

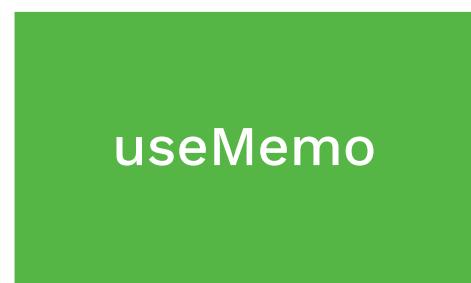
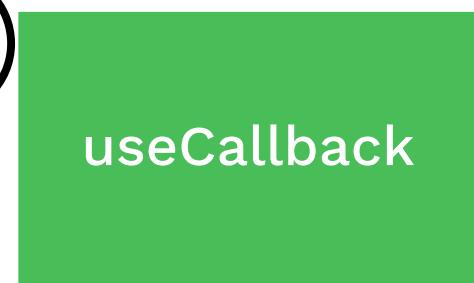


Hooks en React

Centro de Servicios y Gestión Empresarial
SENA Regional Antioquia

Concepto - Hooks

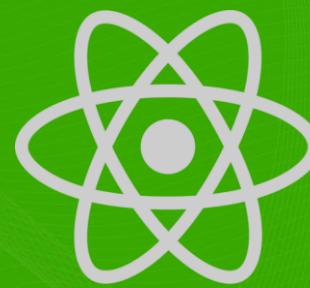
Hooks son funciones que permiten usar estado y otros aspectos de React sin escribir una clase. Los Hooks son un enlace que permiten enganchar los componentes funcionales a todas las características que ofrece React.



Importancia - Hooks

Los hooks han transformado la manera en que se manejan el estado y los efectos secundarios en React, haciéndolos más intuitivos y fáciles de usar en componentes funcionales.

- Los componentes funcionales manejan su ciclo de vida a través de useEffect, en lugar de los métodos de ciclo de vida de clase.
- El Virtual DOM mejora el rendimiento al minimizar las actualizaciones directas al DOM real.
- Los hooks son funciones que permiten usar el estado y otros aspectos de React en componentes funcionales, facilitando el desarrollo y la reutilización del código.



Tipos de Hooks

1. Hook - useState

Es el hook más básico y se usa para manejar el estado en componentes funcionales. Devuelve una variable de estado y una función para actualizarla.

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

// Sintaxis
const [state, setState] = useState(initialValue);
```

- **state:** La variable que representa el valor actual del estado.
- **setState:** La función que se usa para actualizar el valor de state.
- **initialValue:** El valor inicial del estado, puede ser un valor constante o una función que devuelva el valor inicial.

Hook – useState - ejemplo

```
// Sintaxis de useState
import React, { useState } from 'react';
const [sum, setSum] = useState(0);

// Como se utiliza useState
import React, { useState } from 'react';
const App = () => {
  const [sum, setSum] = useState(0);
  return (
    <div>
      <h1>La suma es: {sum}</h1>
      <button onClick={() => setSum(sum + 1)}>Incrementar</button>
    </div>
  );
};
```

2. Hook - useEffect

Permite realizar efectos secundarios en los componentes de React. Es útil para tareas como realizar llamadas a una API, configurar temporizadores o suscribirse a eventos. El efecto se ejecuta después de que el componente se ha renderizado.

`useEffect` puede manejar tres momentos clave del ciclo de vida del componente:

- **Montaje:** cuando el componente se monta en el DOM.
- **Actualización:** cuando cambian sus dependencias.
- **Desmontaje:** cuando el componente se elimina del DOM.

El comportamiento del hook se controla mediante el array de dependencias (`[dependencies]`): Si el array está vacío (`[]`), el efecto se ejecuta solo una vez al montar y se limpia al desmontar. Si se incluyen dependencias, el efecto se ejecutará cada vez que alguna de ellas cambie.

2. Hook - useEffect

```
// Importar React y useEffect desde 'react'
import React, { useEffect } from 'react';

// Definir el componente funcional App
const App = () => {
  useEffect(() => {
    // Código que se ejecuta al montar el componente
    return () => {
      // Código opcional que se ejecuta al desmontar el componente
    };
  }, // Dependencias que se ejecutan al cambiar los estados del componente
  []);
  return (
    // Código JSX del componente
  );
};
```

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

export function Contador() {
  const [contador, setContador] = useState(0);

  useEffect(() => {
    // Este código se ejecuta cuando el componente se monta o cuando 'contador' cambia
    console.log('El componente se montó o el contador cambió:', contador);
    return () => {
      // Este return se ejecuta cuando el componente se desmonta
      console.log('El componente se va a desmontar o el contador cambiará');
    };
  }, [contador]); // Dependencia: el efecto se ejecuta cuando 'contador' cambia

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={() => setContador(contador + 1)}>Incrementar</button>
    </div>
  );
}
```

2. Hook - useEffect - ejemplo

3. Hook - useContext

El hook `useContext` permite acceder a un contexto de React sin necesidad de utilizar explícitamente los componentes **Provider** y **Consumer**. Esto simplifica la forma en que los datos se comparten entre los componentes de una aplicación.

