React Hooks

Sistemas y Tecnologías Web (CC3062) - 2026

Hooks

Semestre 01, 2026

Definición

Antes de 2019, la lógica con estado solo existía en componentes de clase

- this.state, this.setState, componentDidMount, componentDidUpdate...
- La lógica se repetía en distintos componentes sin forma de reutilizarla
- Los ciclos de vida mezclaban código no relacionado

Los Hooks son funciones especiales que permiten usar estado y otras funciones de React en componentes funcionales

- Introducidos en React 16.8 (2019)
- Permiten extraer y reutilizar lógica entre componentes

React + Hooks = componentes más simples y poderosos.

Reglas

Dos reglas que siempre deben cumplirse:

1. Solo llamar Hooks en el nivel superior, nunca dentro de `if`, `for` o funciones anidadas

2. Solo llamar Hooks en componentes React, no en funciones JavaScript ordinarias

```
// MAL
function Componente() {
  if (condicion) {
    const [valor, setValor] = useState(0); // Rompe la regla 1
  }
}

// BIEN
function Componente() {
  const [valor, setValor] = useState(0);

  if (condicion) {
    // usar valor aquí
  }
}
```

hooks

useState

Permite agregar una variable de estado a un componente funcional

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

function Contador() {
  const [conteo, setConteo] = useState(0);

  return (
```

```
    <div>
      <p>Conteo: {conteo}</p>
      <button onClick={() => setConteo(conteo + 1)}>
        Incrementar
      </button>
    </div>
  );
}
```

- `useState(valorInicial)` retorna `[estado, setter]`
- Llamar al setter provoca un re-render del componente

Estado con objetos

El estado puede ser cualquier tipo de dato

```
function Formulario() {
  const [usuario, setUsuario] = useState({
    nombre: '',
    email: '',
  });

  return (
    <input
      value={usuario.nombre}
      onChange={(e) =>
        setUsuario({ ...usuario, nombre: e.target.value })
      }
    />
  );
}
```

- Al actualizar objetos, siempre crear una copia con spread (`...`)
- React compara referencias cambiar el objeto directamente no dispara el re-render

Estado con arrays

```
function ListaTareas() {
  const [tareas, setTareas] = useState([]);

  function agregarTarea(nueva) {
    setTareas([...tareas, nueva]); // nueva referencia
  }

  function eliminarTarea(id) {
    setTareas(tareas.filter(t => t.id !== id)); // nuevo array
  }

  return (/* ... */);
}
```

| Operación | Método correcto | |---|---| | Agregar | `[...array, nuevo]` | | Eliminar | `array.filter(...)` | | Modificar | `array.map(...)` |

Actualización basada en el estado anterior

```
function Contador() {
  const [conteo, setConteo] = useState(0);

  function incrementarDoble() {
    // MAL — puede usar valor desactualizado
    setConteo(conteo + 1);
    setConteo(conteo + 1);

    // BIEN — función updater garantiza el valor más reciente
    setConteo(prev => prev + 1);
    setConteo(prev => prev + 1);
  }
}
```

Usar la función updater `prev =>` cuando el nuevo estado depende del anterior

useEffect

Efecto es cualquier acción que ocurre fuera del render puro:

- Llamadas a APIs / fetch de datos
- Suscripciones a eventos
- Modificar el título de la página
- Timers (setTimeout, setInterval)
- Manipulación directa del DOM

Sintaxis de useEffect

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

useEffect(() => {
  // código a ejecutar (el efecto)
  return () => {
    // función de limpieza (opcional)
  };
}, [/* dependencias */]);
```

- Se ejecuta después de que React pinta la pantalla
- El array de dependencias controla cuándo se vuelve a ejecutar
- La función de retorno limpia recursos cuando el componente se desmonta

Variantes según dependencias

```
// 1. Sin array — ejecuta después de CADA render
useEffect(() => {
  console.log('Renderizado');
});
```

```
// 2. Array vacío — ejecuta solo al MONTAR el componente
useEffect(() => {
  console.log('Montado');
}, []);

// 3. Con dependencias — ejecuta cuando cambie `id`
useEffect(() => {
  console.log('El id cambió:', id);
}, [id]);
```

fetch de datos

```
function PerfilUsuario({ userId }) {
  const [usuario, setUsuario] = useState(null);

  useEffect(() => {
    fetch(`/api/usuarios/${userId}`)
      .then(res => res.json())
      .then(data => setUsuario(data));
  }, [userId]); // se vuelve a ejecutar si userId cambia

  if (!usuario) return <p>Cargando...</p>;

  return <h1>Hola, {usuario.nombre}</h1>;
}
```

Título del documento

```
function Tarea({ titulo }) {
  useEffect(() => {
    document.title = `${titulo}`;

    // Limpieza: restaurar el título al desmontar
    return () => {
      document.title = 'Mi App';
    };
  }, [titulo]);
```

```
  return <h1>{titulo}</h1>;
}
```

