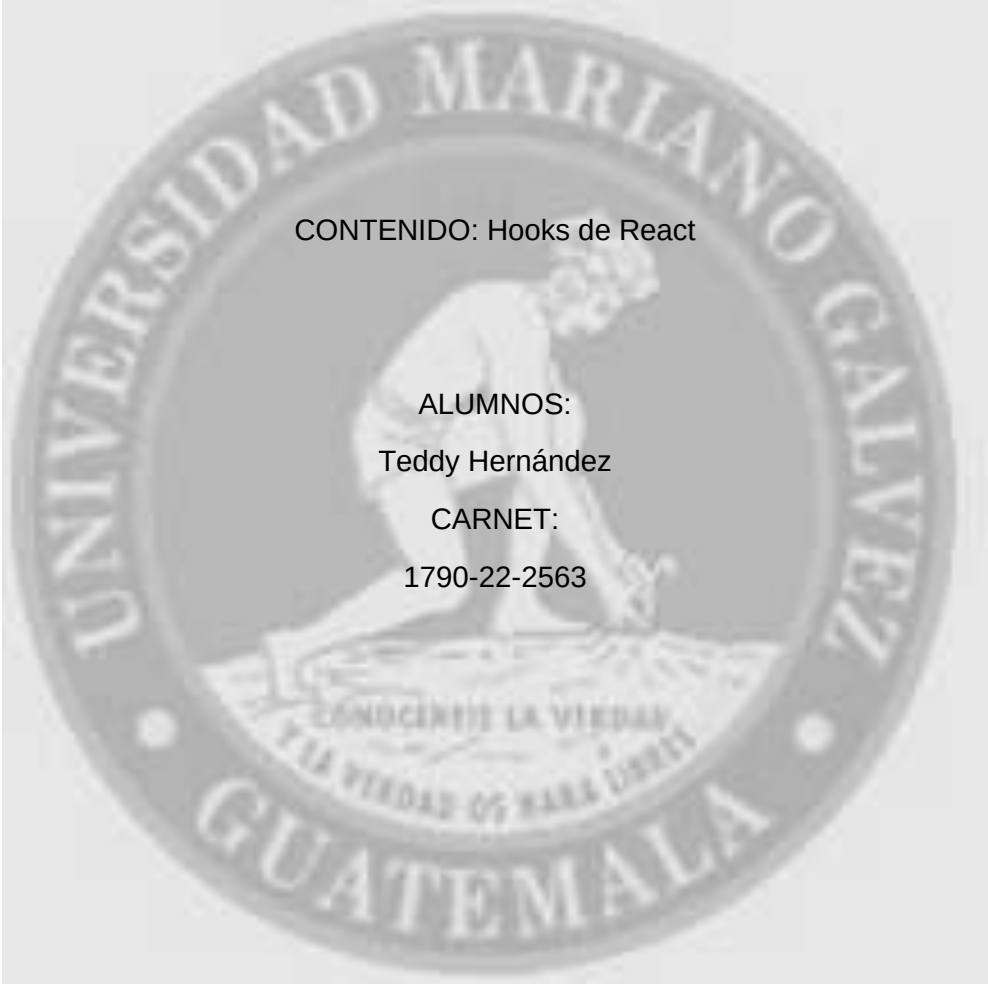
Universidad Mariano Gálvez de Guatemala

Facultad de Ingeniería en Sistemas

CURSO: Desarrollo WEB

CATEDRATICO: Ing. Carmelo Estuardo Mayen Monterroso

SEMESTRE: Octavo



CONTENIDO: Hooks de React

ALUMNOS:

Teddy Hernández

CARNET:

1790-22-2563

Chuiquimulia, 05 de septiembre 2025

Introducción

Los **Hooks de React** son funciones que habilitan estado, efectos y otras capacidades en **componentes funcionales**, permitiendo escribir UIs más declarativas y reutilizables. En esta investigación se estudian los hooks esenciales —useState, useEffect, useMemo, useRef, useContext, useReducer y useCallback— bajo un esquema uniforme (**firma, definición, cuándo usar, ejemplo y errores comunes**). El objetivo es que puedas seleccionar el hook adecuado según el problema (estado simple, sincronización con “sistemas externos”, memorización, contexto o lógica compleja), aplicarlo correctamente siguiendo las **Reglas de los Hooks**, y evitar anti-patrones que afectan la corrección y el rendimiento.

