**useEffect Hook-II**

In React, components have a lifecycle that consists of different phases. Each phase has a set of lifecycle methods that are called at specific points in the component's lifecycle. These methods allow you to control the component's behavior and perform specific actions at different stages of its lifecycle.

A component's lifecycle has three main phases: the Mounting Phase, the Updating Phase, and the Unmounting Phase.

The Mounting Phase begins when a component is first created and inserted into the DOM. The Updating Phase occurs when a component's state or props change. And the Unmounting Phase occurs when a component is removed from the DOM.

**Component Mounting Phase**

🔄 **Lifecycle Phases:**

1. **Mounting Phase:**
   * **Creation & Insertion:** Begins when a component is initially created and added to the DOM.
   * **Lifecycle Methods:**
     + **useState()** and **useEffect()** hooks play a pivotal role in managing state and side effects during this phase.
2. **Updating Phase:**
   * **State/Props Alteration:** Occurs when a component's state or props undergo changes.
   * **Lifecycle Methods:**
     + **useEffect()** hook is vital for handling side effects triggered by state or props modifications.
3. **Unmounting Phase:**
   * **Removal from DOM:** Takes place when a component is eliminated from the DOM.
4. **Lifecycle Methods:**
   * **useEffect()** hook can be leveraged with cleanup functions to manage resources before unmounting.

🌟 **Functional Components' Perspective:** In functional components, these phases are managed primarily through the efficient utilization of React hooks such as **useState()** and **useEffect()**. These hooks enable developers to encapsulate stateful logic and side effects seamlessly within the functional component paradigm.

React’s useEffect cleanup function saves applications from unwanted behaviors like memory leaks by cleaning up effects. In doing so, we can optimize our application’s performance.

To follow along with this article, you should have a basic understanding of what useEffect is, including using it to fetch APIs. This article will explain the cleanup function of the useEffect Hook and, by the end of this article, you should be able to use the cleanup function comfortably.

## What is the useEffect cleanup function?

As the name implies, useEffect cleanup is a function [in the useEffect Hook](https://blog.logrocket.com/guide-to-react-useeffect-hook/) that allows us to tidy up our code before our component unmounts. When our code runs and reruns for every render, useEffect also cleans itself up using the cleanup function.

The useEffect Hook is designed to allow the return of a function within it, which serves as a cleanup function. The cleanup function prevents memory leaks — a situation where your application tries to update a state memory location that no longer exists — and removes unnecessary and unwanted behaviors.

Note that you don’t update the state inside the return function either:

useEffect(() => {

effect

return () => {

cleanup

}

}, [input])

## Why is the useEffect cleanup function useful?

As previously stated, the useEffect cleanup function helps developers clean effects that prevent unwanted behaviors, thereby optimizing application performance.

However, it is important to note that the useEffect cleanup function does not only run when our component wants to unmount — it also runs right before the execution of the next scheduled effect.

In fact, after our effect executes, the next scheduled effect is usually based on the dependency array:

The `dependency` in the code below is an array

useEffect(callback, dependency)

Therefore, when our effect is dependent on our prop or whenever we set up something that persists, we have a reason to call the cleanup function.

Let’s look at this scenario: imagine we request the server to fetch a particular user’s information using the user’s id. Before the request is completed, we change our mind and try to make another request to get a different user’s information.

At this point, both fetch requests would continue to run even after the component unmounts or the dependencies change. This can lead to unexpected behavior or errors, such as displaying outdated information or attempting to update components that are no longer mounted.

So, it is necessary for us to abort the fetch using the cleanup function. That way, we prevent these memory leak-related issues in our application.

