

💡 Guía Definitiva de React.js – De Novato a Experto

💡 **Objetivo:** Esta guía es tu recurso definitivo para dominar React moderno, desde los primeros pasos hasta técnicas avanzadas. Diseñada especialmente para desarrolladores junior que buscan crecer profesionalmente, cada concepto incluye explicaciones paso a paso, ejemplos de código comentados, diagramas visuales y ejercicios prácticos. Dominarás los fundamentos, hooks, patrones de diseño, optimización, seguridad, testing y mucho más.

🎯 Para quién es esta guía:

- **Principiantes absolutos:** Te guiaremos desde cero con explicaciones claras
- **Desarrolladores junior:** Perfecciona tus habilidades y comprende los conceptos avanzados
- **Desarrolladores intermedios:** Profundiza en patrones arquitectónicos y optimización

❖ Características especiales:

- Comentarios línea por línea en todos los ejemplos de código
- Ejercicios prácticos con soluciones guiadas
- Cheatsheets descargables para referencia rápida
- Explicaciones visuales con diagramas
- Mejores prácticas actualizadas a 2025

📋 Índice Detallado

- [💡 Guía Definitiva de React.js – De Novato a Experto](#)
 - [📋 Índice Detallado](#)
 - [⭐ Características Especiales de Esta Guía](#)
 - [📘 1.1 Introducción y novedades](#)
 - [🔍 ¿Qué es React?](#)
 - [NEW Novedades de la versión más reciente \(React 19\)](#)
 - [⚙️ Preparando el entorno](#)
 - [📋 Requisitos previos](#)
 - [🛠️ Crear un nuevo proyecto con Vite](#)
 - [📘 1.2 Conceptos clave para empezar con React](#)
 - [⚙️ Primer Componente](#)
 - [📝 Comentarios paso a paso](#)
 - [📘 Conceptos básicos](#)
 - [✍️ Mini-ejercicio guiado](#)
 - [useEffect](#)
 - [useContext](#)
 - [useReducer](#)
 - [useRef](#)
 - [useMemo](#)
 - [useCallback](#)
 - [useImperativeHandle](#)
 - [useLayoutEffect](#)
 - [useDeferredValue](#)
 - [useTransition](#)

- useDarkMode
- Custom Hook: useLocalStorage
- Mini-ejercicio
- Diagramas de flujo y patrones de diseño
 - ↗ ¿Qué es un diagrama de flujo?
 - Flujo de datos
- Patrones de diseño
 - Mini-ejercicios guiados
 - Resumen
- Conexión con backend y fetch
 - Peticiones básicas con fetch
 - Creando un custom hook para fetch
 - POST, PUT y DELETE con fetch
 - Manejo de autenticación con tokens
 - Mejores prácticas
 - Ejercicio práctico
 - Resumen
- Errores comunes y cómo solucionarlos
- Testing básico y avanzado
 - Unit tests, integración, E2E, snapshots, mocks con MSW y buenas prácticas
- Formularios sencillos y complejos
- Mejorando performance y caching
- Animaciones y UI
- Optimización de carga y lazy loading
- PWA y modo offline
- Internacionalización (i18n)
- SEO y accesibilidad
- Seguridad básica y avanzada
- CI/CD y despliegue
 - Objetivos del capítulo
 - Integración Continua con GitHub Actions
 - Despliegue en Vercel
 - Despliegue en Netlify
 - AWS Amplify
 - Despliegue con Docker
 - Estrategias de despliegue
 - Buenas prácticas
 - Ejercicio práctico
 - Recursos adicionales
- Referencias rápidas, checklist y mentalidad de depuración
- Integración con librerías populares
 - Objetivos de esta sección
 - Librerías de UI Components
 - Tailwind CSS
 - Material UI
 - Chakra UI
 - Enrutamiento con React Router v6
 - Gestión de datos remotos
 - React Query

- SWR (Stale-While-Revalidate)
- Animaciones con Framer Motion
- Recomendaciones para elegir librerías
- Ejercicio práctico
- Ejercicios prácticos y mini proyectos guiados
- Hooks avanzados y patrones
 - Introducción a los hooks avanzados
 - useReducer avanzado
 - useImperativeHandle para referencias avanzadas
 - useDeferredValue para performance
 - useTransition para UI responsiva
 - Patrones avanzados para custom hooks
 - 1. Hook Compuesto
 - 2. Hook con máquina de estados
 - Ejercicio práctico: Crear un hook avanzado
 - Consejos para crear hooks avanzados
 - Recursos adicionales
- ⚡ Avanzado: Optimización, Concurrent Mode y patrones críticos
 - Error Boundaries: Manejo elegante de errores
 - Portales: Renderizado fuera del DOM padre
 - Suspense avanzado: Carga de datos elegante
- 🏗 Arquitectura de aplicaciones grandes y cheatsheets
 - Modularización y organización de código
 - Feature-Based Architecture
 - Nomenclatura y convenciones coherentes
 - Cheatsheets rápidas
 - 📄 Cheatsheet de Hooks
 - 📄 Cheatsheet de Patrones
 - Diagrama de Flujo de Aplicación React
- 🔑 Optimización avanzada, seguridad y performance en producción
 - Lazy loading y code splitting
 - Estrategias de caché efectivas
 - Seguridad en producción
- ⚡ Accesibilidad avanzada (a11y) y buenas prácticas
 - Implementación correcta de roles ARIA
 - Gestión avanzada del foco
 - Herramientas de validación y buenas prácticas
- Monitoreo y logging en producción
- Patrones avanzados de hooks personalizados
- Ejemplos de apps completas
- Plantillas y snippets listos para usar
- Errores comunes de novatos y cómo evitarlos
- Pensamiento Reactivo y mentalidad de desarrollo
- 🎙 Preguntas frecuentes (FAQ)
 - Hooks y estado
 - Peticiones y APIs
 - Formularios y validación
 - Testing y Calidad
 - Arquitectura y patrones