La función de limpieza se llama:

- Antes de ejecutar el efecto nuevamente (cuando cambian las dependencias)
- Cuando el componente se desmonta del DOM

Suscripción a eventos

```
function VentanaAncho() {
  const [ancho, setAncho] = useState(window.innerWidth);

  useEffect(() => {
    function handleResize() {
      setAncho(window.innerWidth);
    }

    window.addEventListener('resize', handleResize);

    // Limpieza: remover el listener al desmontar
    return () => {
      window.removeEventListener('resize', handleResize);
    };
  }, []); // solo montar y desmontar

  return <p>Ancho: {ancho}px</p>;
}
```

Dependencias faltantes

```
function Busqueda({ query }) {
  const [resultados, setResultados] = useState([]);

  // MAL — query se usa pero no está en el array
```

```
  useEffect(() => {
    buscar(query).then(setResultados);
  }, []);

  // BIEN — query en las dependencias
  useEffect(() => {
    buscar(query).then(setResultados);
  }, [query]);
}
```

Regla: todo valor del componente que se use dentro del efecto debe estar en el array de dependencias

useRef

Un ref es una caja mutable cuyo contenido no provoca re-renders al cambiar

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

function Cronometro() {
  const intervaloRef = useRef(null);

  function iniciar() {
    intervaloRef.current = setInterval(() => {
      console.log('tick');
    }, 1000);
  }

  function detener() {
    clearInterval(intervaloRef.current);
  }

  return (
    <>
      <button onClick={iniciar}>Iniciar</button>
      <button onClick={detener}>Detener</button>
```

```
    </>
  );
}
```

- `useRef(valorInicial)` devuelve un objeto `{ current: valorInicial }`
- Modificar `.current` no re-renderiza el componente

useRef vs useState

| | useState | useRef | |---|---|---| | ¿Re-renderiza al cambiar? | Sí | No | | ¿Para mostrar en la UI? | Sí | No | | Caso de uso | Datos que la UI debe reflejar | Valores internos, timers, IDs |

Acceder al DOM

El uso más común — obtener una referencia directa a un elemento del DOM

```
function CampoAutoFocus() {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus(); // acceso directo al DOM
  }, []);

  return <input ref={inputRef} placeholder="Escribe aquí..." />;
}
```

React asigna el nodo del DOM a `ref.current` después del primer render

Valor anterior

```
function Componente({ valor }) {
  const valorAnterior = useRef(valor);
```

```
  useEffect(() => {
    valorAnterior.current = valor;
  });

  return (
    <p>
      Actual: {valor}, Anterior: {valorAnterior.current}
    </p>
  );
}
```

useContext

Prop drilling

```
App (tiene: usuario)
└── Layout
    └── Sidebar
        └── MenuUsuario  ← necesita: usuario
```

Para llegar de App a MenuUsuario hay que pasar usuario por cada nivel intermedio — aunque Layout y Sidebar no lo usen.

Problemático cuando el árbol es profundo.

useContext: leer el contexto

Context permite "transmitir" datos a cualquier nivel del árbol sin props intermedias

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

// 1. Crear el contexto
const TemaContext = createContext('claro');
```

```
// 2. Proveer el valor en un ancestro
function App() {
  return (
    <TemaContext.Provider value="oscuro">
      <Pagina />
    </TemaContext.Provider>
  );
}

// 3. Consumir en cualquier descendiente
function Boton() {
  const tema = useContext(TemaContext);
  return <button className={`btn-${tema}`}>Clic</button>;
}
```

Usuario autenticado

```
const UsuarioContext = createContext(null);

function App() {
  const [usuario, setUsuario] = useState(null);

  return (
    <UsuarioContext.Provider value={usuario}>
      <NavBar />
      <Contenido />
    </UsuarioContext.Provider>
  );
}

function NavBar() {
  const usuario = useContext(UsuarioContext);
  return <p>{usuario ? `Hola, ${usuario.nombre}` : 'Invitado'}</p>;
}
```

useReducer

Cuando useState se vuelve complejo

```
// Estado con muchas piezas relacionadas
function Formulario() {
  const [nombre, setNombre] = useState('');
  const [email, setEmail] = useState('');
  const [cargando, setCargando] = useState(false);
  const [error, setError] = useState(null);
  const [enviado, setEnviado] = useState(false);

  // lógica dispersa en múltiples setters...
}
```

Cuando el estado tiene múltiples sub-valores que cambian juntos → usar useReducer

Sintaxis

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

function reducer(estado, accion) {
  switch (accion.type) {
    case 'INCREMENTAR':
      return { ...estado, conteo: estado.conteo + 1 };
    case 'DECREMENTAR':
      return { ...estado, conteo: estado.conteo - 1 };
    case 'RESET':
      return { conteo: 0 };
    default:
      return estado;
  }
}
```