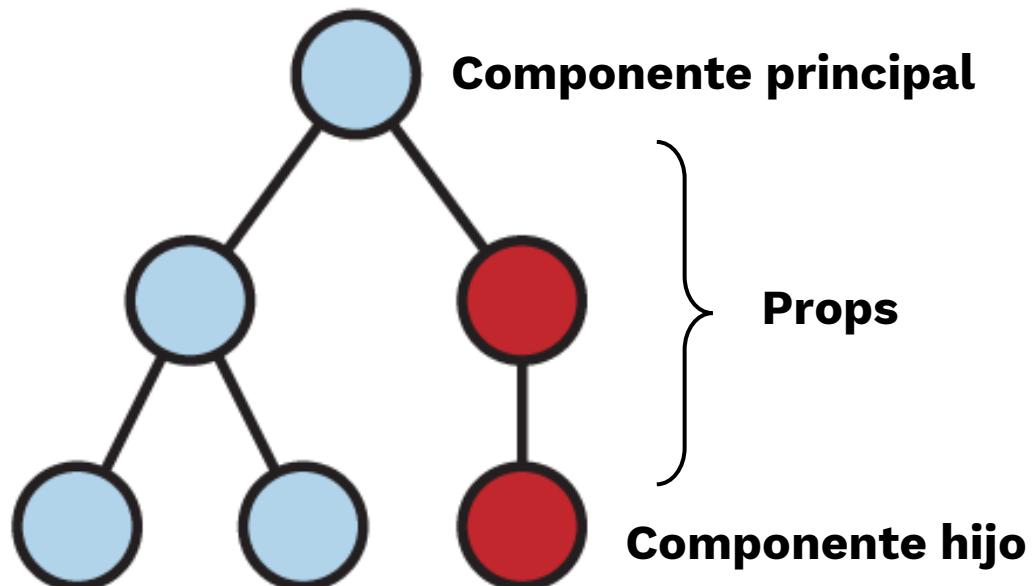
Gracias a `useContext`, es posible intercambiar información a lo largo del árbol de componentes de manera más eficiente, facilitando la gestión de un estado global sin necesidad de pasar props manualmente en cada nivel.

```
import React, { createContext, useContext } from 'react';

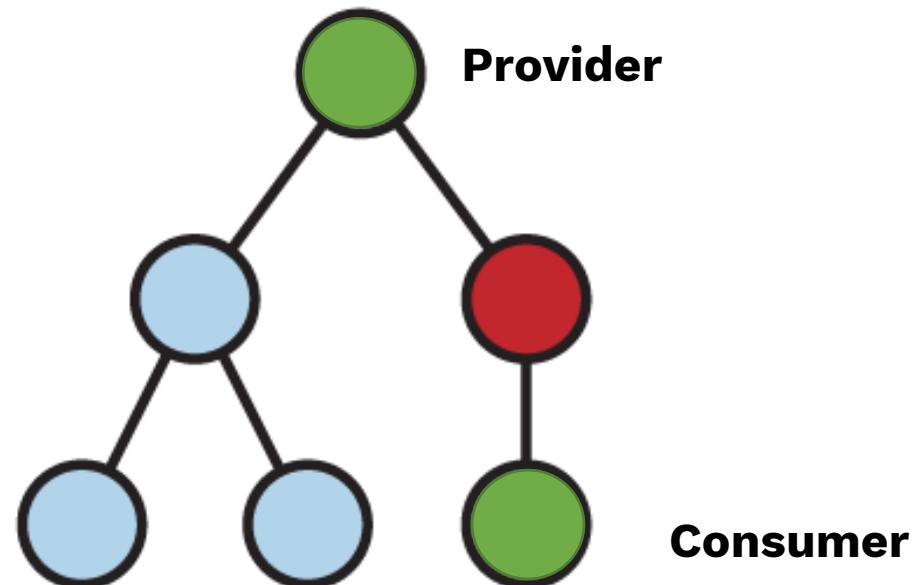
// Acceso al contexto
const value = useContext(MyContext);
```

3. Hook - useContext

Sin contexto



Con Contexto



Hook - useContext

Aunque useContext facilita el acceso al contexto, es indispensable envolver los componentes consumidores con un Provider. Este componente especial permite proveer un valor que estará disponible para todos los componentes descendientes que usen el contexto.

Se suele emplear para compartir datos como configuraciones, temas o información de autenticación.

```
// Provider - Sintaxis
<MyContext.Provider value={/* valor a compartir */}>
  {/* componentes que pueden acceder al contexto */}
</MyContext.Provider>
```

Hook - useContext

Beneficios:

- Elimina la necesidad de pasar props manualmente entre múltiples niveles de componentes.
- Centraliza el estado compartido.