- Despliegue y DevOps
 - Accesibilidad e Internacionalización
 -  Documentación y JSDoc/TSDoc
 - Mejores prácticas de documentación
 -  Conclusión
 - ¿Qué hacer ahora?
-

★ Características Especiales de Esta Guía

- Ejemplos Comentados Línea por Línea** - Cada ejemplo de código incluye explicaciones detalladas
 - Ejercicios Prácticos** - Aprende haciendo con ejercicios diseñados para diferentes niveles
 - Soluciones Guiadas** - Nunca te quedarás atascado, te guiamos paso a paso
 - Patrones Reales** - Usamos patrones que se utilizan en empresas reales
 - Optimizado para Juniors** - Explicaciones claras sin jerga innecesaria
 - Actualizaciones Frecuentes** - Contenido siempre al día con las últimas prácticas
-

■ 1.1 Introducción y novedades

🔍 ¿Qué es React?

React es una biblioteca de JavaScript para construir interfaces de usuario modernas, desarrollada y mantenida por Meta (antes Facebook) y una comunidad activa de desarrolladores.

Conceptos fundamentales:

- **Componentes**: Bloques de construcción reutilizables que encapsulan UI y lógica
- **Virtual DOM**: Sistema eficiente de renderizado que actualiza solo lo necesario
- **Flujo unidireccional de datos**: Facilita el seguimiento y depuración
- **JSX**: Sintaxis declarativa que combina HTML y JavaScript

React se ha convertido en el estándar de la industria para desarrollo frontend debido a su rendimiento, flexibilidad y el enorme ecosistema que lo rodea.

🆕 Novedades de la versión más reciente (React 19)

React evoluciona constantemente para mejorar la experiencia del desarrollador y el rendimiento. Las características más destacadas incluyen:

1. **Concurrent Rendering**: React ahora puede pausar, interrumpir o continuar renderizados sin bloquear la UI. Esto hace que las aplicaciones se sientan más rápidas y fluidas.
2. **Nuevos Hooks**: Se han introducido nuevos hooks como `useTransition` y `useDeferredValue` para manejar tareas pesadas en segundo plano sin bloquear la interfaz.
3. **Strict Mode Mejorado**: Ayuda a detectar problemas en los componentes y a escribir código más seguro.
4. **Suspense para Datos Asíncronos**: Permite mostrar estados de carga más fácilmente cuando se esperan datos.

⚙️ Preparando el entorno

📋 Requisitos previos

Para comenzar con React necesitas:

- **Node.js**: Versión 18.0.0 o superior (recomendado)
- **Editor de código**: VS Code con las siguientes extensiones:
 - ES7+ React/Redux/React-Native snippets
 - ESLint
 - Prettier
 - JavaScript and TypeScript Nightly
- **Conocimientos básicos**: JavaScript moderno (funciones flecha, desestructuración, etc.)

🛠️ Crear un nuevo proyecto con Vite

Vite es una herramienta de compilación moderna que ofrece un entorno de desarrollo extremadamente rápido:

```
# PASO 1: Crea un nuevo proyecto React con Vite
npm create vite@latest mi-primer-app-react -- --template react
```

```
# PASO 2: Navega al directorio del proyecto  
cd mi-primer-app-react  
  
# PASO 3: Instala todas las dependencias  
npm install  
  
# PASO 4: Inicia el servidor de desarrollo  
npm run dev
```

💡 Tip para juniors: Vite es significativamente más rápido que Create React App. El tiempo de inicio del servidor y las actualizaciones en caliente son casi instantáneos, lo que mejora enormemente tu experiencia de desarrollo.

