

wawiwa

React

2022 © Wawiwa Tech | Confidential

This image shows a woman with dark hair, wearing a red and white patterned top, sitting at a desk and looking down at a computer screen. She is surrounded by multiple monitors displaying lines of code, suggesting she is working on a software development project. In the bottom left corner, there is a red arrow-shaped graphic containing the text "wawiwa". In the bottom right corner, the word "React" is written vertically. The background is slightly blurred, showing an office environment with other people and equipment. A small yellow sticky note with the word "DESIGN" is visible on the monitor on the right.

wawiwa

11 - Hooks

This slide introduces the concept of Hooks in React. It features a woman working on a laptop displaying code related to hooks. A red arrow-shaped logo for 'wawiwa' is positioned in the bottom-left corner of the slide.

Introducción

Los Hooks fueron agregados a React in 2018, en versión 16.8.

Los Hooks otorgan acceso al estado y otras funcionalidades de los componentes funcionales de React.

Como resultado, los componentes de clase generalmente ya no se necesitan. A pesar del hecho que los hooks generalmente reemplazan los componentes de clase, no existen planes para eliminar las clases de React.

Antes hemos aprendido del Hook de estado. Más Hooks van a ser cubiertos en este capítulo.

wawiwa

¿Qué es un Hook?

Hooks son funciones que te permiten “engancharte en” el **estado de React** y **funcionalidades del ciclo de vida** de los componentes funcionales.

React provee algunos Hooks incorporados como useState, useMemo, useState.

También puedes **crear tus propios Hooks** para reusar el comportamiento con estado entre los componentes distintos.

Usa Hooks siempre y solo en el nivel superior de la función React. No llames a los Hooks dentro de ciclos, condiciones o funciones anidadas. Siguiendo esta regla, puedes asegurar que tus Hooks sean llamados en el mismo orden cada vez que un componente se renderiza

Recuerda: antes de usar un Hook, necesitamos **importarlo**.

Hook de estado

wawiwa

`useState` permite agregar una variable de estado a tu componente.

La sintaxis:

```
const [state, setState] = useState(initialState);
```

state - el nombre del estado

setState - La función “set” retornada del `useState` te permite actualizar el estado a un valor diferente y dispara una re-renderización.

useState - un Hook.

initialState - el valor que quieres que esté en el estado inicialmente.

Hook de efecto

useEffect nos permite ejecutar efectos secundarios dentro de los componentes.

Ejemplos de los efectos secundarios:

- Traer datos
- Directamente actualizar el DOM
- Usar la función de temporizador setTimeout()

Hook de efecto

La sintaxis:

```
useEffect(() => { ... }, [dependencies]);
```

Una función que debe ejecutarse después de cada evaluación de componente si las dependencias especificadas cambiaron.

Un arreglo lleno de dependencias y cuando cualquiera de estas dependencias cambie, esa primera función va a correr de nuevo.

Puedes poner cualquier código secundario que sea ejecutado cuando las dependencias cambien, y **no** cuando el componente se re-renderize.

Hook de efecto - Sin dependencias pasadas

wawiwa

¿Qué pasará si renderizamos el componente? Se imprime cada vez que la página se carga. **useEffect corre en cada renderización**. Eso quiere decir que cuando el conteo cambia, una renderización ocurre, que después dispara otro efecto.

```
import React, { useState, useEffect } from "react";
function App() {
  const [count, setCounter] = useState(0);
  useEffect(() => {
    console.log("Hello World");
  });
  return (
    <>
      {count}
      <button onClick={() => {
        setCounter(count + 1);
      }}>Click Me
      </button>
    </>
  );
}
```



export default App; //wa Tech | Confidential

Hook de efecto - Dependencias - Un arreglo vacío

Para controlar cuando los efectos secundarios corren, incluimos un segundo parámetro. Pasamos las dependencias bajo un arreglo.

En el caso de un arreglo vacío:

```
useEffect(() => {  
  console.log("Hello World");  
}, []);
```



Corre solamente una primera renderización

Hook de efecto - Con Dependencias

En caso de un arreglo con un valor:

```
const [count, setCount] = useState(0);  
useEffect(() => {  
  console.log("Hello World");  
}, [count]);
```

El callback va a ejecutar el código cada vez que el valor de nuestra variable de estado cambie.

