**🔄 Building a Counter with React Hooks**

**🧮 Basic Counter with useState**

function Counter() {

const [count, setCount] = React.useState(0);

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increment</button>

<button onClick={() => setCount(count - 1)}>Decrement</button>

</div>

);

}

**🔄 Understanding useEffect**

**📝 Basic Syntax**

useEffect(() => {

// Effect code here

}, [/\* dependency array \*/]);

**🔧 useEffect Parameters**

* **Callback Function:** Code to run for the side effect
* **Dependency Array:** Controls when effect runs

**🔄 useEffect Patterns**

**1️⃣ Run Once (On Mount)**

function Counter() {

const [count, setCount] = React.useState(0);

useEffect(() => {

console.log("Component mounted");

}, []); // Empty dependency array

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**2️⃣ Run on State Change**

function Counter() {

const [count, setCount] = React.useState(0);

useEffect(() => {

console.log("Count changed:", count);

}, [count]); // Runs when count changes

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**3️⃣ Run on Every Render (No Dependency Array)**

function Counter() {

const [count, setCount] = React.useState(0);

useEffect(() => {

console.log("Component rendered");

}); // No dependency array

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**🔄 Advanced useEffect Examples**

**🧹 1. Cleanup Function**

function Counter() {

const [count, setCount] = React.useState(0);

useEffect(() => {

console.log("Effect started");

// Cleanup function

return () => {

console.log("Cleanup before next effect");

};

}, [count]);

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**📊 2. Multiple Effects**

function Counter() {

const [count, setCount] = React.useState(0);

const [isEven, setIsEven] = React.useState(true);

// Effect for count changes

useEffect(() => {

console.log("Count changed:", count);

}, [count]);

// Effect for even/odd status

useEffect(() => {

setIsEven(count % 2 === 0);

}, [count]);

return (

<div>

<h1>Counter: {count}</h1>

<p>Number is: {isEven ? 'Even' : 'Odd'}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**🔄 Common Use Cases**

**📡 1. Data Fetching**

function UserCounter() {

const [count, setCount] = React.useState(0);

const [data, setData] = React.useState(null);

useEffect(() => {

const fetchData = async () => {

try {

const response = await fetch('API\_URL');

const result = await response.json();

setData(result);

} catch (error) {

console.error("Error fetching data:", error);

}

};

fetchData();

}, []); // Fetch once on mount

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={() => setCount(count + 1)}>Increment</button>

{data && <div>Data: {JSON.stringify(data)}</div>}

</div>

);

}

**⏱️ 2. Side Effects with Cleanup**

function Counter() {

const [count, setCount] = React.useState(0);

useEffect(() => {

const timer = setInterval(() => {

setCount(c => c + 1);

}, 1000);

// Cleanup

return () => clearInterval(timer);

}, []); // Run once on mount

return <h1>Counter: {count}</h1>;

}

**🌟 Best Practices**

**📋 Dependency Array Management**

// ✅ Good

useEffect(() => {

document.title = `Count: ${count}`;

}, [count]);

// ❌ Avoid

useEffect(() => {

document.title = `Count: ${count}`;

}); // Missing dependency array

**🧹 Cleanup Functions**

useEffect(() => {

const subscription = someAPI.subscribe();

// Always cleanup subscriptions

return () => subscription.unsubscribe();

}, []);

**🔄 Async Operations**

useEffect(() => {

let isSubscribed = true;

const fetchData = async () => {

const result = await someAPI();

if (isSubscribed) {

// Only update if component is mounted

setData(result);

}

};

fetchData();

return () => {

isSubscribed = false;

};

}, []);

**💡 Remember:**

* Use empty dependency array for mount-only effects
* Include all dependencies used in the effect
* Always clean up subscriptions and timers
* Avoid infinite loops in effects
* Keep effects focused and simple

**🔄 Life cycle methods**

useEffect in React Hooks serves a similar purpose to the lifecycle methods found in class components (componentDidMount, componentDidUpdate, and componentWillUnmount), but it provides a more unified and streamlined way to handle side effects. Here's a detailed explanation of how useEffect corresponds to these lifecycle methods:

**1. componentDidMount**

This lifecycle method is called once in the lifecycle of a class component, immediately after it is inserted into the DOM. It's typically used for API calls, subscriptions, or any operations that should run once and set up.

**useEffect Equivalent:** To mimic componentDidMount using useEffect, you pass an empty dependency array [], which tells React that your effect does not depend on any values from props or state, so it never needs to re-run.

useEffect(() => {

console.log('Component did mount');

// Any setup that needs to happen once on mount

}, []); // The empty array makes this effect only run on mount

**2. componentDidUpdate**

This method is invoked immediately after updating occurs. This method is not called for the initial render. You might use this method for more DOM operations or when you need to make network requests in response to a change in props or state.

**useEffect Equivalent:** useEffect without specifying the second argument or with specific dependencies mimics this behavior. If you include a list of dependencies, the effect will only activate if the dependencies have changed between renders.

useEffect(() => {

console.log('Component did update');

// Actions to perform after updating

}, [dependencies]); // Only re-run the effect if dependencies change

**3. componentWillUnmount**

This lifecycle method is called right before a component is destroyed. It's used for cleanup, such as invalidating timers, canceling network requests, or cleaning up subscriptions.

**useEffect Equivalent:** The return function in useEffect is used for cleanup and is similar to componentWillUnmount. This function runs when the component is about to unmount, or before the effect runs again.

useEffect(() => {

// Setup phase

return () => {

console.log('Component will unmount');

// Cleanup code here

};

}, []); // The cleanup runs on unmount or before the effect re-runs

**Practical Example of Combining All Behaviors**

Here is an example that combines all behaviors to illustrate how a single useEffect can be used to handle component mount, update, and unmount scenarios:

import React, { useState, useEffect } from 'react';

function ExampleComponent(props) {

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

useEffect(() => {

// This runs on mount

console.log('Component did mount');

// This runs on update when count changes

console.log(`Count has updated to: ${count}`);

// Cleanup function

return () => {

console.log('Component will unmount or before effect re-runs');

};

}, [count]); // Depend on count

return (

<div>

<p>{count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

In this example, the effect handles the component's lifecycle events by logging messages on mount, updating the count, and cleaning up before unmounting or before the effect runs again. This demonstrates the powerful and flexible capabilities of useEffect compared to traditional class component lifecycle methods.