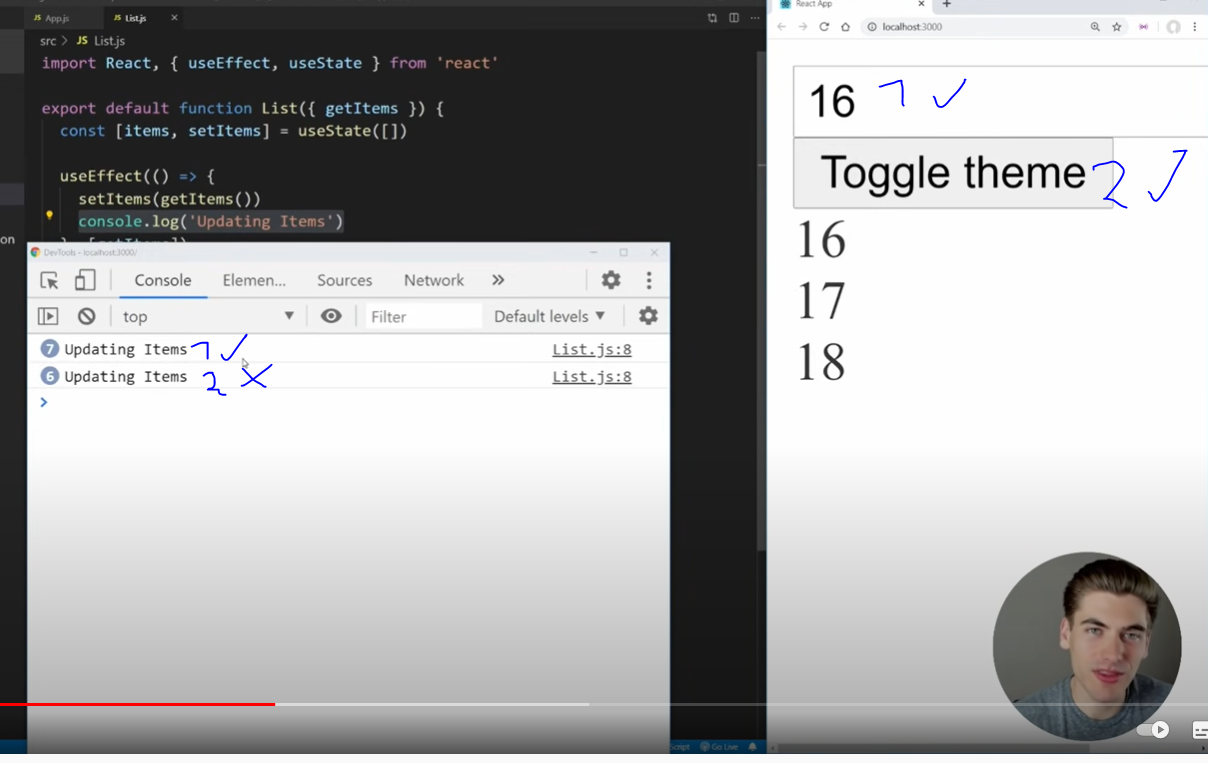
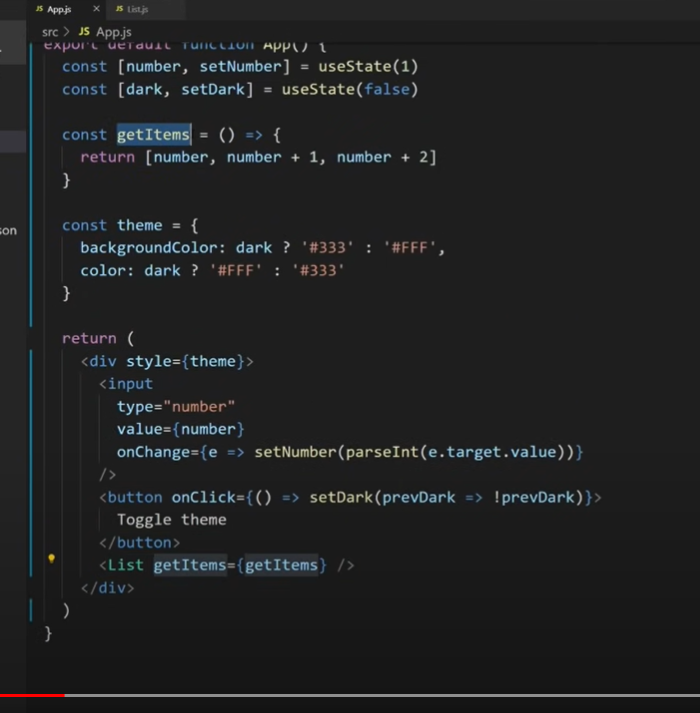
**Learn useCallback In 8 Minutes**