## When should we use the useEffect cleanup?

Let’s say we have a React component that fetches and renders data. If our component unmounts before our promise resolves, useEffect will try to update the state (on an unmounted component) and send an error that looks like this:

!<https://blog.logrocket.com/wp-content/uploads/2021/10/Warning-error.png>

To fix this error, we use the cleanup function. According to [React’s official documentation]([https://legacy.reactjs.org/docs/hooks-effect.html#:~:text=When exactly does,issues later below.)](https://legacy.reactjs.org/docs/hooks-effect.html#:~:text=When%20exactly%20does,issues%20later%20below.)), “React performs the cleanup when the component unmounts. However, effects run for every render and not just once. This is why React also cleans up effects from the previous render before running the effects next time.”

As a side note before we continue: useEffects can be made to run once by simply passing an empty array to the dependency list. When you provide an empty array as the dependency list for useEffect, it indicates that the effect does not depend on any values from the component’s state or props. As a result, the effect will only run once, after the initial render, and it won’t run again for subsequent renders unless the component is unmounted and remounted:

useEffect(() => {

// Effect implementation

}, []); // Empty dependency array indicates the effect should only run once

Now that we understand how to make useEffect run once, let’s get back to our cleanup function conversation.

# More About UseState

Setting a state variable will queue another render. But sometimes you might want to perform multiple operations on the value before queueing the next render. To do this, it helps to understand how React batches state updates.

# ****React batches state updates****

You might expect that clicking the “+3” button will increment the counter three times because it calls setNumber(number + 1) three times:

//App.js

import { useState } from 'react';

export default function Counter() {

const [number, setNumber] = useState(0);

return (

<>

<h1>{number}</h1>

<button onClick={() => {

setNumber(number + 1);

setNumber(number + 1);

setNumber(number + 1);

}}>+3</button>

</>

)

}

However, as you might recall from the previous section, [each render’s state values are fixed](https://react.dev/learn/state-as-a-snapshot#rendering-takes-a-snapshot-in-time), so the value of number inside the first render’s event handler is always 0, no matter how many times you call setNumber(1):

setNumber(0 + 1);setNumber(0 + 1);setNumber(0 + 1);

But there is one other factor at play here. **React waits until all code in the event handlers has run before processing your state updates.** This is why the re-render only happens after all these setNumber() calls.

This might remind you of a waiter taking an order at the restaurant. A waiter doesn’t run to the kitchen at the mention of your first dish! Instead, they let you finish your order, let you make changes to it, and even take orders from other people at the table.

This lets you update multiple state variables—even from multiple components—without triggering too many [re-renders.](https://react.dev/learn/render-and-commit#re-renders-when-state-updates) But this also means that the UI won’t be updated until after your event handler, and any code in it, completes. This behavior, also known as **batching,** makes your React app run much faster. It also avoids dealing with confusing “half-finished” renders where only some of the variables have been updated.

**React does not batch across multiple intentional events like clicks**—each click is handled separately. Rest assured that React only does batching when it’s generally safe to do. This ensures that, for example, if the first button click disables a form, the second click would not submit it again.

# ****Updating the same state multiple times before the next render****

It is an uncommon use case, but if you would like to update the same state variable multiple times before the next render, instead of passing the next state value like setNumber(number + 1), you can pass a function that calculates the next state based on the previous one in the queue, like setNumber(n => n + 1). It is a way to tell React to “do something with the state value” instead of just replacing it.

Try incrementing the counter now:

//App.js

import { useState } from 'react';

export default function Counter() {

const [number, setNumber] = useState(0);

return (

<>

<h1>{number}</h1>

<button onClick={() => {

setNumber(n => n + 1);

setNumber(n => n + 1);

setNumber(n => n + 1);

}}>+3</button>

</>

)

}

Here, n => n + 1 is called an **updater function.** When you pass it to a state setter:

1. React queues this function to be processed after all the other code in the event handler has run.
2. During the next render, React goes through the queue and gives you the final updated state.

setNumber(n => n + 1);setNumber(n => n + 1);setNumber(n => n + 1);

Here’s how React works through these lines of code while executing the event handler:

1. setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.
2. setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.
3. setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.