Conclusión de Hook de efecto - Dependencias

1. Sin pasar dependencias:

```
useEffect(() => {  
  // corre con cada renderización  
});
```

1. Un arreglo vacío:

```
useEffect(() => {  
  // corre solamente con la primera renderización  
}, []);
```

1. Props o valores del estado:

```
useEffect(() => {  
  //Corre en la primera renderización  
  //Y cada vez que cualquier dependencia cambia  
, [prop, state]);
```

Ejercicio 9

wawiwa

Crea un contador de segundos.

- El estado “seconds” inicializado a 0
- Cada segundo (1000 milisegundos) el estado “seconds” incrementa en 1
- Usa los hooks useState y useEffect
- Usa setInterval
- Usa la función clearInterval para limpiar el intervalo cuando el componente sea desmontado.

← → ⌂ ⓘ localhost:3000
Second counter: 1

← → ⌂ ⓘ localhost:3000
Second counter: 2

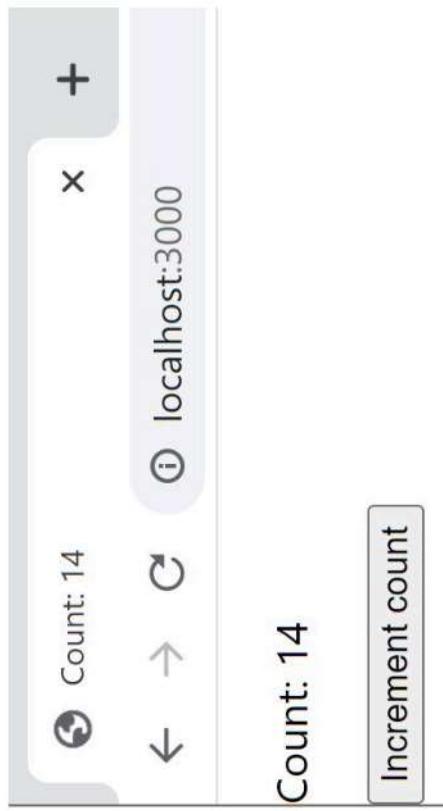
Ejercicio 10

wawiwa

Crea un app que renderice un contador y agrega un botón para incrementarlo.

Actualiza el título del documento con la cuenta actual.

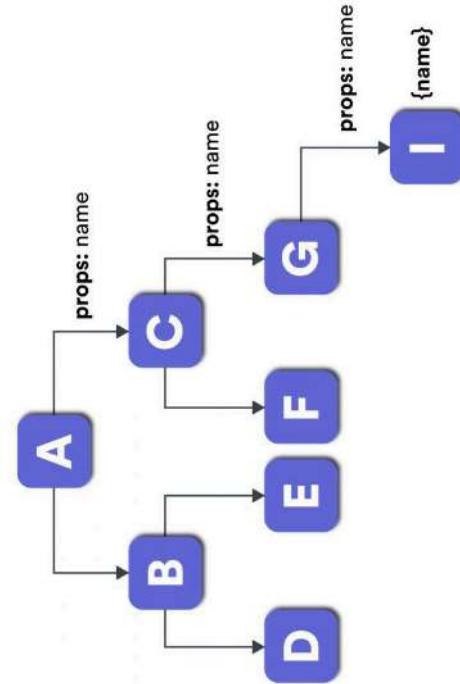
El estado "count" inicializado a 0.



Hook de contexto

Hasta ahora, para pasar información o un estado entre componentes, teníamos que pasarlo como "props" mediante componentes anidados. En esta manera pasamos información sin que se use en esos componentes. Por ejemplo:

```
import React, { useState } from "react";
function App() {
  const [name, setName] = useState("James");
  return (
    <>
      <Component1 name={name}>/>
    </>
  )
}
function Component1(props) {
  return (
    <Component2 name={props.name}>/>
  );
}
function Component2(props) {
  return (
    <div>Name: {props.name}</div>
  );
}
export default App;
```



The result:

Name: James

Hook de contexto

wawiwa

Para pasar y usar estados entre componentes usamos useContext. React Context es una manera de controlar el estado **globalmente**.