📐 1.2 Conceptos clave para empezar con React

Los siguientes conceptos son fundamentales y te acompañarán durante toda tu carrera con React:

Concepto	Descripción	Por qué es importante
JSX	Sintaxis que combina HTML y JavaScript	Permite escribir UI de forma declarativa e intuitiva
Componentes	Bloques de construcción reutilizables	La base de cualquier app React, mejoran la organización y mantenimiento
Props	Datos pasados de un componente a otro	Permiten comunicación entre componentes y personalización
State	Datos internos gestionados por un componente	Permiten que los componentes sean interactivos y dinámicos
Hooks	Funciones especiales para usar características de React	Facilitan la gestión de estado y efectos secundarios
Context	Sistema para compartir datos sin pasar props	Simplifica la comunicación entre componentes distantes
Efectos	Acciones que ocurren después del renderizado	Permiten sincronizar el componente con sistemas externos

💡 Nota para juniors: Al principio puedes centrarte solo en JSX, Componentes, Props y State. A medida que te sientas más cómodo, ve explorando los demás conceptos.

❖ Primer Componente

Vamos a crear nuestro primer componente React, explicando cada paso:

```
// PASO 1: Importamos React (en versiones modernas esto ya no es obligatorio)
import React from 'react';

// PASO 2: Creamos un componente funcional
// Los componentes en React siempre comienzan con mayúscula
function Bienvenida() {
    // PASO 3: Definimos cualquier variable o lógica que necesitemos
    const nombre = "Developer";
    const estiloTitulo = { color: 'blue', fontSize: '24px' };

    // PASO 4: Retornamos JSX (HTML + JavaScript)
    return (
        <div className="contenedor-bienvenida">
            {/* PASO 4.1: Usamos llaves {} para insertar expresiones JavaScript */}
            <h1 style={estiloTitulo}>¡Hola, {nombre}!</h1>

            {/* PASO 4.2: className en lugar de class (reservada en JS) */}
            <p className="descripcion">Bienvenido a tu primera aplicación React.</p>

            {/* PASO 4.3: Evento onClick con función flecha */}
    
```

```

<button onClick={() => alert('¡React es increíble!')}>
  Haz clic
</button>
</div>
);
}

// PASO 5: Exportamos el componente para poder importarlo en otros archivos
export default Bienvenida;

```

📝 Comentarios paso a paso

Analicemos cada parte importante del componente anterior:

- 1. Componente funcional:** `function Bienvenida()` - Los componentes en React son funciones que retornan JSX o clases que tienen un método render. El enfoque funcional es el recomendado.
- 2. JSX:** La sintaxis similar a HTML pero con superpoderes:
 - Permite insertar JavaScript entre llaves `{nombre}`
 - Usa `className` en lugar de `class` (palabra reservada en JS)
 - Los estilos inline se pasan como objetos JavaScript `style={estiloTitulo}`
 - Eventos en camelCase como `onClick` (en lugar de `onclick`)
- 3. Un solo elemento raíz:** Todo componente debe devolver un solo elemento raíz (en este caso el `<div>`), o puedes usar fragmentos `<>...</>` para agrupar elementos sin añadir nodos extra al DOM.

📘 Conceptos básicos

Los siguientes conceptos son los bloques fundamentales para construir aplicaciones React:

- 1. Componentes:** Piezas reutilizables de UI con su propia lógica y presentación.

```

// Componente simple sin props
function Saludo() {
  return <h1>¡Bienvenido a React!</h1>;
}

// Componente que acepta y utiliza props
function SaludoPersonalizado(props) {
  return <h1>¡Hola, {props.nombre}!</h1>;
}

```

- 2. Props:** Datos que se pasan a los componentes para personalizarlos, como parámetros de una función.

```

// Definición del componente con props
function Saludo({ nombre, edad }) {
  return (
    <div>
      <p>Hola, {nombre}!</p>
      <p>Tienes {edad} años.</p>
    </div>
  );
}

```

```
};

}

// Uso del componente pasando props
function App() {
  return (
    <div>
      <Saludo nombre="Ana" edad={25} />
      <Saludo nombre="Carlos" edad={30} />
    </div>
  );
}
```

3. **Estado (State)**: Información interna que permite a los componentes ser interactivos y responder a eventos.

```
import { useState } from 'react'; // PASO 1: Importamos el hook useState

function Contador() {
  // PASO 2: Declaramos una variable de estado "count" con valor inicial 0
  // useState devuelve un array con dos elementos:
  // - El valor actual del estado (count)
  // - Una función para actualizar ese estado (setCount)
  const [count, setCount] = useState(0);

  // PASO 3: Funciones para manipular el estado
  const incrementar = () => {
    setCount(count + 1); // Actualiza el estado sumando 1
  };

  const decrementar = () => {
    // Usamos el callback para asegurar que operamos con el valor más reciente
    setCount(prevCount => prevCount > 0 ? prevCount - 1 : 0);
  };

  // PASO 4: Renderizamos la UI que muestra y modifica el estado
  return (
    <div className="contador">
      <h2>Contador: {count}</h2>
      <div className="botones">
        <button onClick={decrementar}>-</button>
        <button onClick={incrementar}>+</button>
      </div>
      {/* PASO 5: Mostramos mensajes condicionales basados en el estado */}
      {count > 5 && <p>¡Estás en racha!</p>}
    </div>
  );
}
```

✍ Mini-ejercicio guiado

Vamos a aplicar lo aprendido con un ejercicio paso a paso:

1. Crea un componente llamado **Bienvenida** que reciba props **nombre** y **rol** y muestre un mensaje personalizado:

2. Crea un componente Contador que empiece en **0** y tenga botones para sumar y restar.
3. Integra ambos componentes dentro de App.jsx.