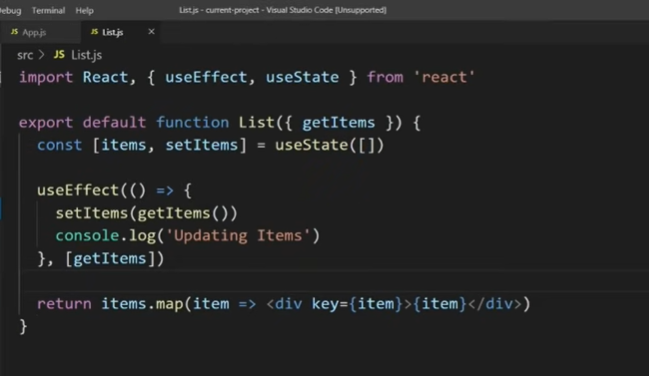
Link: <https://www.youtube.com/watch?v=_AyFP5s69N4>

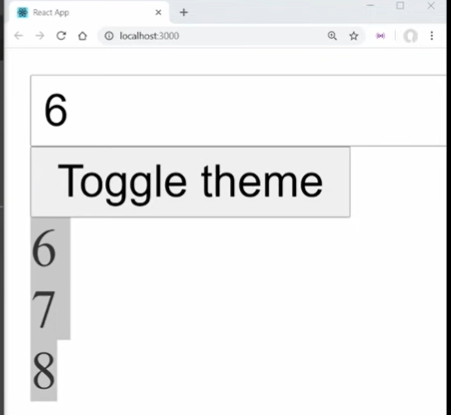


Según esta App básica, el componente updatea los ítems cada vez que cambia la función getItems.

getItems cambia cada vez que cambiamos el input y el número.





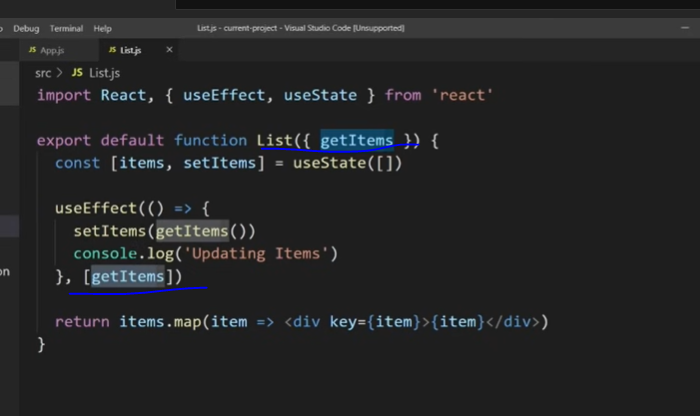


Notar lo siguiente:

**Cada vez que cambiemos el input (números) o hagamos el toggle del theme, se repintará el componente dado que cambia una pieza de state.**

La dependencia getItems se vuelve a ejecutar. Entonces, el console.log se producirá, no solo en el caso que se cambie el input, sino también en el caso que haga el Toggle del Theme.