Los props habilitan que la información sea pasada de un componente padre a su hijo directamente. La información puede ser pasada de un componente padre a cualquier componente hijo anidado usando Context. Puedes usar el API de Context para especificar cuál información debe estar disponible para todos los componentes anidados en este context.

La sintaxis:

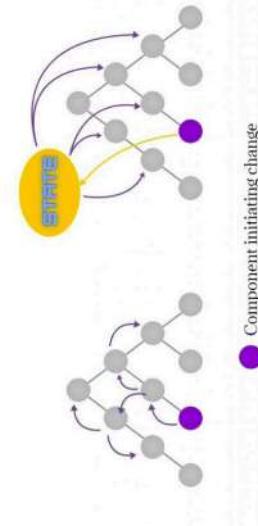
```
const nameContext = useContext(initialValue);
```

Hook de contexto - ¿Cómo funciona?

React Context te permite obtener tu información sin tener que pasártela manualmente.

Para hacer esto, envuelve el componente padre que está en la parte superior del árbol con un **Provider** (Proveedor). Todos los componentes hijos pueden conectarse al Provider, accediendo a los props sin tener que pasar por todos los componentes intermedios. Estos componentes hijos se les llama **Consumers** (Consumidores).

without
Context



Hook de contexto

wawiwa

¿Cómo se usa?

1. Crea el contexto
2. Provee el contexto
3. Consuma el contexto

1. Crea el contexto

Primero, crea un archivo para el contexto:

JS userContext.js

Importa createContext al nivel más alto de tu componente:

```
import React, {createContext} from "react";
```

La función incorporada **createContext(value)** crea un contexto:

```
const UserContext = createContext();
```

Exporta el contexto para usarlo con el consumidor:

```
export default UserContext;
```

wawiwa

2. Provee el contexto

Context.Provider se usa para proveer el contexto a sus componentes hijo, independientemente de su profundidad.

- Configura el proveedor en el componente padre (el más alto).
- Usa el prop de “value” para darle el valor que tu quieras.

Importa el contexto que has creado:

```
import UserContext from "./UserContext";
```

Agrega UserContext.Provider como un componente y configura un valor (nuestro estado global):

```
<UserContext.Provider value={"Daniel"}>  
  // Child components  
</UserContext.Provider>
```

3. Consume el contexto

Podemos consumir el contexto de 2 maneras:

1. Función para renderizar usando Context.Consumer

```
import React, { useContext } from "react";
import UserContext from "./userContext";

function App() {
  function Component1() {
    return (
      <>
        <Component2 />
        <UserContext.Consumer>
          {value => <p>Hey {value}</p>}
        </UserContext.Consumer>
      </>
    );
  }
}

function Component2() {
  return (
    <>
      <UserContext.Provider value="Daniel">
        <Component1 />
      </UserContext.Provider>
    </>
  );
}
```

1. Usa useContext(Contextname) - la recomendada

```
function Component2() {
  const value = useContext(UserContext);
  return (
    <p>Good morning, {value}!</p>
  );
}
```

Good morning, Daniel!
Hey Daniel



¿Cuándo, por qué y por qué usamos Context?

Context resuelve el problema de pasar props en componentes anidados.

Usamos el contexto para **usar información global y re-renderizar** cuando la información global es cambiada.

Ejemplos:

- La elección del usuario: Idioma de preferencia o tema
- Estado global
- Detalles del usuario (por ejemplo username)
- Autenticación

Debemos de pensar **antes de integrar contexto** porque **agrega complejidad** (crear el contexto y envolver en Provider usando useContext() cada consumidor).

Ejercicio 11

wawiwa

Usa la función createContext para crear un nuevo React context. Este es el “User Context”. En el User Context, define un objeto “user” inicial con las propiedades “name”, “email” y “age”.

Crea un componente UserProvider que envuelve al componente UserDisplay. Contiene información del usuario.

User Data

Name: John Doe

Email: john.doe@example.com

Age: 25

useContext con useState

wawiwa

El hook useContext en React te permite acceder y actualizar el estado compartido a través de componentes múltiples.