Resumen del Capítulo

- *. React es una librería declarativa y basada en componentes.
- *. La clave son los componentes, props y estado.
- *. React 18 trae mejoras de rendimiento (Concurrent Mode, Suspense, nuevos hooks).
- *. Aprendimos a crear un proyecto con Vite y nuestro primer componente.
- *. Hicimos un mini-ejercicio práctico con props y estado.

```
<div style="page-break-after: always;"></div>
```

2. Hooks básicos y personalizados

Objetivos del capítulo

⌚ Objetivos del capítulo

- *. Comprender qué son los hooks en React.

- *. Usar los hooks nativos más importantes:

useState, useEffect, useContext, useReducer, useRef, useMemo, useCallback.

- *. Crear hooks personalizados (custom hooks).

- *. Practicar con mini-ejercicios guiados.

⚡ ¿Qué son los hooks?

Los hooks son funciones especiales de React que permiten:

- *. Usar estado en componentes de función.
- *. Manejar efectos secundarios (llamadas API, timers, etc.).
- *. Reutilizar lógica en forma de custom hooks.

👉 Antes de hooks, estas funcionalidades solo existían en los componentes de clase.

useState

- ****Objetivo**:** Permitir que un componente funcional tenga estado local.
- ****Explicación**:** `useState` es un hook que permite agregar estado a componentes funcionales. Devuelve un array con dos elementos: el estado actual y una función para actualizarlo.

- ****Tips**:**
- Siempre inicializa el estado con un valor adecuado.
- Utiliza la función de actualización del estado en lugar de modificar el estado directamente.
- Evita usar el estado directamente en el renderizado, en su lugar, utiliza el valor de estado actualizado.

```
```javascript
// Importamos React y el hook useState desde la librería de React
import React, { useState } from 'react';

const Counter = () => {
 // PASO 1: Declaramos una variable de estado llamada "count"
 // useState(0) crea una variable de estado con valor inicial 0
 // Devuelve un array con dos elementos que desestructuramos:
 // - count: el valor actual del estado
 // - setCount: función para actualizar el estado
 const [count, setCount] = useState(0);

 return (
 <div>
 {/* PASO 2: Mostramos el valor actual del estado */}
 <p>Count: {count}</p>

 {/* PASO 3: Al hacer click en el botón, llamamos a setCount para incrementar el
 estado */}
 {/* La función flecha toma el valor actual de count y le suma 1 */}
 {/* React automáticamente vuelve a renderizar el componente con el nuevo valor */}
 <button onClick={() => setCount(count + 1)}>Increment</button>
 </div>
);
};

```

```

- **Explicación detallada:**

- `useState(0)` inicializa el estado `count` en 0. Podríamos usar cualquier valor inicial: números, strings, booleanos, arrays u objetos.
- `count` es el valor actual del estado que React "recuerda" entre renderizaciones.
- `setCount` es la función que actualiza el estado y provoca un nuevo renderizado del componente.
- Cuando `setCount` se llama, React programará una actualización del componente con el nuevo valor.
- React preserva el estado entre renderizados, a diferencia de las variables normales que se reinician.
- Cuando `setCount` se llama, React vuelve a renderizar el componente con el nuevo valor de `count`.

useEffect

- **Objetivo:** Manejar efectos secundarios en componentes funcionales.
- **Explicación:** `useEffect` es un hook que permite realizar efectos secundarios en componentes funcionales, como suscripciones a eventos o llamadas a APIs, timers/intervalos o subscripciones a eventos. Se ejecuta después de que el componente se ha renderizado.
- **Tips:**
 - Utiliza el array de dependencias para controlar cuándo se ejecuta el efecto.
 - Limpia los efectos secundarios en la función de limpieza para evitar fugas de memoria.
 - Evita depender de valores que cambian con frecuencia, ya que esto puede causar re-renderizados innecesarios.

```
// Importamos los hooks que necesitamos
import { useState, useEffect } from 'react';

function Reloj() {
  // PASO 1: Creamos un estado para la hora actual
  // Inicializamos con la hora actual formateada como string
  const [hora, setHora] = useState(new Date().toLocaleTimeString());

  // PASO 2: Configuramos un efecto secundario con useEffect
  useEffect(() => {
    // PASO 3: Creamos un intervalo que se ejecutará cada segundo (1000ms)
    // Este intervalo actualizará nuestro estado con la hora actual
    const intervalo = setInterval(() => {
      // Actualizamos el estado con la nueva hora formateada
      setHora(new Date().toLocaleTimeString());
      // Este setHora causará un re-renderizado del componente con la hora actualizada
    }, 1000);

    // PASO 4: Función de limpieza (cleanup)
    // Esta función se ejecuta cuando:
    // - El componente se desmonta del DOM
    // - Las dependencias del efecto cambian (antes de ejecutar el efecto nuevamente)
    return () => {
      // Limpiamos el intervalo para evitar fugas de memoria
      // Si no limpiáramos, el intervalo seguiría ejecutándose incluso después
      // de que el componente desaparezca de la pantalla
      clearInterval(intervalo);
    };
  }, []);
  // PASO 5: Array de dependencias vacío
  // El [] significa que este efecto solo se ejecuta una vez
  // después del primer renderizado, y la limpieza solo cuando se desmonte

  // PASO 6: Renderizamos la hora actual
  return <h2>{hora}</h2>;
}

}
```

