Title: How Effect hook works?

Answer:

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.

Example Using Hooks

```
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. 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.

Tags: react-hooks