Aquí hay un ejemplo de cómo utilizar para manejar un estado “num” y su función setNum correspondiente, permitiendo a los componentes leer y modificar el valor de “num” con un contexto compartido:

Componente NumDisplay:

Componente Padre:

```
const [ num, setNum ] = useState(0);

return (
  <div>
    <h2>{num}</h2>
    <button onClick={() => setNum(num + 1)}>+</button>
  </div>
);
```

```
const { num, setNum } = useContext(NumContext);

return (
  <h2>{num}</h2>
  <button onClick={() => setNum(num + 1)}>+</button>
);
```

Ejercicio 12

wawiwa

Crea un nuevo componente llamado EditUser.

El componente permite al usuario actualizar su información con campos de entrada para "name", "email" y "age".

User Data

Name: John Doe

Email: john.doe@example.com

Age: 35

User Form

John Doe	john.doe@example.com	35
----------	----------------------	----

useReducer

wawiwa

useReducer nos ayuda a controlar lógicas de estado complejas en React. Es usado para guardar y actualizar estados (como el hook useState).

La diferencia entre useState y useReducer:

- useState - mejor para estados interdependientes no complejos. useState es más corto.
- useReducer - mejor para estados complejos y conectados. Úsalo para crear actualizaciones más complejas que solamente configurar un valor nuevo del estado.

useReducer es recomendado porque retorna un método “dispatch” que no cambia entre re-renderizaciones, y podemos hacer una lógica de manipulación.

useReducer

Sintaxis:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

El hook retorna un arreglo de:

1. El estado actual - El estado retornado por useReducer
2. La función “dispatch” - Una función usada para actualizar el estado. La lógica de estado personalizada

El hook acepta dos parámetros:

1. La función reductora - la función toma dos argumentos:
 - a. El estado previo
 - b. Una acción que es un objeto que contiene información usada para actualizar el estado
2. El valor con el que el estado es inicializado. Puede un valor simple pero usualmente contiene un objeto

Ejemplo de useReducer

Queremos crear un app contadora. Vamos a crear el estado con la función reductora que maneja los métodos.

Etapas:

1. Declara el estado inicial
2. Declara un estado
3. Declara la función reductora
4. Crea la app visual con una etiqueta con el conteo y botones

Counter: 0

<input style="width: 40px; height: 40px;" type="button" value="+"/>	<input style="width: 40px; height: 40px;" type="button" value="-"/>	<input style="width: 40px; height: 40px;" type="button" value="Reset"/>
---	---	---

Ejemplo de useReducer

wawiwa

1. Declara el estado inicial:

```
const initialState = { count: 0 }
```

Inicializa un estado con 0

1. Declara un estado:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

1. Declara la función reducida:

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 }
    case 'decrement':
      return { count: state.count - 1 }
    case 'reset':
      return { count: state.count = 0 }
    default:
      return { count: state.count }
  }
}
```

Crea la función reducida que acepta el
estado actual del conteo y una acción.

El estado se actualiza con el reducer basado en el tipo de
acción.

"increment", "decrement" y "reset" son los tipos de acción.
Cuando se despachan, el estado cambia.

Ejemplo useReducer

4. Agrega el la etiqueta del contador y botones:

```
return (
  <div>
    Counter: {state.count}
    <br />
    <button onClick={() => dispatch({ type: 'increment' })}>+</button>
    <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
  </div>
)
```

Ejemplo de useReducer - Código completo

```
import React, { useReducer } from "react";
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 }
    case 'decrement':
      return { count: state.count - 1 }
    case 'reset':
      return { count: state.count = 0 }
    default:
      return { count: state.count }
  }
}

function App() {
  const initialState = { count: 0 }
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      Counter: {state.count}
      <br />
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  )
}

export default App;
```

useReducer - Más ejemplos

Inicio de sesión del usuario:

```

function loginReducer(state, action) {
  switch (action.type) {
    case 'field':
      return {
        ...state,
        [action.fieldname]: action.payload,
        b: ''
      }
    case 'login':
      return {
        ...state,
        error: '',
        isLoading: true,
        b: ''
      }
    case 'success':
      return {
        ...state,
        isLoggedIn: true,
        isLoading: false,
        b: ''
      }
    case 'error':
      return {
        ...state,
        error: 'Incorrect username or password',
        isLoggedIn: false,
        isLoading: false,
        username: '',
        password: '',
        b: ''
      }
    case 'logout':
      return {
        ...state,
        isLoggedIn: false,
        b: ''
      }
    default:
      return state;
  }
}

```

useReducer - Más Ejemplos

Cargando información:

```
const reducer = (state, action) => {
  switch (action.type) {
    case "getData":
      return {
        ...state,
        isLoading: true,
      };
    case "getDataSuccess":
      return {
        ...state,
        isLoading: false,
        data: action.payload,
      };
    case "getDataFailure":
      return {
        ...state,
        isLoading: false,
      };
    default:
      return state;
  };
};
```

Ejercicio 13

wawiwa

Crea una app calculadora:

- La app tiene los siguientes botones:
 $+1$, $+5$, -1 , -5 , $/2$, $/5$, **Reset(0)**
- La app tiene botones "Show" y "Hide" para la etiqueta "counter"

Introducción:

- El estado inicial contiene:
 - a. Counter (0)
 - b. Show (true)
- Tiene 4 tipos de acciones

20

$+1$	$+5$	Reset	-1	-5	$/2$	$/5$
Show	Hide					

Hidden

$+1$	$+5$	Reset	-1	-5	$/2$	$/5$
Show	Hide					

useRef

useRef es un hook de React que retorna un objeto que puedes usar durante todo el ciclo de vida de tu componente, entre **re-renderizaciones**.

- El uso principal para el hook useRef es **acceder un hijo del DOM directamente**.
- También puede ser muy útil para mantener un valor mutable que no cause una re-renderización cuando se actualiza.

Recuerda importar useRef de react:

```
import { useRef } from 'react';
```

La sintaxis:

```
const myRef = useRef(initialValue);
```

El nombre El valor inicial

useRef

useRef() retorna un objeto llamado "current".

El uso más común de useRef es **referenciar elementos HTML**. Referenciar a Input (elemento de HTML) ejemplos:

Enfocar un input:

```
const inputRef = useRef(null);
const handlePrint = () => {
  inputRef.current.focus();
}

return (
  <>
  <input ref={inputRef} type="text"/>
  <button onClick={handlePrint}>Focus input</button>
  </>
)
```

Obtener el valor de un input

```
const inputNameRef = useRef(null);
const handlePrint = () => {
  alert(inputNameRef.current.value);
}

return (
  <>
  <input ref={inputNameRef} type="text"/>
  <button onClick={handlePrint}>Print Name</button>
  </>
)
```

useRef

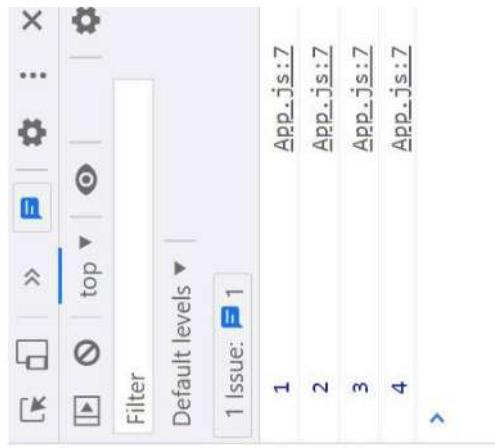
wawiwa

useRef no dispara una re-renderización (como cuando usamos useState).

Ejemplo del Counter:

```
const counter = useRef(0);
const handleIncrease = () => {
  counter.current = counter.current + 1;
  console.log(counter.current);
}

return (
  <>
    <h3>Value: {counter.current}</h3>
    <button onClick={handleIncrease}>
      Increase counter
    </button>
  </>
)
```



Ejercicio 14

wawiwa

Construir un input de texto simple que incluya un botón "Clear".

- Cuando el botón es presionado, quieres despejar el input de texto y enfocarlo para que el usuario pueda empezar a escribir otra vez.
- Esa el hook useRef para guardar una referencia al elemento de input de texto para que puedas acceder a sus propiedades y métodos directamente.

useMemo

useMemo es otro hook React que me ayude a optimizar el rendimiento del componente retornando un valor memorizado. Nos permite “recordar” un valor calculado entre renderizaciones.

La sintaxis:

```
const cachedValue = useMemo(calculateValue, dependencies)
```

useMemo acepta 2 argumentos:

1. Una función “compute” que calcula el resultado
2. El arreglo de dependencias
 - Bueno para mejorar el rendimiento
 - La función useMemo corre solamente cuando es necesario - cuando uno de estas dependencias se actualiza

useMemo - ¿Cómo funciona?

wawiwa

```
const cachedValue = useMemo(calculateValue, dependencies)
```

- En su inicialización, useMemo() invoca el cálculo, memoriza el resultado del cálculo y lo retorna en el componente.
- Durante las siguientes renderizaciones, si las **dependencias no cambian**, entonces useMemo() **no invoca el cálculo** pero retorna el valor memorizado.
- Si las **dependencias cambian** durante una re-renderización, entonces useMemo() **invoca el cálculo**, memoriza el valor nuevo y lo retorna.

useMemo - ¿Cuándo lo usamos?

Debes usar useMemo() cuando:

- Tengas un cálculo complejo que consume tiempo y quieres evitar re-calcularlo en cada renderización. (Por ejemplo, si necesitas filtrar una gran lista de objetos o contar el mismo valor con una diferencia pequeña).
- El cálculo depende de valores que vienen de props o de un estado que no cambia frequentemente.
- Quieres mejorar el rendimiento de tu componente reduciendo el número de re-renderizaciones innecesarias.

useMemo - Ejemplo

```
const calculateFactorial = (num) => {
  return num <= 0 ? 1 : num * calculateFactorial(num - 1);
}

const [number, setNumber] = useState(5);
```

```
const factorial = useMemo(() => { calculateFactorial(number) }, [number]);
```

```
return (
  <div>
    <h3>Factorials Calculator</h3>
    <input
      type="number"
      defaultValue={number}
      onChange={(e) => setNumber(e.target.value)}
    /> factorial is {factorial}
  </div>
)
```

- En este ejemplo, useMemo() es usado para guardar el resultado de calcular el número factorial de "number" a "factorial". El cálculo se hace solamente cuando cambia "number".

useCallback

useCallback es un hook de React que retorna una versión memorizada de una función **callback**.

La función no es recreada en cada renderización, solamente cuando una de las dependencias cambia.

Se usa para optimizar el rendimiento de tu componente React evitando la recreación de la misma función en cada renderización.

wawiwa useCallback - La sintaxis

```
const memoizedCallback = useMemo(calculateValue, dependencies);

const memoizedCallback = useCallback(
  () => {
    // aquí tu código
  },
  [dep1, dep2, ...], // Lista de dependencias
);
```

useCallback toma dos argumentos:

1. La función que quieres memorizar (Esta función puede ser cualquier función válida de JavaScript).
2. Un arreglo de dependencias que son valores que el hook usa para determinar si se debe actualizar el callback memorizado (si no tienes ninguna dependencia, pasa un arreglo vacío []).

El hook va a recrear la función memorizada si alguna de sus dependencias ha cambiado.

useCallback - Ejemplo

wawiwa

```
const calculateSum = (num) => {
  let counter = 0;
  for (let index = 1; index <= num; index++) {
    counter += index;
  }
  console.log("calculate sum");
  return counter;
}

const [inc, setInc] = useState(0);
const sum = useMemo(() => { calculateSum(100) }, []);
return (
  <div>
    <h3>Sum: {sum}</h3>
    <button onClick={()=>{setInc((i)=>i+1)}}>Re-render</button>
  </div>
)
```

wawiwa

useCallback - Ejemplo de re-renderización usando useState()

```
const calculateSum = (num) => {
  let counter = 0;
  for (let index = 1; index <= num; index++) {
    counter += index;
  }
  console.log("calculate sum");
  return counter;
}

const [inc, setInc] = useState(0);
// const sum = useMemo(() => { calculateSum(100) }, []);
const [sum, setSum] = useState(calculateSum(100));
useMemo() a useState()

return (
  <div>
    <h3>Sum: {sum}</h3>
    <button onClick={()=>{setInc((i)=>i+1)}}>Re-render</button>
  </div>
)
```

Ahora trata de cambiar useState() a useMemo()

useMemo VS useCallback

Ambos hooks son similares.

La diferencia principal es que:

- useMemo retorna un valor memorizado
- useCallback retorna una función memorizada