useContext

- **Objetivo:** Compartir estado y funciones entre componentes sin necesidad de props drilling (a evitar).
- **Explicación:** `useContext` es un hook que permite acceder al contexto de React, facilitando la compartición de datos entre componentes. Se utiliza junto con `React.createContext()` para crear un contexto.
- **Tips:**
 - Utiliza `useContext` para evitar pasar props a través de múltiples niveles de componentes.
 - Asegúrate de envolver los componentes que necesitan acceso al contexto con el `Provider` correspondiente.
 - Ten en cuenta que los cambios en el contexto provocarán re-renderizados en todos los componentes que lo consumen.

```
// Importamos las funciones necesarias para crear y usar contextos
import { createContext, useContext } from 'react';

// PASO 1: Crear el contexto
// createContext() crea un objeto de contexto que pueden consumir componentes hijos
// Podemos pasar un valor por defecto (opcional) que se usará si no hay Provider
const TemaContext = createContext();

// PASO 2: Componente proveedor que comparte el contexto
function App() {
  return (
    // El Provider envuelve a todos los componentes que necesitan acceder al contexto
    // La prop "value" contiene el dato que queremos compartir (puede ser cualquier
    tipo)
    <TemaContext.Provider value="dark">
      {/* Todos los componentes dentro del Provider pueden acceder al valor */}
      <Toolbar />
    </TemaContext.Provider>
  );
}

// PASO 3: Componente consumidor que usa el contexto
function Toolbar() {
  // useContext recibe el objeto de contexto y devuelve el valor actual
  // Si el valor del contexto cambia, este componente se volverá a renderizar
  const tema = useContext(TemaContext);

  // Usamos el valor del tema para configurar la clase CSS del botón
  // Este botón se renderizará con el tema que viene del contexto
  return <button className={tema}>Botón con tema {tema}</button>;
}
```

useReducer

- **Objetivo:** Manejar estados complejos y lógicos en componentes funcionales.
- **Explicación:** `useReducer` es un hook que permite manejar estados complejos que requieren múltiples acciones, utilizando un reductor similar a Redux. Devuelve el estado actual y una función para despachar acciones.
- **Tips:**
 - Utiliza `useReducer` cuando el estado es un objeto complejo o cuando se necesita manejar múltiples sub-valores.
 - Define acciones claras y concisas para el reductor.
 - Aprovecha la inmutabilidad al actualizar el estado en el reductor.

```
// Importamos useReducer, alternativa a useState para estados complejos
import { useReducer } from 'react';

// PASO 1: Definimos la función reducer
// Esta función define cómo se comporta nuestro estado cuando ocurren acciones
// Recibe dos argumentos:
// - state: el estado actual
```

```
// - action: la acción que se está ejecutando (generalmente con una prop "type")
function reducer(state, action) {
  // Usamos un switch para manejar diferentes tipos de acciones
  switch (action.type) {
    case 'incrementar':
      // Para cada acción, devolvemos un NUEVO objeto de estado
      // Nunca modificamos el estado directamente (inmutabilidad)
      return { count: state.count + 1 };
    case 'decrementar':
      return { count: state.count - 1 };
    case 'resetear':
      return { count: 0 };
    default:
      // Si la acción no coincide con ningún caso, devolvemos el estado sin cambios
      // Esto es importante como caso de seguridad
      return state;
  }
}

function ContadorAvanzado() {
  // PASO 2: Usamos useReducer proporcionando:
  // - La función reducer que hemos definido
  // - El estado inicial (un objeto con count: 0)
  // Y obtenemos:
  // - state: el estado actual
  // - dispatch: función para enviar acciones al reducer
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      {/* PASO 3: Mostramos el valor actual desde el estado */}
      <p>Contador: {state.count}</p>

      {/* PASO 4: Creamos botones que envían diferentes acciones */}
      {/* Cada botón llama a dispatch con un objeto que tiene la prop "type" */}
      {/* El tipo de acción determina cómo el reducer modificará el estado */}
      <button onClick={() => dispatch({ type: 'incrementar' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrementar' })}>-</button>
      <button onClick={() => dispatch({ type: 'resetear' })}>Reset</button>
    </div>
  );
}
```

useRef

- **Objetivo:** Acceder y manipular elementos del DOM directamente.
- **Explicación:** `useRef` es un hook que permite crear una referencia mutable que persiste durante todo el ciclo de vida del componente. Se utiliza comúnmente para acceder a elementos del DOM o almacenar valores que no provocan re-renderizados.
- **Tips:**
 - Utiliza `useRef` para acceder a elementos del DOM, como inputs o botones.
 - La propiedad `current` de la referencia se puede utilizar para leer o modificar el valor del elemento referenciado.