**Nuestra getItems function es re-creada cada vez que se repinta el componente, y por eso se ejecuta el console.log también cuando hacemos el click en toggle theme, dado que el cambio de estado produce un repintado del componente.**



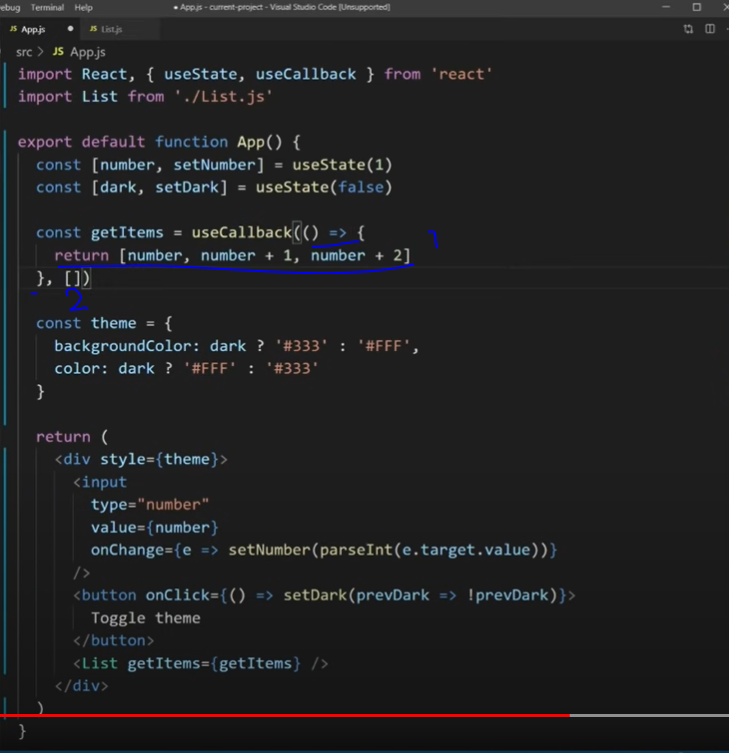
**Dado que pasamos la función como prop, cada vez que se repinta el componente App, la función getItems es una nueva función getItems.**

**Esto es un problema que se puede resolver utilizando el useCallback hook.**

**useCallback:**

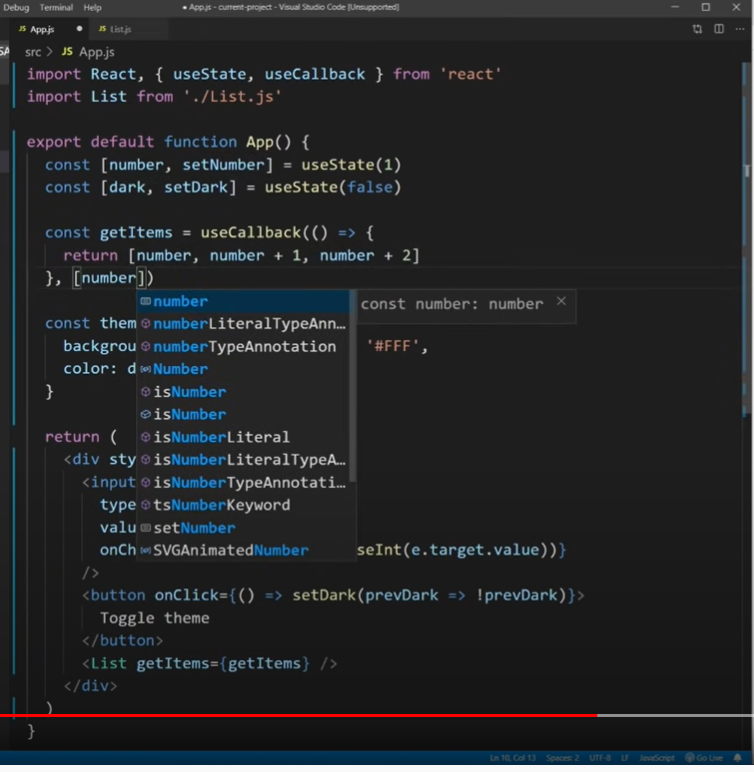
useCallback es muy parecido a useMemo.