Qué son los Hooks y para qué sirven

Los Hooks son funciones que te permiten usar estado, efectos y otras capacidades de React en componentes funcionales (sin clases). React ofrece hooks integrados y también puedes crear custom hooks para reutilizar lógica entre componentes. react.dev

Reglas de los Hooks (imprescindible)

Llamarlos solo en el tope del componente (no dentro de if, loops, callbacks ni después de un return).

Solo desde componentes de función o desde custom hooks (no en clases ni funciones normales/event handlers).
Estas reglas garantizan que React mantenga el orden de llamadas y el estado correcto. react.dev

Hooks fundamentales (qué resuelve cada uno)

- useState: estado local simple.
- useEffect: sincroniza el componente con sistemas externos (fetch, subscripciones, DOM, timers). Evita usarlo para cálculos que puedes hacer durante el render. react.dev+1
- useRef: referencia mutable (p. ej., a un nodo DOM) que no dispara re-render.
- useMemo: *memorización* de cálculos costosos para no recomputarlos en cada render.
- useCallback: *memorización* de funciones para no crear nuevas instancias innecesarias en cada render.
- useContext: consume valores de un contexto sin prop drilling.
- useReducer: alternativa a useState cuando la lógica del estado es más compleja o compuesta.
- useTransition: marca actualizaciones como “no urgentes” para mantener la UI fluida. (Útil en UIs intensas.)

Novedades relevantes en React 19

React 19 introduce y estandariza acciones de formularios y nuevos hooks pensados para flujos asíncronos y UIs optimistas:

useActionState: maneja estado asociado a una *acción* (p. ej., envío de un formulario) y simplifica casos comunes.

useFormStatus: consulta el estado de envío de un <form>.

useOptimistic: muestra una UI optimista mientras corre una acción asíncrona (actualiza la UI "como si ya hubiera salido bien" y se reconcilia luego).

use: permite "esperar" (await) *promises* directamente en componentes (integración con datos asíncronos).
Estas capacidades se integran con <form> *Actions* de react-dom. react.dev+1

Buenas prácticas (y anti-patrones típicos)

Piensa "¿realmente necesito un effect?"
Si puedes calcular algo durante el render, no uses useEffect. Usa useMemo para cachear cómputos pesados; usa claves (key) para resetear subárboles; mueve lógica a eventos cuando dependa de interacciones del usuario. react.dev

Dependencias correctas en useEffect
Declara todo lo que lees dentro del effect en el arreglo de dependencias (o reestructura para no depender de variables cambiantes). react.dev

Evita llamar Hooks en condiciones/loops
Que siempre se llamen en el mismo orden en cada render. react.dev

Separa efectos por responsabilidad
Varios useEffect pequeños y claros > uno gigante que hace de todo. (Refuerzo de la guía oficial.) react.dev

Prefiere useReducer cuando el estado tenga múltiples campos con transiciones claras (mejora testeo y mantenimiento). react.dev

Hooks

1.UseState

Firma

```
function useState<S>(
  initialState: S | (() => S)
): [S, React.Dispatch<React.SetStateAction<S>>]
```

Definición
Permite **estado local** en un componente funcional. Devuelve el valor actual y una función para actualizarlo. Si pasas una función como inicial, se evalúa **una sola vez** (lazy init).

Cuándo usar

- Formularios controlados, toggles, contadores.
- Estado simple que vive y muere con el componente.
- Cuando **no** necesitas una máquina de estados más compleja.

Ejemplo

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

export function Contador() {
  const [count, setCount] = useState(0); // estado local

  return (
    <button onClick={() => setCount(c => c + 1)}>
      Clicks: {count}
    </button>
  );
}
```

2. UseEffect

Firma