- `useRef` no provoca re-renderizados cuando su valor cambia.

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

function EnfoqueInput() {
  const inputRef = useRef(null);

  const enfocar = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} placeholder="Escribe algo..." />
      <button onClick={enfocar}>Enfocar input</button>
    </div>
  );
}
```

useMemo

- **Objetivo:** Memorizar valores calculados para evitar cálculos innecesarios en cada renderizado.
- **Explicación:** `useMemo` es un hook que permite memorizar el resultado de una función costosa y solo recalcularlo cuando cambian sus dependencias. Esto puede mejorar el rendimiento al evitar cálculos innecesarios.
- **Tips:**
 - Utiliza `useMemo` para optimizar componentes que realizan cálculos pesados.
 - Asegúrate de proporcionar un array de dependencias adecuado para evitar cálculos innecesarios.
 - Recuerda que `useMemo` solo memoriza el valor, no la función en sí.

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

function Numeros() {
  const [num, setNum] = useState(0);

  const doble = useMemo(() => {
    console.log('Calculando...');
    return num * 2;
  }, [num]);

  return (
    <div>
      <p>Número: {num}</p>
      <p>Doble: {doble}</p>
      <button onClick={() => setNum(num + 1)}>Incrementar</button>
    </div>
  );
}
```

useCallback

- **Objetivo:** Memorizar funciones para evitar recrearlas en cada renderizado.
- **Explicación:** `useCallback` es un hook que devuelve una versión memorizada de la función que solo cambia si alguna de las dependencias ha cambiado. Esto es útil para optimizar el rendimiento de componentes que dependen de funciones que se pasan como props.
- **Tips:**
 - Utiliza `useCallback` para evitar recrear funciones en cada renderizado.
 - Asegúrate de proporcionar un array de dependencias adecuado para evitar comportamientos inesperados.
 - Recuerda que `useCallback` solo memoriza la función, no su resultado.

```
// Importamos los hooks que necesitamos
import { useState, useCallback } from 'react';

// PASO 1: Creamos un componente hijo que recibe una función por props
// Este componente será re-renderizado si sus props cambian
function BotonMemo({ onClick }) {
    // En un caso real, este podría ser un componente costoso de renderizar
    return <button onClick={onClick}>Haz click</button>;
}

function App() {
    // PASO 2: Definimos el estado para nuestro contador
    const [count, setCount] = useState(0);

    // PASO 3: Creamos una función memorizada con useCallback
    // Esta función NO se recreará en cada renderizado porque el array de dependencias
    // está vacío []
    const handleClick = useCallback(() => {
        // Esta función siempre ejecutará el mismo código independientemente del
        // renderizado
        console.log('Click!');
        // [] como dependencias significa que la función nunca se recreará
    }, []);

    // PASO 4: Renderizamos nuestra interfaz
    return (
        <div>
            <p>Clicks: {count}</p>
            {/* PASO 5: Pasamos la función memorizada como prop */}
            {/* Como handleClick está memorizada y no depende de count, BotonMemo nunca se
            re-renderizará */}
            <BotonMemo onClick={handleClick} />

            {/* PASO 6: Este botón usa una función anónima que se recrea en cada renderizado
            */}
            {/* No hay problema aquí porque no es una prop que se pase a componentes hijos
            */}
            <button onClick={() => setCount(count + 1)}>Incrementar contador</button>
        </div>
    );
}
```