```
function Contador() {
  const [estado, dispatch] = useReducer(reducer, { conteo: 0 });

  return (
    <>
      <p>{estado.conteo}</p>
      <button onClick={() => dispatch({ type: 'INCREMENTAR' })}>+</button>
      <button onClick={() => dispatch({ type: 'DECREMENTAR' })}>-</button>
      <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
    </>
  );
}
```

Formulario de registro

```
const initialState = { nombre: '', email: '', cargando: false, error: null };

function reducer(state, action) {
  switch (action.type) {
    case 'CAMPO':
      return { ...state, [action.field]: action.value };
    case 'ENVIAR':
      return { ...state, cargando: true, error: null };
    case 'EXITO':
      return { ...state, cargando: false };
    case 'ERROR':
      return { ...state, cargando: false, error: action.mensaje };
    default:
      return state;
  }
}

function Formulario() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <input
```

```
      value={state.nombre}
      onChange={e => dispatch({ type: 'CAMPO', field: 'nombre', value: e.target.value })}
    />
  );
}
```

useMemo y useCallback

En cada render, React re-ejecuta toda la función del componente

```
function ListaProductos({ productos, filtro }) {
  // Se recalcula en cada render aunque productos y filtro no cambien
  const productosFiltrados = productos.filter(p =>
    p.nombre.toLowerCase().includes(filtro.toLowerCase())
  );

  return productosFiltrados.map(p => <Producto key={p.id} {...p} />);
}
```

Si el cálculo es costoso y las dependencias no cambiaron, es un trabajo innecesario.

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

function ListaProductos({ productos, filtro }) {
  const productosFiltrados = useMemo(() => {
    return productos.filter(p =>
      p.nombre.toLowerCase().includes(filtro.toLowerCase())
    );
  }, [productos, filtro]); // solo recalcula si cambian estos valores

  return productosFiltrados.map(p => <Producto key={p.id} {...p} />);
}
```

- `useMemo(fn, deps)` ejecuta `fn` y guarda el resultado

- Solo lo recalcula cuando alguna dependencia cambia
- Útil para cálculos costosos sobre listas grandes o datos complejos

useCallback

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

function Padre() {
  const [conteo, setConteo] = useState(0);

  // Sin useCallback: nueva referencia en cada render → Hijo re-renderiza
  // Con useCallback: misma referencia si las deps no cambian
  const handleClick = useCallback(() => {
    console.log('clic desde hijo');
  }, []); // sin dependencias → misma función siempre

  return (
    <>
      <p>{conteo}</p>
      <button onClick={() => setConteo(c => c + 1)}>+</button>
      <HijoOptimizado onClick={handleClick} />
    </>
  );
}
```

useMemo vs useCallback

| Hook | Memoriza | Caso de uso | |---|---|---| | `useMemo` | El resultado de una función | Cálculos costosos, filtros, ordenamientos | | `useCallback` | La función en sí | Callbacks pasados a componentes hijos |

```
// useMemo — memoriza el valor
const totalVentas = useMemo(() =>
  ventas.reduce((acc, v) => acc + v.monto, 0),
  [ventas]
```

```
);

// useCallback — memoriza la función
const handleSubmit = useCallback((e) => {
  e.preventDefault();
  enviarFormulario(datos);
}, [datos]);
```

No memoizar

La memoización tiene un costo.

- Cálculos simples: `useMemo(() => a + b, [a, b])` es peor que solo `a + b`
- Todo: añade complejidad sin beneficio

Custom Hooks

Un Custom Hook es una función JavaScript que empieza con `use` y llama a otros Hooks

```
// Hook personalizado
function useDocumentTitle(titulo) {
  useEffect(() => {
    document.title = titulo;
    return () => { document.title = 'Mi App'; };
  }, [titulo]);
}

// Uso en cualquier componente
function Perfil({ nombre }) {
  useDocumentTitle(`Perfil de ${nombre}`);
  return <h1>{nombre}</h1>;
}

function Configuracion() {
```

```
  useDocumentTitle('Configuración');
  return <h1>Ajustes</h1>;
}
```

useFetch

```
function useFetch(url) {
  const [data, setData] = useState(null);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    setCargando(true);
    fetch(url)
      .then(res => res.json())
      .then(setData)
      .catch(setError)
      .finally(() => setCargando(false));
  }, [url]);

  return { data, cargando, error };
}

// Uso
function Usuarios() {
  const { data, cargando, error } = useFetch('/api/usuarios');

  if (cargando) return <p>Cargando...</p>;
  if (error) return <p>Error: {error.message}</p>;
  return <ul>{data.map(u => <li key={u.id}>{u.nombre}</li>)}</ul>;
}
```

Ventajas

- Reutilización
- Separación de responsabilidades

- Testabilidad
- Legibilidad