```
function useEffect(
  effect: () => (void | (() => void)),
```

```
  deps?: React.DependencyList
): void
```

Definición
Sincroniza el componente con **sistemas externos**: peticiones de red, suscripciones, timers, manipulación del DOM, localStorage, etc. Puede devolver una **función de limpieza** (cleanup).

Cuando usar

- Fetch de datos (con cancelación/cleanup).
- Suscribirse a eventos (y desuscribirse en cleanup).
- Sincronizar con localStorage o document.title.

Ejemplo (fetch con cancelación)

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

type User = { id: number; name: string };

export function ListaUsuarios({ q }: { q: string }) {
  const [users, setUsers] = useState<User[]>([]);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    const ctrl = new AbortController();

    (async () => {
      try {
        const r = await fetch(`/api/users?q=${encodeURIComponent(q)}`, { signal:
ctrl.signal });
        if (!r.ok) throw new Error("Error al cargar");
        const data: User[] = await r.json();
        setUsers(data);
      } catch (e: any) {
        if (e.name !== "AbortError") setError(e.message);
      }
    })();

    return () => ctrl.abort(); // cleanup al cambiar q o desmontar
  }, [q]);

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

3. UseMemo

Firma

function useMemo<T>(factory: () => T, deps: React.DependencyList): T

Definición

Memoriza el **resultado** de un cálculo costoso o un valor derivado para evitar recomputarlo en cada render, siempre que las **dependencias** no cambien.

Cuándo usar

- Ordenar/filtrar grandes listas.
- Cálculos pesados (parse, agregaciones).
- Crear objetos estables (p. ej., props complejas).

Ejemplo (filtro costoso)

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

export function Buscador({ items }: { items: string[] }) {
  const [q, setQ] = useState("");

  const results = useMemo(() => {
    const term = q.toLowerCase();
    // simula cómputo “pesado”
    return items.filter(it => it.toLowerCase().includes(term));
  }, [q, items]);

  return (
    <>
      <input value={q} onChange={e => setQ(e.target.value)} placeholder="Buscar..." />
      <ul>{results.map(r => <li key={r}>{r}</li>)}</ul>
    </>
  );
}
```

4. UseRef

Firma

function useRef<T>(initialValue: T): React.MutableRefObject<T>

Definición
Guarda un **valor mutable** que **no** dispara re-render al cambiar. Usado para referenciar nodos DOM o para **recordar** valores entre renders (p. ej., ID de un timer).

Cuándo usar

- Acceder a un elemento del DOM (input.focus()).
- Guardar valores que no deban re-renderizar la UI (timers, contadores internos).
- Evitar recrear objetos/instancias entre renders.

Ejemplo (enfocar input)

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

export function Foco() {
  const inputRef = useRef<HTMLInputElement>(null);
  return (
    <>
      <input ref={inputRef} />
      <button onClick={() => inputRef.current?.focus()}>
        Dar foco
      </button>
    </>
  );
}
```

5.UseContext

Firma

function useContext<T>(ctx: React.Context<T>): T

Definición
Consume un **Contexto** para leer valores proporcionados por un <Context.Provider>, evitando el **prop drilling**.

Cuándo usar

- Temas (dark/light), idioma, usuario autenticado.
- Parámetros compartidos por muchos componentes.
- Evitar pasar props en cadenas largas.

Ejemplo (tema de UI)

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

type Theme = "light" | "dark";
const ThemeContext = createContext<Theme>("light");

function Titulo() {
  const theme = useContext(ThemeContext);
  return <h1>{theme === "dark" ? " 🌙" : " 🌞"} Bienvenido</h1>;
}

export function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Titulo />
    </ThemeContext.Provider>
  );
}
```

6. UseReducer

Firma

```
function useReducer<R extends React.Reducer<any, any>, I>(
  reducer: R,
  initialArg: React.ReducerState<R> | I,
  init?: (arg: I) => React.ReducerState<R>
): [React.ReducerState<R>, React.Dispatch<React.ReducerAction<R>>]
```

Definición
Maneja **estado complejo** con una función reducer(state, action) que describe **transiciones**. Devuelve [state, dispatch].

Cuándo usar

- Múltiples campos de estado con reglas claras.
- Secuencias de acciones (agregar/editar/eliminar).
- Prefieres **predecibilidad** y fácil testeo de la lógica.

Ejemplo (to-do básico)

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

type Todo = { id: number; text: string; done: boolean };
type Action =
  | { type: "add"; text: string }
```

```
  | { type: "toggle"; id: number }
  | { type: "remove"; id: number };

function reducer(state: Todo[], action: Action): Todo[] {
  switch (action.type) {
    case "add":
      return [...state, { id: Date.now(), text: action.text, done: false }];
    case "toggle":
      return state.map(t => t.id === action.id ? { ...t, done: !t.done } : t);
    case "remove":
      return state.filter(t => t.id !== action.id);
    default:
      return state;
  }
}

export function Todos() {
  const [todos, dispatch] = useReducer(reducer, []);

  return (
    <>
      <button onClick={() => dispatch({ type: "add", text: "Estudiar hooks" })}>
        Añadir
      </button>
      <ul>
        {todos.map(t => (
          <li key={t.id}>
            <label>
              <input
                type="checkbox"
                checked={t.done}
                onChange={() => dispatch({ type: "toggle", id: t.id })}
              />
              {t.text}
            </label>
            <button onClick={() => dispatch({ type: "remove", id: t.id })}>X</button>
          </li>
        ))}
      </ul>
    </>
  );
}
```

7. useCallback

Firma