No corre nuevamente el código adentro del useCallback, **a menos que ciertos parámetros cambien.** **Esto quiere decir que cada vez que se repiten el App component, la función getItems cambiará solo cuando lo necesite. Es decir, cuando el número cambie.**

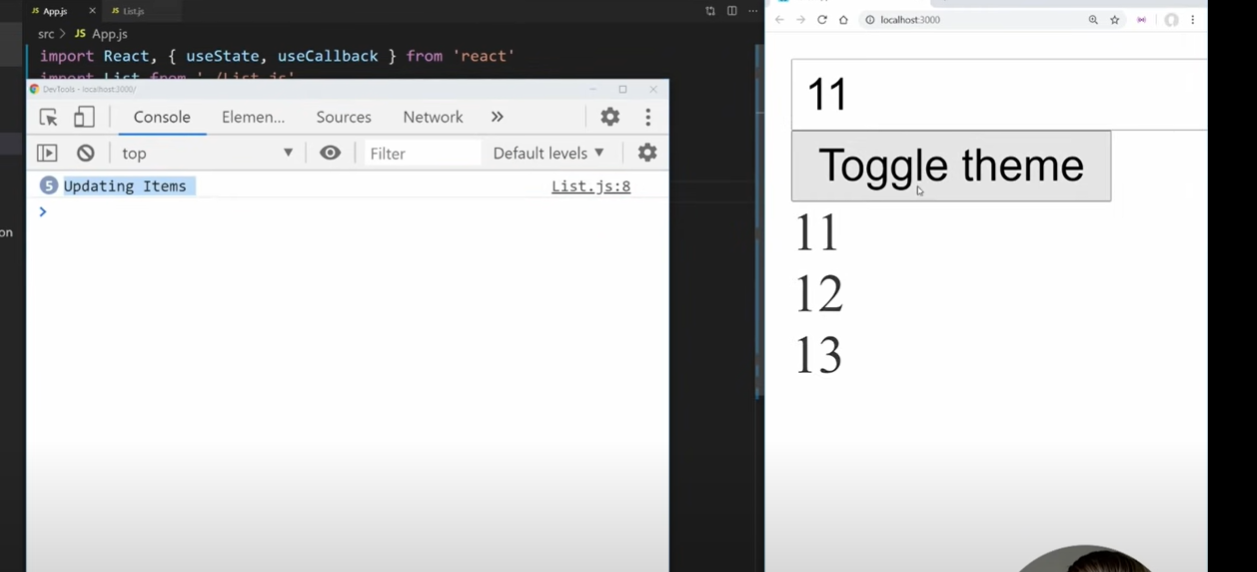


useCallback tiene 2 parámetros:

* Callback function que queremos que se ejecute solo cuando cambia el parámetro de la función
* Arreglo de dependencias.



En ese caso, hacemos el useCallback() para que dependa de number.



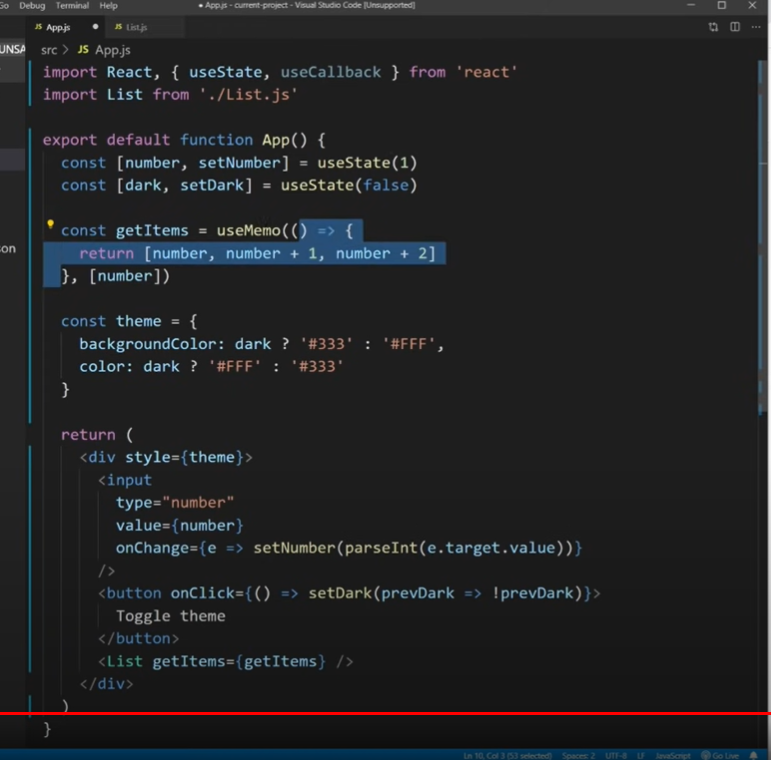
Como podemos ver, updateItems solo se consologea en el caso que cambie el estado de numbers.

