The useEffect Hook

(https://github.com/learn-co-curriculum/react-hooks-use-effect) (https://github.com/learn-co-curriculum/react-hooks-use-effect/issues/new)

Learning Goals

- Understand side effects in programming
- Use the `useEffect` hook to write side effects in components
- Control when the side effects run by using a dependencies array with `useEffect`

Introduction

In this lesson, we'll talk about how to use **side effects** in our function components with the `useEffect` hook, and how to run additional code in our components that isn't triggered by a user event such as clicking a button.

Reviewing What We Know

Here's a quick recap of some of the key concepts we've learned about React components:

- A **component** is a function that takes in **props** and returns **JSX**
- When we call `ReactDOM.render` and pass in our components, it will **render** all of our components by calling our component functions, passing down props, and building the DOM elements out of our components' JSX
- When a React app's **state** is updated by calling the `setState` function, React will **re-render** the component, along with all of its children

Side Effects

When we think about the functionality of a function, we generally think about its return value. However, functions can also have **side effects**:

> an operation, function or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, that is to say has an observable effect besides returning a value (the main effect) to the invoker of the operation. — **Wikipedia on Side Effects**

> (https://en.wikipedia.org/wiki/Side_effect_(computer_science)#:%7E:text=In%20computer%20science%2C%20an%20operation,the%20invoker%20of%20the%20operation.)

Put more simply, if we call a function and that function causes change in our application *outside of the function itself*, it's considered to have caused a **side effect**. Things like making network requests, accessing data from a database, writing to the file system, etc. are common examples of side effects in programming.

In terms of a React component, the **main effect** of the component is to return some JSX. That's been true with all of the components we've been working with! One of the first rules we learned about function components is that they take in props, and return JSX.

However, it's often necessary for a component to perform some **side effects** in addition to its main job of returning JSX. For example, we might want to:

- Fetch some data from an API when a component loads
- Start or stop a timer
- Manually change the DOM
- Subscribe to a chat service

In order to handle these kinds of side effects within our components, we'll need to use another special **hook** from React: `useEffect`.

Using the useEffect Hook

To use the `useEffect` hook, we must first import it:

```
import React, { useEffect } from "react";
```

Then, inside our component, we call `useEffect` and pass in a **callback function** to run as a **side effect**:

```
function App() {
  useEffect(
    // side effect function
    () => {
      console.log("useEffect called");
```

```
    }
  );

  console.log("Component rendering");

  return (
    <div>
      <button>Click Me</button>
      <input type="text" placeholder="Type away..." />
    </div>;
  )
}
```

If you run the example code now, you'll see the console messages appear in this order:

- Component rendering
- useEffect called

So we are now able to run some extra code as a **side effect** any time our component is rendered.

> 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. — **React docs on the useEffect hook (https://reactjs.org/docs/hooks-effect.html)**

Let's add some state into the equation, and see how re-rendering the component by updating state interacts with our `useEffect` hook:

```
function App() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("");

  useEffect(() => {
    console.log("useEffect called");
  });
```

```
  console.log("Component rendering");

  return (
    <div>
      <button onClick={() => setCount((count) => count + 1)}>
        I've been clicked {count} times
      </button>
      <input
        type="text"
        placeholder="Type away..."
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
    </div>
  );
}
```

Try clicking the button or typing in the input field to trigger updates in state. Every time state is set, we should also see those same two console messages in the same order:

- Component rendering
- useEffect called

By default, `useEffect` will run our side effect function every time the component re-renders.

```
render -> useEffect -> setState -> re-render -> useEffect
```

useEffect Dependencies

Sometimes we only want to run our side effect in certain conditions. For example: imagine we're using the `useEffect` hook to fetch some data from an external API (a common use case for `useEffect`). We don't want to make a network request every time our component is updated, only

the first time our component renders.

If we write a component that updates state from inside the `useEffect` callback, we'll see an issue:

```
function DogPics() {
  const [images, setImages] = useState([]);

  useEffect(() => {
    fetch("https://dog.ceo/api/breeds/image/random/3")
      .then((r) => r.json())
      .then((data) => {
        // setting state in the useEffect callback
        setImages(data.message);
      });
  });

  return (
    <div>
      {images.map((image) => (
        <img src={image} key={image} />
      ))}
    </div>
  );
}
```

Running this code will result in an endless loop of `fetch` requests (until the API kicks us out for hitting the rate limit 👀). We'd end up in a cycle like this:

```
render -> useEffect -> setImages -> render -> useEffect -> setImages -> render -> etc...
```

So how can we control when `useEffect` will run our side effect function?

