React Reducers

Sistemas y Tecnologías Web (CC3062) - 2026

React Reducers

Semestre 01, 2026

El problema con useState

Estado simple

`useState` funciona bien para valores independientes

```
// Ideal para useState
const [nombre, setNombre] = useState('');
const [abierto, setAbierto] = useState(false);
const [pagina, setPagina] = useState(1);
```

Estado complejo

```
// Problemas
const [items, setItems] = useState([]);
const [total, setTotal] = useState(0);
const [cargando, setCargando] = useState(false);
const [error, setError] = useState(null);
const [descuento, setDescuento] = useState(0);
```

Lógica dispersa

```
function Carrito() {
  const [items, setItems] = useState([]);
  const [total, setTotal] = useState(0);

  function agregarItem(producto) {
    const nuevos = [...items, producto];
    setItems(nuevos);
    setTotal(nuevos.reduce((acc, i) => acc + i.precio, 0)); // duplicado
  }

  function eliminarItem(id) {
    const nuevos = items.filter(i => i.id !== id);
    setItems(nuevos);
    setTotal(nuevos.reduce((acc, i) => acc + i.precio, 0)); // duplicado
  }
}
```

- La lógica de negocio está mezclada con la UI
- Es fácil olvidar actualizar alguna pieza del estado
- El cálculo del total se repite en cada función

useReducer

El patrón Reducer

Inspirado en la programación funcional

```
estadoActual + acción → nuevoEstado
```

- El estado describe cómo está la aplicación ahora
- Una acción describe qué ocurrió (`{ type, payload }`)
- El reducer es una función pura que combina ambos y devuelve el nuevo estado

La lógica de actualización vive fuera del componente (centralizada y testeable)

Sintaxis

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

const [estado, dispatch] = useReducer(
    reducer,
    estadoInicial
);
```

| Elemento | Descripción | |---|---| | `estado` | El valor actual del estado | | `dispatch` | Función para enviar acciones al reducer | | `reducer` | Función `(estado, accion) => nuevoEstado` | | `estadoInicial` | Valor inicial del estado |

Acción

Una acción es un objeto con al menos `type`:

```
// Acción simple — sin datos adicionales
dispatch({ type: 'INCREMENTAR' });

// Acción con payload — datos para aplicar el cambio
dispatch({ type: 'AGREGAR_ITEM', payload: { id: 1, nombre: 'Laptop', precio: 5000 } });

// Acción con campo explícito
dispatch({ type: 'CAMBIAR_NOMBRE', nombre: 'Carlos' });
```

- `type` — identifica qué ocurrió (MAYÚSCULAS)
- `payload` — datos adicionales necesarios para la actualización

```
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:
      // Siempre retornar el estado actual
      // si no se reconoce la acción
      return estado;
  }
}
```

Reglas del reducer:

1. Función pura: misma entrada, misma salida
2. No mutar el estado: siempre devolver un nuevo objeto
3. Manejar el default: retornar el estado si el tipo no se reconoce

Contador

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

const estadoInicial = { conteo: 0 };

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

```
      return estado;
  }
}

function Contador() {
  const [estado, dispatch] = useReducer(reducer, estadoInicial);

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

Patrones

Action Creators

En lugar de escribir el objeto de acción cada vez, se encapsula en una función.

```
// Sin action creators: repetitivo y posible a errores de tipeo
dispatch({ type: 'AGREGAR_ITEM', payload: producto });
dispatch({ type: 'AGREGAR_ITEM', payload: otroProducto });

// Con action creators: reutilizable
const agregarItem = (producto) => ({ type: 'AGREGAR_ITEM', payload: producto });
const eliminarItem = (id) => ({ type: 'ELIMINAR_ITEM', payload: id });
const limpiarCarrito = () => ({ type: 'LIMPIAR' });

// Uso
dispatch(agregarItem(producto));
dispatch(eliminarItem(3));
dispatch(limpiarCarrito());
```

Estado inicial complejo

Cuando el estado inicial necesita calcularse, se usa una función de inicialización:

```
function calcularEstadoInicial(productosPorDefecto) {
  return {
    items: productosPorDefecto ?? [],
    total: 0,
    descuento: 0,
    cargando: false,
    error: null,
  };
}

// Tercer argumento: función de inicialización (lazy init)
const [estado, dispatch] = useReducer(
  reducer,
  productosPorDefecto,     // argumento inicial
  calcularEstadoInicial    // función que recibe ese argumento
);
```

La función se ejecuta una sola vez

Inmutabilidad con arrays

El reducer nunca muta el estado

```
function reducer(estado, accion) {
  switch (accion.type) {
    case 'AGREGAR':
      return {
        ...estado,
        items: [...estado.items, accion.payload], // nuevo array
      };

    case 'ELIMINAR':
```

```
      return {
        ...estado,
        items: estado.items.filter(i => i.id !== accion.payload), // nuevo array
      };

    case 'ACTUALIZAR':
      return {
        ...estado,
        items: estado.items.map(i =>
          i.id === accion.payload.id ? { ...i, ...accion.payload } : i
        ), // nuevo array con item modificado
      };

    default:
      return estado;
  }
}
```

useReducer + Context

useReducer maneja el cómo cambia el estado

Context maneja el dónde llega el estado

Combinarlos = mundo donde queremos vivir