**useCallback() solo recrea nuestra función getItems cuando cambia el number state.**

**Diferencia entre useMemo y useCallback:**

useMemo toma como parámetro una callback function, y te va a retornar el return value de esa función.

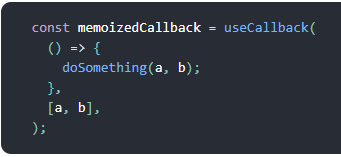
useCallback es diferente. Toma como parámetro una callback function, pero retorna esa misma función (en vez del return value de esa función).



Si hubiéramos usado useMemo en vez de useCallback, getItems se setearía al array simplemente, y no a la función.

**useCallback**

Link: <https://es.reactjs.org/docs/hooks-reference.html#usecallback>



Devuelve un callback memorizado.

**Pasa un callback en línea y un arreglo de dependencias.** **useCallback devolverá una versión memorizada del callback que solo cambia si una de las dependencias ha cambiado.**

**Esto es útil cuando se transfieren callbacks a componentes hijos optimizados que dependen de la igualdad de referencia para evitar renders innecesarias (por ejemplo, shouldComponentUpdate).**



El arreglo de dependencias no se pasa como argumentos al callback. Sin embargo, conceptualmente, eso es lo que representan: **cada valor al que se hace referencia dentro del callback también debe aparecer en el arreglo de dependencias. En el futuro, un compilador lo suficientemente avanzado podría crear este arreglo automáticamente.**

Recomendamos usar la regla exhaustive-deps que forma parte de nuestro paquete eslint-plugin-react-hooks. Esta regla advierte cuando las dependencias se especifican incorrectamente y sugiere una solución.

**Your Guide to React.useCallback()**

Link: <https://dmitripavlutin.com/dont-overuse-react-usecallback/>

**He said his teammates, no matter the situation, were wrapping every callback function inside useCallback():**



**"Every callback function should be memoized to prevent useless re-rendering of child components that use the callback function" is the reasoning of his teammates.**

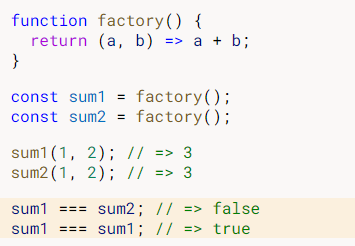
**This reasoning is far from the truth. Such usage of useCallback() without profiling makes the component slower.**

**1. Understanding functions equality check**

Before diving into useCallback() usage, let's distinguish the **problem useCallback() solves — the functions equality check**

Functions in JavaScript are first-class citizens, meaning that a function is a regular object. The function object can be returned by other functions, be compared, etc.: anything you can do with an object.

Let's write a function factory() that returns functions that sum numbers:



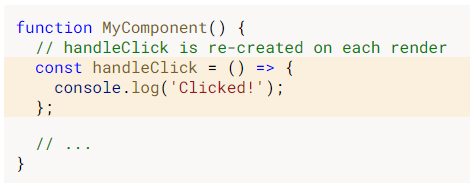
sum1 and sum2 are functions that sum two numbers. They've been created by the factory() function.

The functions sum1 and sum2 share the same code source but they are different function objects. Comparing them sum1 === sum2 evaluates to false.

**That's just how JavaScript objects works. An object (including a function object) equals only to itself.**

**2. The purpose of useCallback()**

**Different function objects sharing the same code are often created inside React components:**



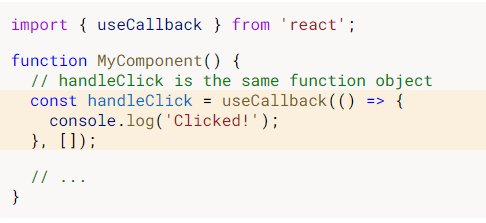
**handleClick is a different function object on every rendering of MyComponent.**

Because inline functions are cheap, the re-creation of functions on each rendering is not a problem. A few inline functions per component are acceptable.

