you can focus an input control upon first render, using `useEffect` combined with the `useRef` hook.

```jsx
import React, { useEffect, useState, useRef } from "react";
import ReactDOM from "react-dom";

function App() {
  // Store a reference to the input's DOM node
  const inputRef = useRef();

        // Store the input's value in state
  const [value, setValue] = useState("");
  useEffect(
    () => {
      // This runs AFTER the first render,
      // so the ref is set by now.
      console.log("render");
      // inputRef.current.focus();
    },
                // The effect "depends on" inputRef
    [inputRef]
  );

  return (
    <input
      ref={inputRef}
      value={value}
      onChange={e => setValue(e.target.value)}
    />
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));
```

At the top, we're creating an empty ref with `useRef`. Passing it to the input's `ref` prop takes care of setting it up once the DOM is rendered. And, importantly, the value returned by `useRef` will be stable between renders – it won't change.

So, even though we're passing `[inputRef]` as the 2nd argument of `useEffect`, it will effectively only run once, on initial mount. This is basically "componentDidMount" (except the timing of it, which we'll talk about later).

To prove it, try out the example. Notice how it focuses (it's a little buggy with the CodeSandbox editor, but try clicking the refresh button in the "browser" on the right). Then try typing in the box. Each character triggers a re-render, but if you look at the console, you'll see that "render" is only printed once.



When Does useEffect Run?

once
similar to componentDidMount

```
useEffect(() => {
  // put 'run once' code here
}, [])
```

pass empty array

on props change
similar to componentDidUpdate

```
function YourComponent({ someProp }) {
  useEffect(() => {
    // code to run when someProp changes
  }, [someProp]);
}
```

include all monitored props

after every render
similar to componentDidUpdate

```
useEffect(() => {
  // put 'every update' code here
})
```

no second argument

on state change
similar to componentDidUpdate

```
function YourComponent() {
  const [state, setState] = useState()
  useEffect(() => {
    // code to run when state changes
  }, [state])
}
```

include all state vars to watch

on unmount
similar to componentWillUnmount

```
useEffect(() => {
  return () => {
    // put unmount code here
  }
})
```

return the cleanup function

daveceddia.com

I put together a cheatsheet with different use cases for useEffect, and how they match up with lifecycle methods! Drop your email to get it delivered to your inbox.

Your email address

Get the Cheatsheet

Fetch Data With useEffect

Let's look at another common use case: fetching data and displaying it. In a class component, you'd put this code in the `componentDidMount` method. To do it with hooks, we'll pull in `useEffect`. We'll also need `useState` to store the data.

It's worth mentioning that when the data-fetching portion of React's new Suspense feature is ready, that'll be the preferred way to fetch data. Fetching from `useEffect` has one big gotcha (which we'll go over) and the Suspense API is going to be much easier to use.

Here's a component that fetches posts from Reddit and displays them:

```jsx
import React, { useEffect, useState } from "react";
import ReactDOM from "react-dom";

function Reddit() {
  // Initialize state to hold the posts
  const [posts, setPosts] = useState([]);

  // effect functions can't be async, so declare the
  // async function inside the effect, then call it
  useEffect(() => {
    async function fetchData() {
      // Call fetch as usual
      const res = await fetch(
        "https://www.reddit.com/r/reactjs.json"
      );

      // Pull out the data as usual
      const json = await res.json();

      // Save the posts into state
      // (look at the Network tab to see why the path is like this)
      setPosts(json.data.children.map(c => c.data));
    }

    fetchData();
  }); // <-- we didn't pass a value. what do you think will happen?

  // Render as usual
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

ReactDOM.render(
  <Reddit />,
  document.querySelector("#root")
);
```

You'll notice that we aren't passing the second argument to `useEffect` here. This is bad. Don't do this.

Passing no 2nd argument causes the `useEffect` to run every render. Then, when it runs, it fetches the data and *updates the state*. Then, once the state is updated, the component re-renders, which triggers the `useEffect` again. You can see the problem.

To fix this, we need to pass an array as the 2nd argument. What should be in the array?

Go ahead, think about it for a second.

…

…

The only variable that `useEffect` depends on is `setPosts`. Therefore we should pass the array `[setPosts]` here. Because `setPosts` is a setter returned by `useState`, it won't be recreated every render, and so the effect will only run once.

Fun fact: When you call `useState`, the setter function it returns is only created once! It'll be the exact same function instance every time the component renders, which is why it's safe for an effect to depend on one. This fun fact is also true for the `dispatch` function returned by `useReducer`.

Re-fetch When Data Changes

Let's expand on the example to cover another common problem: how to re-fetch data when something changes, like a user ID, or in this case, the name of the subreddit.

First we'll change the `Reddit` component to accept the subreddit as a prop, fetch the data based on that subreddit, and only re-run the effect when the prop changes:

```jsx
// 1. Destructure the `subreddit` from props:
function Reddit({ subreddit }) {
  const [posts, setPosts] = useState([]);
  useEffect(() => {
    async function fetchData() {
      // 2. Use a template string to set the URL:
      const res = await fetch(
        `https://www.reddit.com/r/${subreddit}.json`
      );

      const json = await res.json();
      setPosts(json.data.children.map(c => c.data));
    }

    fetchData();

    // 3. Re-run this effect when `subreddit` changes:
  }, [subreddit, setPosts]);

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

// 4. Pass "reactjs" as a prop:
ReactDOM.render(
  <Reddit subreddit='reactjs' />,
  document.querySelector("#root")
);
```

This is still hard-coded, but now we can customize it by wrapping the `Reddit` component with one that lets us change the subreddit. Add this new App component, and render it at the bottom:

```jsx
function App() {
  // 2 pieces of state: one to hold the input value,
  // another to hold the current subreddit.
  const [inputValue, setValue] = useState("reactjs");
  const [subreddit, setSubreddit] = useState(inputValue);

  // Update the subreddit when the user presses enter
  const handleSubmit = e => {
    e.preventDefault();
    setSubreddit(inputValue);
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <input
          value={inputValue}
          onChange={e => setValue(e.target.value)}
        />
      </form>
      <Reddit subreddit={subreddit} />
    </>
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));
```

Try the working example on CodeSandbox.

