React: Hooks, Custom Hooks y Context

Diego Muñoz

09 de noviembre de 2025

Introducción

- Profundizar en los **hooks** de React.
- Entender su propósito y reglas.
- Crear **custom hooks** reutilizables.
- Introducir **Context** como solución al *prop drilling*.

¿Qué es un Hook?

- Función especial de React para usar **estado y ciclo de vida** en componentes funcionales.
- Todos comienzan con `use`.
- Ejemplos: `useState`, `useEffect`, `useContext`.
  Antes de los hooks (React 16.8), solo los componentes de clase podían tener estado.

Reglas básicas de los Hooks

1. Solo se usan en el **nivel superior** del componente.
2. Solo se llaman dentro de **componentes React** o **otros hooks**.
3. Deben llamarse siempre en el mismo orden.
   Esto permite que React asocie cada hook con su valor de estado correcto.

useState — Estado local

- Hook más básico: permite crear un valor **reactivo**.
- Al cambiar el estado, React vuelve a renderizar el componente.

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Valor: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

useState — Varios estados

- Puedes tener varios `useState` en el mismo componente.
- Cada uno mantiene su propio valor.

```jsx
function Form() {
  const [name, setName] = useState("");
  const [age, setAge] = useState(0);

  return (
    <div>
      <input value={name} onChange={e => setName(e.target.value)} />
      <input value={age} onChange={e => setAge(+e.target.value)} />
    </div>
  );
}
```

- Si el nuevo valor depende del anterior, usar **función de actualización**.

```
setCount(prev => prev + 1);
```

Esto evita errores cuando hay múltiples actualizaciones seguidas.

useEffect — Efectos secundarios

- Permite ejecutar código fuera del render: llamadas a APIs, timers, logs, etc.
- Se ejecuta después de que React renderiza el componente.

```
import { useEffect } from "react";

useEffect(() => {
  console.log("Componente montado");
  return () => console.log("Desmontado");
}, []);
```

useEffect — Dependencias

El segundo argumento (`[]`) indica **cuándo** ejecutar el efecto.

Comportamiento de Dependencias

- `[]` Solo al montar componente
- `[args]` Cada vez que `args` cambien
- Sin `[]` Cada ciclo de render

```
useEffect(() => {
  document.title = `Clicks: ${count}`;
}, [count]);
```

useEffect — Ejemplo práctico

```
function Users() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(r => r.json())
      .then(setUsers);
  }, []);

  return (
    <ul> {users.map(u => <li key={u.id}>{u.name}</li>)} </ul>
  );
}
```

Custom Hooks — Lógica reutilizable

- Un **custom hook** encapsula lógica que usa otros hooks.
- Permite compartir comportamiento entre componentes.

Custom Hooks — Ejemplo

```
function useToggle(initial = false) {
  const [value, setValue] = useState(initial);
  const toggle = () => setValue(v => !v);
  return [value, toggle];
}
function App() {
  const [open, toggleOpen] = useToggle();
  return (
    <>
      <button onClick={toggleOpen}> {open ? "Cerrar" : "Abrir"} </button>
      {open && <p>Contenido visible</p>}
    </>
  );
}
```

useContext — Evitar prop drilling

- Context permite **compartir datos globales** sin pasar props manualmente.
- Ideal para tema, idioma o usuario.

```
import { createContext } from "react";
export const ThemeContext = createContext();
```

Proveedor de contexto

```
function ThemeProvider({ children }) {
  const [dark, setDark] = useState(false);

  return (
    <ThemeContext.Provider value={{ dark, setDark }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

Consumir el contexto

```
function ToggleTheme() {
  const { dark, setDark } = useContext(ThemeContext);
  return (
    <button onClick={() => setDark(!dark)}>
      {dark ? "Modo claro" : "Modo oscuro"}
    </button>
  );
}
function App() {
  return (
    <ThemeProvider> <ToggleTheme /> </ThemeProvider>
  );
}
```

Estructura recomendada

```
src/
 ├─ components/
 ├─ hooks/
 │   └─ useToggle.js
 ├─ context/
 │   └─ ThemeContext.jsx
 └─ App.jsx
```

Organiza el código por **función y responsabilidad**.

Proyecto de ejemplo

Referir a ejemplo en GitHub.

Resumen

- `useState`: manejar estado local.
- `useEffect`: ejecutar efectos secundarios.
- `Custom hooks`: encapsular lógica reutilizable.
- `Context`: compartir datos globales.
- `useContext`: consumir el contexto.