useImperativeHandle

- **Objetivo:** Personalizar la instancia del componente expuesta a través de `ref`.
- **Explicación:** `useImperativeHandle` es un hook que permite personalizar el valor de la referencia que se pasa a un componente hijo. Se utiliza junto con `forwardRef` para permitir que los componentes padres accedan a funciones o propiedades específicas del componente hijo.
- **Tips:**
 - Utiliza `useImperativeHandle` para exponer funciones específicas del componente hijo a su padre.
 - Asegúrate de envolver el componente en `ForwardRef` para que pueda recibir la referencia.
 - Ten en cuenta que `useImperativeHandle` se ejecuta cada vez que se renderiza el componente.

```
// Importamos los hooks y utilidades necesarias
import React, { useImperativeHandle, forwardRef, useRef } from 'react';

// PASO 1: Creamos un componente hijo con forwardRef
// forwardRef permite que este componente reciba una ref del componente padre
// props: propiedades normales del componente
// ref: referencia enviada desde el componente padre
const TextInput = forwardRef((props, ref) => {
    // PASO 2: Creamos una referencia interna al elemento input real del DOM
    const inputRef = useRef(null);

    // PASO 3: Personalizamos lo que expone nuestra ref usando useImperativeHandle
    // ref: referencia recibida del parente que estamos personalizando
    // función: devuelve un objeto con los métodos/propiedades que queremos exponer
    useImperativeHandle(ref, () => {
        // Solo exponemos un método llamado 'focus'
        // Esto significa que el parente solo podrá acceder a este método, no a todo el
        // elemento
        focus: () => {
            // Cuando el parente llame a textInputRef.current.focus(),
            // realmente estará ejecutando esta función que enfoca el input interno
            inputRef.current.focus();
        },
        // Podríamos exponer más métodos aquí si fuera necesario
    });
}

// PASO 4: Renderizamos el input real y le asignamos nuestra ref interna
return <input ref={inputRef} type="text" />;
});

// PASO 5: Componente parente que usa el componente hijo con ref
const Parent = () => {
    // PASO 6: Creamos una ref que pasaremos al componente hijo
    const textInputRef = useRef();

    // PASO 7: Función que usará la ref para interactuar con el hijo
    const focusInput = () => {
        // Accedemos al método 'focus' que expusimos con useImperativeHandle
        textInputRef.current.focus();
    };

    // PASO 8: Renderizamos el componente hijo pasándole la ref
    return (

```

```

<div>
  <TextInput ref={textInputRef} />
  <button onClick={focusInput}>Focus Input</button>
</div>
);
};

```

useLayoutEffect

- **Objetivo:** Sincronizar efectos con el layout del DOM.
- **Explicación:** `useLayoutEffect` es un hook similar a `useEffect`, pero se ejecuta de manera sincrónica después de que todas las mutaciones del DOM han sido realizadas. Esto permite leer el layout del DOM y realizar cambios antes de que el navegador pinte la pantalla.
- **Tips:**
 - Utiliza `useLayoutEffect` para medir el tamaño de un elemento o realizar animaciones.
 - Ten en cuenta que `useLayoutEffect` puede afectar el rendimiento si se utiliza incorrectamente, ya que bloquea el pintado del navegador.
 - Asegúrate de limpiar cualquier efecto secundario en la función de limpieza.

```

// Importamos los hooks necesarios
import React, { useLayoutEffect, useRef, useState } from 'react';

const LayoutEffectExample = () => {
  // PASO 1: Creamos una referencia para acceder directamente al elemento DOM
  const divRef = useRef();
  // PASO 2: Estado para demostrar el orden de ejecución
  const [message, setMessage] = useState("Renderizado inicial");

  // PASO 3: Usamos useEffect (comentado) para compararlo con useLayoutEffect
  // useEffect(() => {
  //   // Este código se ejecutaría DESPUÉS de que el navegador pinte la pantalla
  //   console.log("useEffect ejecutado");
  //   const div = divRef.current;
  //   div.style.border = '2px solid red';
  //   setMessage("Actualizado por useEffect");
  // }, []);

  // PASO 4: Usamos useLayoutEffect que se ejecuta ANTES del pintado del navegador
  useLayoutEffect(() => {
    // Este código se ejecuta de manera síncrona ANTES de que el navegador pinte
    console.log("useLayoutEffect ejecutado");

    // PASO 5: Accedemos al elemento DOM directamente
    const div = divRef.current;

    // PASO 6: Realizamos modificaciones del DOM que serán visibles inmediatamente
    // El usuario nunca verá el div sin el borde, porque esta modificación
    // ocurre antes de que el navegador pinte la pantalla
    div.style.border = '2px solid blue';
    div.style.padding = '10px';
    div.style.backgroundColor = '#f0f0ff';

    setMessage("Actualizado por useLayoutEffect");
  });
}

```

```
  }, []); // Array vacío significa que solo se ejecuta una vez después del montaje inicial

  // PASO 7: Renderizamos el componente
  return <div ref={divRef}>{message}</div>;
};
```