React gives us a way to control when the side effect will run by passing a second argument to `useEffect`, a **dependencies array**. Let's set that up in our `App` component:

```
useEffect(
  // 1st arg: side effect (callback function)
  () => console.log("useEffect called"),
  // 2nd arg: dependencies array
  [count]
);
```

Now, if you try running the code again, the side effect will only run when the `count` variable changes. We won't see any console messages from `useEffect` when typing in the input — we'll only see them when clicking the button!

We can also pass in an *empty* array of dependencies as a second argument, like this:

```
useEffect(() => {
  console.log("useEffect called");
}, []); // second argument is an empty array
```

Now, the side effect will only run the *first time* our component renders! That same approach can be used to fix the infinite loop we saw in the fetch example as well:

```
useEffect(() => {
  fetch("https://dog.ceo/api/breeds/image/random/3")
    .then((r) => r.json())
    .then((data) => {
      setImages(data.message);
    });
}, []);
```

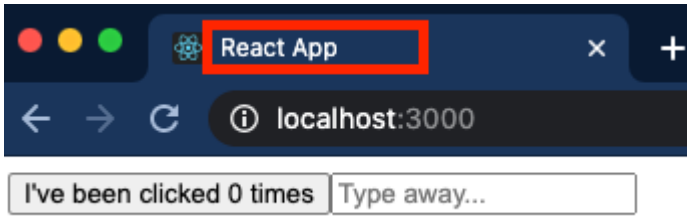
In this example, our component rendering cycle now looks like this:

```
render -> useEffect -> setImages -> render
```

Performing Side Effects

Running a `fetch` request as a side effect is one great example of when you'd use the `useEffect` hook and we'll explore that in more detail in the coming lessons. For now, let's take a look at a couple of other examples where you might use the `useEffect` hook.

One kind of side effect we can demonstrate here is *updating parts of the webpage outside of the React DOM tree*. React is responsible for all the DOM elements rendered by our components, but there are some parts of the webpage that live outside of this tree. Take, for instance, the `<title>` of our page — this is what shows up in the browser tab, like this:



Updating this part of the page would be considered a *side effect*, so let's use `useEffect` to update it!

```
useEffect(() => {
  document.title = text;
}, [text]);
```

Here, what we're telling React is:

"Hey React! When my component renders, I *also* want you to update the document's title. But you should only do that when the `text` variable changes."

Let's add another side effect, this time running a `setTimeout` function. We want this function to *reset* the `count` variable back to 0 after 5 seconds. Running a `setTimeout` is another example of a side effect, so once again, let's use `useEffect`:

```
useEffect(() => {
  setTimeout(() => setCount(0), 5000);
}, []);
```

With this side effect, we're telling React:

"Hey React! When my App component renders, I also want you to set this timeout function. In 5 seconds, you should update state and set the count back to 0. You should only set this timeout function once, I don't want a bunch of timeouts running every time my component updates. kthxbye!"

All together, here's what our updated component looks like:

```
function App() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("");

  useEffect(() => {
    document.title = text;
  }, [text]);

  useEffect(() => {
    setTimeout(() => setCount(0), 5000);
  }, []);

  console.log("Component rendering");

  return (
    <div>
      <button onClick={() => setCount((count) => count + 1)}>
        I've been clicked {count} times
      </button>
      <input
        type="text"
```

```
          placeholder="Type away..."
          value={text}
          onChange={(e) => setText(e.target.value)}
        />
      </div>
    );
  }
```

Explore this code to familiarize yourself with `useEffect`, and see what changes by changing the dependencies array. It's also a good idea to add some console messages or put in a debugger to see exactly when the side effects will run.

useEffect Dependencies Cheatsheet

Here's a quick guide on how to use the second argument of `useEffect` to control when your side effect code will run:

- `useEffect(() => {})` : No dependencies array
  - Run the side effect **every time our component renders** (whenever state or props change)
- `useEffect(() => {}, [])` : Empty dependencies array
  - Run the side effect **only the first time our component renders**
- `useEffect(() => {}, [variable1, variable2])` : Dependencies array with elements in it
  - Run the side effect **any time the variable(s) change**

Or, to put it another way:


Ryan Florence

Conclusion

So far, we've been working with components solely for rendering to the DOM based on JSX, and updating based on changes to state. It's also useful to introduce **side effects** to our components so that we can interact with the world outside of the React DOM tree and do things like making network requests or setting timers.

Resources

- **React Docs on useEffect (https://reactjs.org/docs/hooks-effect.html)**
- **A Complete Guide to useEffect (https://overreacted.io/a-complete-guide-to-useeffect/)**