Pasos a Seguir

- 1. Creación del Contexto:** Utiliza createContext para crear un nuevo contexto.
- 2. Uso del Provider:** Envuelve los componentes que necesitan acceder al contexto con el Provider y proporciona un valor que será accesible para ellos.
- 3. Consumo del Contexto:** Usa useContext dentro de los componentes hijos para acceder a los valores compartidos.

Hook - useContext

```
import React, { createContext, useContext } from 'react';

export const ThemeContext = createContext('light');

export function DisplayTheme() {
  const theme = useContext(ThemeContext);
  const icon = theme === 'light' ? '☀️' : '🌙';
  return (
    <p>Tema seleccionado es: <strong>{theme} - {icon}</strong></p>
  );
}

export function App() {
  return (
    <div>
      <ThemeContext.Provider value="dark">
        <DisplayTheme />
      </ThemeContext.Provider>

      <DisplayTheme />
    </div>
  );
}
```

Hook - useContext

Ventajas de Usar un Provider

- **Simplificación del Paso de Datos:** Permite que cualquier componente acceda a ciertos datos sin tener que pasarlo manualmente a través de cada componente en el árbol, lo que mejora la eficiencia del código.
- **Reusabilidad:** Puedes cambiar el valor del contexto de manera dinámica. Los componentes que dependen de ese contexto se actualizarán automáticamente, lo que facilita la gestión del estado.
- **Mantenibilidad:** Ayuda a organizar y separar los datos de configuración o temas comunes, lo cual contribuye a que el código sea más legible y fácil de mantener. Además, esta estructura fomenta una mayor claridad en la lógica del negocio.

4. Hook - useReducer

Es una alternativa a **useState** cuando se necesita una lógica de estado más compleja o cuando el próximo estado depende del anterior. Es similar a reduce en JavaScript y funciona muy bien para manejar múltiples valores de estado.

```
import React, { useReducer } from 'react';

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

- **reducer:** Una función que recibe el estado actual y una acción, y devuelve el nuevo estado.
- **initialState:** el estado inicial.
- **dispatch:** La función que dispara las acciones para modificar el estado.

```
import React, { useReducer } from 'react';

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



4. Hook - useReducer



```
export function CounterReducer() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const increment = () => dispatch({ type: 'increment' });
  const decrement = () => dispatch({ type: 'decrement' });

  return (
    <div>
      <button onClick={increment}>+</button>
      <p> {state.count} </p>
      <button onClick={decrement}>-</button>
    </div>
  );
}
```

4. Hook - useReducer

¿Cuándo usarlo?

- Cuando el estado tiene múltiples subvalores.
- Cuando las acciones que modifican el estado pueden agruparse en tipos (type).
- Cuando necesitas organizar mejor la lógica de actualización de estado.

Casos de uso típicos:

Escenario	Ejemplo
Formulario complejo	Manejo de múltiples campos, validación, errores.
Tabla con filtros	Estado con filtros, paginación, selección múltiple.
Gestión de entidades	Reducer con acciones tipo CREATE, UPDATE, DELETE.

5. Hook - useRef

Crea una referencia mutable que persiste durante el ciclo de vida del componente.

Es útil para acceder directamente a elementos del DOM o para almacenar un valor que no necesita causar una nueva renderización cuando cambia.

Sintaxis:

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

const refContainer = useRef(initialValue);
```

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

export function FocusInput() {
  const inputRefEmail = useRef(null);
  const inputRefPass = useRef(null);
  const handleFocus = () => {
    inputRefEmail.current.value = '';
    inputRefPass.current.value = '';
    inputRefEmail.current.focus();
  };

  return (
    <div>
      <label>Usuario:</label>
      <input type="email" defaultValue="aperezn@sena.edu.co" ref={inputRefEmail} />
      <br />
      <label>Contraseña:</label>
      <input type="password" defaultValue="123456" ref={inputRefPass} />
      <button onClick={handleFocus}>Restablecer</button>
    </div>
  );
}
```

5. Hook - useRef

5. Hook - useRef

¿Cuándo usarlo?

- Cuando necesitas una referencia al DOM (input, div, canvas, etc.).
- Para guardar valores entre renders sin causar re-render.
- Para medir tiempos, guardar valores previos, timeouts, etc.

Casos de uso típicos:

Escenario	Ejemplo
Enfocar automáticamente un input	Al abrir un modal de edición o crear.
Evitar re-render innecesario	Guardar un prevState, contador, timeout o un valor temporal.
Acceder al DOM	scrollIntoView(), medir alturas, manejar focus, etc.



G R A C I A S

Presentó: Alvaro Pérez Niño

Instructor Técnico

Correo: aperezn@misena.edu.co

<http://centrodesseriviciosygestionempresarial.blogspot.com/>

Línea de atención al ciudadano: 01 8000 910270

Línea de atención al empresario: 01 8000 910682



www.sena.edu.co