**1.Explain why would one use a useCallback() hook?**

The useCallback hook is used when you have a component in which the child is re-rendering again and again without need.

Pass an inline callback and an array of dependencies. useCallback will return a memorized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

**Syntax:**

const memorizedCallback = useCallback(

() => {

doSomething(a, b);

},

[a, b],

);

It depends on referential equality. In JavaScript, functions are first-class citizens, meaning that a function is a regular object. Hence, two function objects even when they share the same code are two different objects. Just remember that a function object is referentially equal only to itself.

**When to use useCallback :** In React, a component usually has some callback function created within it.

**function** MyComponent(){

    // HandleChange is created on every render

    const handleChange = () => {...};

**return** <>

        ...

        </>;

}

**Here handleChange**function objects are different on every rendering of MyComponent. And there are several cases when we may want the same function object between multiple renderings. For example when it’s a dependency for some other hooks (useEffect( …, callbackfunc)) or when the function object itself has some internal state that we need to maintain. In such a case, useCallback hook comes handy. In simple words, **useCallback( callBackFun, deps )** returns a memorized callback when the dependency values **deps** do not change between renderings. (memorized here refers to caching the object for future use).

**2. What happens to the function passed to useCallback() hook?**

A functional component wrapped inside React.memo() accepts a function object prop

When the function object is a dependency to other hooks, e.g. useEffect(..., [callback])

When the function has some internal state, e.g. when the [function is debounced or throttled](https://dmitripavlutin.com/react-throttle-debounce/#2-debouncing-a-callback-the-first-attempt).

That’s when useCallback(callbackFun, deps) is helpful: given the same dependency values deps, the hook returns the same function instance between renderings (aka memorization):

**3. Explain in your own words what does the preventDefault() method do and when do we use it?**

The preventDefault() method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.

For example, this can be useful when:

* Clicking on a "Submit" button, prevent it from submitting a form
* Clicking on a link, prevent the link from following the URL

Note: Not all events are cancellable. Use the cancellable property to find out if an event is cancellable.

Note: The preventDefault() method does not prevent further propagation of an event through the DOM. Use the stopPropagation() method to handle this.

The preventDefault() method is used to prevent the browser from executing the default action of the selected element. It can prevent the user from processing the request by clicking the link.

Syntax :

event.preventDefault()

Parameter**:**

It does not accept any parameter.  
The **event** is used to denote the event or action by the user in the response of which the method works.

**4. What happens if I call a function declaration in the dependency array of a useEffect(), but do not wrap the function in a useCallback() hook?**

In the simplest terms, useEffect is a hook that allows you to perform side effects in functional components. For even more detail, these effects are only executed after the component has rendered, therefore not blocking the render itself.

**So I should use the useEffect hook on side effects?**

Whew, yes. You should. That was a long list of reasons you shouldn’t place it inline with the component, but I hope those examples help you understand why this was built. useEffect allows you to:

* not block the UI
* creates a visual “block” of code that is a visible effect.
* keeps your functions pure (or tries to)
* increases the readability of your code
* is easy to extract when needed to custom hooks, so that we can share amongst other components

### Approach #1: Move the function into the effect

**Usecase**: This fetchData call is only ever used in this local useEffect.

If you plan on ever only using this function in this single useEffect, the most straightforward and suggested solution is to move the function directly into the effect closure. This works for everything we discussed previous, and it ensures that our effect function itself is as pure and referentially transparent as possible. It encapsulates the logic to one area and also lets developers know this function is intended as a side effect. So what does that look like

function **ExampleComponent**({url}) {

**useEffect**(() => {

const **fetchData** = (url) => {

*// fetch call here*

}

**fetchData**(url)

}, [url]);

return (<div></div>);

}

Since the fetchData function is now part of our effect, it is no longer a dependency of our effect, and we can simply remove it from the dependency array.

### Approach #2: Memorize the function with useCallback

**Usecase:** This function is used in multiple local hooks or is going to be passed down in a child component

useCallback is one of the new hooks available to React. It allows us to memorize a function so that on subsequent updates of the component, the function keeps its referential equality, and therefore does not trigger the effect. useCallbacks use the same dependency array that a useEffect does, so if the values or functions it depends on change, it will be reinitialized. To understand how this works, I think it would be useful to jump into Referential and Value Equality.