```
function useCallback<T extends (...args: any[]) => any>(
  callback: T,
  deps: React.DependencyList
): T
```

Definición
Devuelve una **función memorizada** que solo cambia cuando cambian sus **dependencias**. Útil para **estabilidad de props** hacia hijos memorizados (React.memo) o para evitar recrear funciones en cada render cuando **sí** mediste que impacta.

Cuándo usar

- Pasar handlers a componentes **memorizados** (React.memo).
- Evitar recrear funciones que disparan efectos/cálculos en hijos.
- Cuando realmente **mediste** re-renders innecesarios.

Ejemplo (handler estable para hijo memorizado)

```
import { memo, useCallback, useState } from "react";

const Boton = memo(function Boton({ onAdd }: { onAdd: () => void }) {
  console.log("Render Boton");
  return <button onClick={onAdd}>Agregar</button>;
});

export function Carrito() {
  const [items, setItems] = useState(0);

  const handleAdd = useCallback(() => {
    setItems(n => n + 1);
  }, []); // estable

  return (
    <>
      <p>Items: {items}</p>
      <Boton onAdd={handleAdd} />
    </>
  );
}
```

ERRORES COMUNES PARA CADA HOOKS

useState

- **Mutar objetos/arrays en lugar de crear copias.**
  Síntoma: la UI no cambia.
  Tip: setUser(u => ({ ...u, nombre: "Karla" })), setLista(l => [...l, x]).
- **No usar la actualización funcional cuando depende del estado previo.**
  Síntoma: contadores “pierden” incrementos.
  Tip: setCount(c => c + 1).
- **Derivar estado de props sin necesidad (duplicación de fuente de verdad).**
  Síntoma: datos desincronizados.
  Tip: calcula directamente del prop o usa useMemo; solo crea estado si el usuario lo **edita**.
- **Inicialización costosa sin “lazy init”.**
  Síntoma: renders lentos.
  Tip: useState(() => calcularInicial()).
- **Llamar setX durante el render (no en evento/efecto).**
  Síntoma: bucle infinito.
  Tip: mueve a onClick/useEffect con condiciones.
- **Cambiar un input de no controlado a controlado (o viceversa).**
  Síntoma: warnings y comportamiento raro.
  Tip: usa value **o** defaultValue, no ambos.

useEffect

- **Dependencias incompletas o incorrectas.**
  Síntoma: datos “viejos” o bucles.
  Tip: incluye todo lo que uses dentro del effect; si causa bucle, reestructura la lógica.
- **Usar efectos para lógica de render (que no sincroniza con “algo externo”).**
  Síntoma: complejidad y re-renders extra.
  Tip: calcula en el render; si es caro, useMemo.
- **No limpiar recursos (timers, subscripciones, listeners).**
  Síntoma: pérdidas de memoria, dobles disparos.
  Tip: devuelve un **cleanup** en el effect.
- **Fetch sin cancelación/abort.**
  Síntoma: “setState on unmounted component”.
  Tip: AbortController y return () => ctrl.abort().

- **Un solo effect con demasiadas responsabilidades.**
  Síntoma: difícil de mantener.
  Tip: divide en efectos pequeños por objetivo.
- **Suponer que el effect se ejecuta una sola vez en desarrollo.**
  Síntoma: efectos no idempotentes fallan con Strict Mode (doble invocación).
  Tip: hazlos idempotentes y con cleanup correcto.

useMemo

- **Usarlo "por defecto" sin medir.**
  Síntoma: más complejidad y nulo beneficio.
  Tip: solo para **cómputos costosos** o **identidades estables** que importan.
- **Dependencias incompletas.**
  Síntoma: valores memorados desactualizados.
  Tip: pon **todas** las dependencias.
- **Creer que evita renders.**
  Síntoma: el hijo sigue re-renderizando.
  Tip: useMemo evita **recalcular**, no evita renders; usa React.memo si lo necesitas.
- **Memorizar funciones en useMemo en lugar de useCallback.**
  Tip: funciones → useCallback.
- **Memorizar objetos literales que puedes definir fuera del componente.**
  Tip: si no dependen de props/estado, muévelos fuera.

useRef

- **Esperar que cambiar .current repinte la UI.**