**But in some cases you need to maintain a single function instance between renderings:**

1. A functional component wrapped inside React.memo() accepts a function object prop
2. **When the function object is a dependency to other hooks, e.g. useEffect(..., [callback])**
3. When the function has some internal state, e.g. when the function is debounced or throttled.

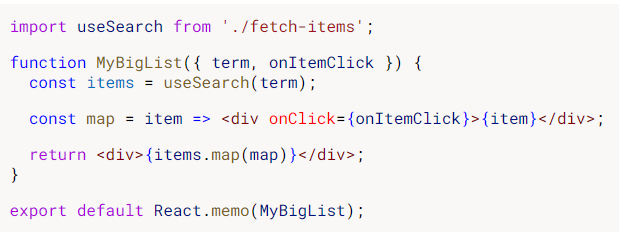
That's when useCallback(callbackFun, deps) is helpful: given the same dependency values deps, the hook returns the same function instance between renderings (aka memoization):



**handleClick variable has always the same callback function object between renderings of MyComponent.**

**3. A good use case**

Imagine you have a component that renders a big list of items:



The list could be big, maybe hundreds of items. To prevent useless list re-renderings, you wrap it into React.memo().

The parent component of MyBigList provides a handler function to know when an item is clicked:

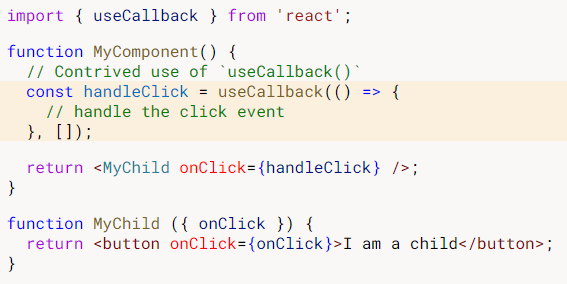
**onItemClick callback is memoized by useCallback(). As long as term is the same, useCallback() returns the same function object.**

**When MyParent component re-renders, onItemClick function object remains the same and doesn't break the memoization of MyBigList.**

That was a good use case of useCallback().

**4. A bad use case**

Let's look at another example:



**Does it make sense to apply useCallback()? Most likely not because <MyChild> component is light and its re-rendering doesn't create performance issues.**

Don't forget that useCallback() hook is called every time MyComponent renders. Even useCallback() returning the same function object, still, the inline function is re-created on every re-rendering (useCallback() just skips it).

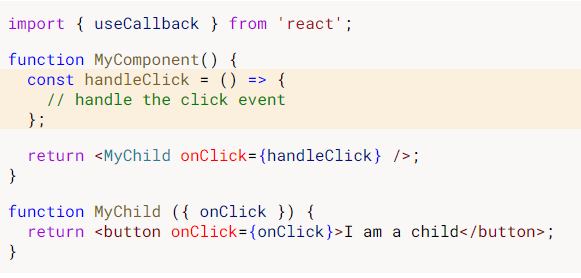
By using useCallback() you also increased code complexity. You have to keep the deps of useCallback(..., deps) in sync with what you're using inside the memoized callback.

In conclusion, the optimization costs more than not having the optimization.

**El arreglo de dependencias vacía significa que la función memoizada no se recreará nunca!**

(No cambiará en los re-renders).

Simply accept that rendering creates new function objects:



**5. Summary**

When deciding to use an optimization technique, including memoization and particularly useCallback(), do:

* First — profile
* Then quantify the increased performance (e.g. 150ms vs 50ms render speed increase)

Then ask yourself: does the increased performance, compared to increased complexity, worth using useCallback()?

To enable the memoization of the entire component output I recommend checking my post Use React.memo() wisely.

Voy por entender BlogPostCardSlider. Luego, seguir con Carousel de my-portfolio-app

**Puntos a indagar:**

* **UseCallback video de YouTube**
* **useCallback documentación**