React

What is a Side Effect

A side effect occurs in a program whenever you use external code in your function — which, as a result, impacts the function’s ability to perform its task

* Dependency on external source
* Modifies external code
* Becomes a non-deterministic function

Impure Function

an impure function is a function that contains one or more side effects.

Pure Function

* A pure function is a function without any side effects.
* Pure functions are independent
* The function must be predictable

Hooks

useState

The React useState Hook allows you to have state variables in functional components. You pass the initial state to this function, and it returns a variable with the current state value (not necessarily the initial state) and another function to update this value.



useEffect

The useEffect hook is a function in React that allows developers to perform side effects in a functional component. This can include things like data fetching, setting up subscriptions, responding to the component's lifecycle events, or updating the DOM in response to changes in state or props.

The useEffect react hook is called after every render and takes a callback function as an argument, which contains the code for the side effect.



useRef

useRef in react returns a mutable reference object that doesn't trigger re-rendering of the component when they change and persist throughout their lifetimes. This object has a current property that holds a mutable value.

* The useRef React hook allows us to save values between renders.
* It stores a mutable value that does not cause a re-render when updated.
* It helps us to access a DOM element directly.



useCallback

The useCallback hook is a built-in hook in React that lets you memoize a callback function by preventing it from being recreated on every render. In simple terms, it means that the callback function is cached and does not get redefined on every render. This will optimize and improve the overall performance of your application.



useMemo

The useMemo is a hook used in the functional component of react that returns a memoized value. Memoization is a concept used in general when we don’t need to recompute the function with a given argument for the next time as it returns the cached result. A memoized function remembers the results of output for a given set of inputs.



So what is the difference?

**useCallback returns its function uncalled** so you can call it later,

while **useMemo calls its function and returns the result**.

useContext

React context provides data to components no matter how deep they are in the components tree. The context is used to manage global data, e.g. global state, theme

usage

* Creating the context
* Providing the context
* Consuming the context



useDebugValue

The useDebugValue hook is simply used to print out debug information for custom hooks. This is incredibly useful if you are creating a library for others to use since they can easily see information about your hook, but it also is useful for your own hooks since you can quickly see detailed information about your hooks.

This debug information is displayed within the React dev tools.



One thing most people worry about with adding debug code to their application is how it will slow down production. This is especially bothersome if the code for calculating the useDebugValue label is slow. Luckily, the React team thought of this which is why useDebugValue can take a second parameter which is a function that is only called when the hooks are actually inspected in the dev tools.

useDefferredValue

The useDeferredValue hook allows us to fix the slow render problem by implementing a delay before some information is calculated. This works in a very similar way to debouncing and throttling since our deferred value will only be calculated after the important state updates have finished running.



The difference is that useTransition() wraps the state updating code, whilst useDeferredValue() wraps a value that's affected by the state update. You don't need to (and shouldn't) use both together, since they achieve the same goal in the end.

useTransition

The useTransition hook allows us to specify some state updates as not as important. These state updates will be executed in parallel with other state updates, but the rendering of the component will not wait for these less important state updates.

Calling the useTransition hook returns an array with the first value being an isPending variable and the second value being the startTransition function. The isPending variable simply returns true while the code inside the startTransition hook is running. Essentially, this variable is true when the slow state update is running and false when it is finished running. The startTransition function takes a single callback and this callback just contains all the code related to the slow state update including the setting of the state.



useId

The main use case of the useId hook is to generate unique ids for use within HTML elements. The best example of this would be to create an id for an input and have a label point to the same id. For example, if you had an EmailForm component you could write it like so



This code would work, but if you try to render this form on the same page multiple times you will have multiple input elements with the same id of email. This is obviously bad since every id on a page should be unique and on top of that your labels when clicked on will now all focus the first email input on the page instead of the email input associated with that label. To fix this we can use useId.

The useId hook is a simple hook that takes no inputs and returns a unique id. This id is unique to each individual component which means we can render this form on our page as many times as we want without having to worry about duplicate ids



useImperativeHandle

useImperativeHandle is generally used whenever we need a child function component to expose some of its functionalities to the parent component. For instance, maybe the function component will encapsulate some logic that might be useful in another part of the parent component.



useLayoutEffect

In useEffect the code in the hook is run asynchronously after React renders the component. This means the code for this hook can run after the DOM is painted to the screen.

The useLayoutEffect hook runs synchronously directly after React calculates the DOM changes but before it paints those changes to the screen. This means that useLayoutEffect code will delay the painting of a component since it runs synchronously before painting, while useEffect is asynchronous and will not delay the paint.

So if useLayoutEffect will delay the painting of a component why would we want to use it. The biggest reason for using useLayoutEffect is when the code being run directly modifies the DOM in a way that is observable to the user.

For example, if I needed to change the background color of a DOM element as a side effect it would be best to use useLayoutEffect since we are directly modifying the DOM and the changes are observable to the user. If we were to use useEffect we could run into an issue where the DOM is painted before the useEffect code is run. This would cause the DOM element to be the wrong color at first and then change to the right color due to the useEffect code.

 useReducer Hook

The useReducer Hook is used to store and update states, just like the useState Hook. It accepts a reducer function as its first parameter and the initial state as the second. useReducer returns an array that holds the current state value and a dispatch function to which you can pass an action and later invoke it.



useInsertionEffect

The signature of useInsertionEffect is identical to useEffect. But it fires synchronously before all DOM mutations.

It should be used for inserting global DOM nodes like <style> or SVG <defs> before reading layout in useLayoutEffect. It is not really meant to be used by anything else other than these CSS libraries.

useInsertionEffect is for CSS-in-JS library authors. Unless you are working on a CSS-in-JS library and need a place to inject the styles, you probably want useEffect or useLayoutEffect instead.