  Síntoma: la vista no se actualiza.
  Tip: usa useState/useReducer para datos que se muestran en la UI.
- **Leer el ref sin comprobar null.**
  Síntoma: errores al montar.
  Tip: ref.current?.focus().
- **Olvidar forwardRef en componentes que deben exponer el ref.**
  Síntoma: el padre no recibe el nodo.
  Tip: const C = forwardRef(...).
- **Guardar en el ref algo que debería ser dependencia de useEffect.**
  Síntoma: efecto no reacciona a cambios.
  Tip: si quieres "escuchar" cambios, úsalo como **estado**, no ref.
- **Usarlo para lógica de negocio persistente** (fuente de verdad).
  Tip: reserva useRef para DOM, IDs de timers y valores **mutables** auxiliares.

UseContex

- **Recrear el value del Provider en cada render.**
  Síntoma: re-render en cadena de todos los consumidores.
  Tip: const value = useMemo(() => ({a,b}), [a,b]) o usa useReducer para {state, dispatch}.
- **Meter demasiada información en un solo contexto.**
  Síntoma: cualquier cambio re-renderiza toda la app.
  Tip: divide en varios contextos (tema, user, permisos…).
- **Leer contexto fuera del Provider.**
  Síntoma: se usa el valor por defecto (o falla).
  Tip: asegura el árbol <Provider>…</Provider>.
- **Mutar el objeto pasado como value.**
  Síntoma: consumidores no se enteran de cambios.
  Tip: crea **nueva referencia** al actualizar.
- **Usar contexto para estado muy local.**
  Tip: prefiere props o levantar estado solo lo necesario.

UseReducer

- **Mutar el estado dentro del reducer.**
  Síntoma: React no detecta cambios.
  Tip: devuelve **nuevos** objetos/arrays.
- **Poner efectos (fetch, timers) en el reducer.**
  Síntoma: reducer “impuro” y bugs.
  Tip: el reducer debe ser **puro**; los efectos van en useEffect.
- **Acciones mal definidas o poco claras.**
  Síntoma: difícil de depurar.
  Tip: usa *discriminated unions* (TS) y tipos claros: { type: "add", payload }.
- **No usar “lazy init” para inicialización costosa.**
  Tip: tercer argumento init o useReducer(reducer, arg, init).
- **Ignorar acciones desconocidas / sin default seguro.**
  Tip: retorna el estado o lanza error en desarrollo.
- **Dependencia oculta de props dentro del reducer.**
  Síntoma: se queda con valores viejos.
  Tip: pasa las props como parte de la **acción** o recalcula en un effect.

UseCallback

- **Dependencias incompletas o usar [] cuando lees props/estado.**
  Síntoma: stale closures (usa valores viejos).
  Tip: añade **todas** las dependencias.
- **Usarlo sin que exista un consumidor que se beneficie (p. ej., hijo sin React.memo).**
  Síntoma: cero mejora, más complejidad.
  Tip: mide antes; acompáñalo con React.memo.
- **Creer que evita renders por sí solo.**
  Tip: solo mantiene **identidad estable** de la función.
- **Provocar bucles al mezclar useCallback y useEffect.**
  Síntoma: effect depende del callback y este depende de estado que cambia.
  Tip: reconsidera la arquitectura; a veces mejor mover la lógica al effect o al evento.
- **Memorizar funciones baratas o que cambian siempre.**
  Tip: si cambia en cada render por necesidad, useCallback no aporta.

Conclusión

Dominar los hooks implica algo más que conocer su API: exige **elegir bien** (estado simple con useState, transiciones complejas con useReducer, sincronización real con useEffect, memorización medida con useMemo/useCallback, referencias con useRef y datos compartidos con useContext), **respetar las reglas** (nivel superior y dependencias completas) y **prevenir errores** (no usar efectos para lógica de render, no mutar estado, no abusar de la memorización). Con estas prácticas, tus componentes serán más predecibles, performantes y fáciles de mantener, y tu código quedará listo para escalar y reutilizarse en proyectos profesionales.