The useEffect Hook - Cleaning Up

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

Learning Goals

- Use a cleanup function with `useEffect` to stop background processes

Introduction

In the last lesson, we saw how to run functions as **side effects** of rendering our components by using the `useEffect` hook. Here, we'll discuss best practices when it comes to cleaning up after those functions so we don't have unnecessary code running in the background when we no longer need it.

useEffect Cleanup

When using the `useEffect` hook in a component, you might end up with some long-running code that you no longer need once the component is removed from the page. Here's an example of a component that runs a timer in the background continuously:

```
function Clock() {
  const [time, setTime] = useState(new Date());

  useEffect(() => {
    setInterval(() => {
      setTime(new Date());
    }, 1000);
  }, []);
  return <div>{time.toString()}</div>;
}
```

When the component first renders, the `useEffect` hook will run and create an interval. That interval will run every 1 second in the background, and set the time.

We could use this Clock component like so:

```
function App() {
  const [showClock, setShowClock] = useState(true);

  return (
    <div>
      {showClock ? <Clock /> : null}
      <button onClick={() => setShowClock(!showClock)}>Toggle Clock</button>
    </div>
  );
}
```

When the button is clicked, we want to remove the clock from the DOM. That *also* means we should stop the `setInterval` from running in the background. We need some way of cleaning up our side effect when the component is no longer needed!

To demonstrate the issue, try clicking the "Toggle Clock" button — you'll likely see a warning message like this:

```
index.js:1 Warning: Can't perform a React state update on an unmounted
component. This is a no-op, but it indicates a memory leak in your application.
To fix, cancel all subscriptions and asynchronous tasks in a useEffect cleanup
function.
```

The reason for this message is that even after removing our `Clock` component from the DOM, the `setInterval` function we called in `useEffect` is **still running in the background**, and updating state every second.

React's solution is to have our `useEffect` function **return a cleanup function**, which will run when the component is unmounted, i.e., when it is no longer being returned by its parent.

Here's how the cleanup function would look:

```
function Clock() {
  const [time, setTime] = useState(new Date());

  useEffect(() => {
    const timerID = setInterval(() => {
      setTime(new Date());
    }, 1000);

    // returning a cleanup function
    return function cleanup() {
      clearInterval(timerID);
    };
  }, []);

  return <div>{time.toString()}</div>;
}
```

If you run this app again in the browser, and click the "Toggle Clock" button, you'll notice we no longer get that error message. That's because we have successfully cleaned up after the unmounted component by running `clearInterval` .

Cleanup Function Lifecycle

So far, we've explained the order of operations for our components like this:

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

Where does the cleanup function fit in this order of operations? In general, it is called by React **after the component re-renders** as a result of setting state and **before the `useEffect` callback is called**:

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

If the change (as in our example) causes the component to be unmounted, the cleanup is the last thing that occurs in the component's life:

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

Here's a way to visualize the different parts of the component lifecycle:

React Hooks Lifecycle

You can also check out this **CodeSandbox (https://codesandbox.io/s/react-hooks-lifecycle-wbgz1)** example to visualize the component lifecycle. The code includes a series of calls to `useEffect` that log messages to the console. If you open the browser console, you can see the sequence of events that happens when the page first loads. Then, if you try clicking the buttons, the order of the logged messages will show the order in which the different stages occur. Note that, for each component (`Parent` and `Child`), there are three different `useEffect` calls: one with no dependencies array, one with an empty array, and one with `count` as a dependency. This enables you to see when `useEffect` and `cleanup` are called for each of the three options.

Conclusion

Cleanup functions like this are useful if you have a long-running function that you want to unsubscribe from when the component is no longer on the page. Common examples include:

- a timer running via `setInterval`
- a subscription to a web socket connection

You don't always have to use a cleanup function as part of your `useEffect` code, but it's good to know what scenarios make this functionality useful.

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/)**