The app is keeping 2 pieces of state here – the current input value, and the current subreddit. Submitting the input “commits” the subreddit, which will cause `Reddit` to re-fetch the data from the new selection. Wrapping the input in a form allows the user to press Enter to submit.

btw: Type carefully. There’s no error handling. If you type a subreddit that doesn’t exist, the app will blow up. Implementing error handling would be a great exercise though! ;)

We could've used just 1 piece of state here – to store the input, and send the same value down to `Reddit` – but then the `Reddit` component would be fetching data with every keypress.

The `useState` at the top might look a little odd, especially the second line:

```jsx
const [inputValue, setValue] = useState("reactjs");
const [subreddit, setSubreddit] = useState(inputValue);
```

We're passing an initial value of "reactjs" to the first piece of state, and that makes sense. That value will never change.

But what about that second line? What if the initial state *changes*? (and it will, when you type in the box)

Remember that `useState` is stateful (read more about useState). It only uses the initial state *once*, the first time it renders. After that it's ignored. So it's safe to pass a transient value, like a prop that might change or some other variable.

A Hundred And One Uses

The `useEffect` function is like the swiss army knife of hooks. It can be used for a ton of things, from setting up subscriptions to creating and cleaning up timers to changing the value of a ref.

One thing it's *not* good for is making DOM changes that are visible to the user. The way the timing works, an effect function will only fire *after* the browser is done with layout and paint – too late, if you wanted to make a visual change.

For those cases, React provides the `useMutationEffect` and `useLayoutEffect` hooks, which work the same as `useEffect` aside from when they are fired. Have a look at the docs for useEffect and particularly the section on the timing of effects if you have a need to make visible DOM changes.

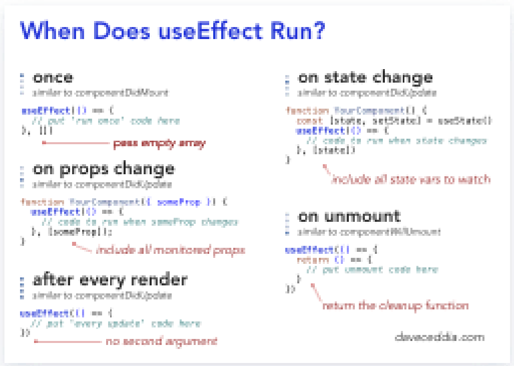
This might seem like an extra complication. Another thing to worry about. It kinda is, unfortunately. The positive side effect of this (heh) is that since `useEffect` runs after layout and paint, a slow effect won't make the UI janky. The down side is that if you're moving old code from lifecycles to hooks, you have to be a bit careful, since it means `useEffect` is almost-but-not-quite equivalent to `componentDidUpdate` in regards to timing.

Try Out useEffect

You can try `useEffect` on your own in this hooks-enabled CodeSandbox. A few ideas…

- Render an input box and store its value with `useState`. Then set the `document.title` in an effect. (like Dan's demo from React Conf)
- Make a custom hook that fetches data from a URL
- Add a click handler to the document, and print a message every time the user clicks. (don't forget to clean up the handler!)

If you're in need of inspiration, here is Nik Graf's Collection of React Hooks – currently at 440 and counting! Most of them are simple to implement on your own. (like `useOnMount`, which I bet you could implement based on what you learned in this post!)



When Does useEffect Run?

once
similar to componentDidMount

```
useEffect(() => {
  // put 'run once' code here
}, [])
```

pass empty array

on props change
similar to componentDidUpdate

```
function YourComponent({ someProp }) {
  useEffect(() => {
    // code to run when someProp changes
  }, [someProp]);
}
```

include all monitored props

after every render
similar to componentDidUpdate

```
useEffect(() => {
  // put 'every update' code here
})
```

no second argument

on state change
similar to componentDidUpdate

```
function YourComponent() {
  const [state, setState] = useState()
  useEffect(() => {
    // code to run when state changes
  }, [state])
}
```

include all state vars to watch

on unmount
similar to componentWillUnmount

```
useEffect(() => {
  return () => {
    // put unmount code here
  }
})
```

return the cleanup function

daveceddia.com

I put together a cheatsheet with different use cases for useEffect, and how they match up with lifecycle methods! Drop your email to get it delivered to your inbox.

Your email address

Get the Cheatsheet

Learning React can be a struggle — so many libraries and tools!
My advice? Ignore all of them :)
For a step-by-step approach, check out my Pure React workshop.

Learn to think in React

90+ screencast lessons

Full transcripts and closed captions

All the code from the lessons

Developer interviews

Start learning Pure React now →

“ Dave Ceddia’s Pure React is a work of enormous clarity and depth. Hats off. I'm a React trainer in London and would thoroughly recommend this to all front end devs wanting to upskill or consolidate.

Alan Lavender
@lavenderlens

An internal error has occurred. If you see this repeatedly, please contact support.