useDeferredValue

- **Objetivo:** Retrasar la actualización de un valor hasta que el navegador esté libre.
- **Explicación:** `useDeferredValue` es un hook que permite retrasar la actualización de un valor hasta que el navegador esté listo para procesarlo. Esto es útil para mejorar la experiencia del usuario al evitar bloqueos en la interfaz durante actualizaciones pesadas.
- **Tips:**
 - Utiliza `useDeferredValue` para mejorar la capacidad de respuesta de la interfaz en situaciones de alta carga.
 - Ten en cuenta que el valor diferido puede no reflejar el estado más reciente inmediatamente.
 - Asegúrate de manejar correctamente los estados de carga y error.

```
// Importamos los hooks necesarios
import React, { useDeferredValue, useState, useMemo } from 'react';

const SearchComponent = () => {
  // PASO 1: Definimos el estado para el valor del input
  const [inputValue, setInputValue] = useState('');

  // PASO 2: Creamos un valor diferido basado en inputValue
  // useDeferredValue permite que React retrase este valor cuando la UI necesita ser
  // responsive
  const deferredValue = useDeferredValue(inputValue);

  // PASO 3: Simulamos un componente costoso que depende del valor de búsqueda
  // En un caso real, esto podría ser una búsqueda o filtrado complejo de datos
  const expensiveSearchResults = useMemo(() => {
    // Simulamos procesamiento pesado
    console.log("Calculando resultados para:", deferredValue);

    // En un caso real, aquí filtrarías una lista grande o harías una operación
    // costosa
    return `Mostrando resultados para: ${deferredValue}`;
  }, [deferredValue]);

  // Esta operación costosa se basa en el valor diferido, no en el valor actual
};

// PASO 4: Renderizamos la interfaz
return (
  <div>
    {/* PASO 5: El input siempre muestra el valor actual (alta prioridad) */}
    <input
      type="text"
      value={inputValue}
      onChange={(e) => setInputValue(e.target.value)}
      placeholder="Escribe para buscar..."
```

```

    />

    {/* PASO 6: Indicamos si hay un retraso en la actualización */}
    {inputValue !== deferredValue && (
      <div style={{color: 'gray', fontSize: '0.8em'}}>Actualizando...</div>
    )}

    {/* PASO 7: Mostramos los resultados usando el valor diferido */}
    <p>Search Term: {deferredValue}</p>

    {/* PASO 8: Componente costoso que usa el valor diferido */}
    <div className="search-results">
      {expensiveSearchResults}
    </div>
  </div>
);
};

```

useTransition

- **Objetivo:** Gestionar transiciones de estado sin bloquear la interfaz de usuario.
- **Explicación:** `useTransition` es un hook que permite marcar actualizaciones de estado como "transiciones", lo que permite que React mantenga la interfaz de usuario receptiva durante actualizaciones pesadas. Esto es útil para mejorar la experiencia del usuario al evitar bloqueos en la interfaz.
- **Tips:**
 - Utiliza `useTransition` para envolver actualizaciones de estado que pueden ser pesadas.
 - Ten en cuenta que `useTransition` devuelve un estado `isPending` que puedes usar para mostrar un indicador de carga.
 - Asegúrate de que las transiciones no afecten negativamente la experiencia del usuario.

```

// Importamos los hooks necesarios
import React, { useTransition, useState, useEffect } from 'react';

const TransitionExample = () => {
  // PASO 1: Definimos un estado para el valor del input
  const [inputValue, setInputValue] = useState('');
  // PASO 2: Definimos un estado para los resultados procesados
  const [searchResults, setSearchResults] = useState([]);

  // PASO 3: Inicializamos useTransition
  // - isPending: booleano que indica si hay una transición en progreso
  // - startTransition: función para marcar actualizaciones como transiciones de baja
  // prioridad
  const [isPending, startTransition] = useTransition();

  // PASO 4: Creamos un manejador para el evento onChange del input
  const handleChange = (e) => {
    const newValue = e.target.value;

    // PASO 5: Actualizamos el valor del input inmediatamente (alta prioridad)
    // Esto asegura que el input sea receptivo y muestre lo que el usuario escribe al
    // instante
    setInputValue(newValue);
  };
}

```

```
// PASO 6: Envolvemos la actualización pesada en startTransition
// Esto marca la actualización como menos prioritaria que la UI
startTransition(() => {
  // PASO 7: Simulamos una operación costosa (como buscar en una gran lista de
  datos)
  // En una aplicación real, esto podría ser un filtrado complejo o llamada a API
  const results = simulateExpensiveOperation(newValue);

  // PASO 8: Actualizamos los resultados de búsqueda (baja prioridad)
  setSearchResults(results);
});

// Función que simula una operación costosa
function simulateExpensiveOperation(query) {
  // En un caso real, esto podría ser filtrar miles de elementos o procesar datos
  console.log("Procesando búsqueda para:", query);

  // Simulamos un conjunto de resultados basados en la consulta
  return Array(10).fill(null).map((_, i) => `Resultado ${i+1} para "${query}"`);
}

// PASO 9: Renderizamos la interfaz
return (
  <div>
    {/* PASO 10: El input siempre es receptivo */}
    <input
      type="text"
      value={inputValue}
      onChange={handleChange}
      placeholder="Escribe para buscar..."
    />

    {/* PASO 11: Mostramos un indicador de carga si hay una transición pendiente */}
    {isPending && <p style={{color: 'blue'}}>Actualizando resultados...</p>}

    {/* PASO 12: Mostramos el valor actual del input */}
    <p>Término de búsqueda: {inputValue}</p>

    {/* PASO 13: Mostramos los resultados */}
    <ul>
      {searchResults.map((result, index) => (
        <li key={index}>{result}</li>
      )))
    </ul>
  </div>
);
```

useDarkMode

- **Objetivo:** Gestionar el modo oscuro en la aplicación.
- **Explicación:** `useDarkMode` es un hook personalizado que permite alternar entre el modo claro y oscuro en una aplicación React. Utiliza el estado local y el almacenamiento local para recordar la preferencia del usuario.

- **Tips:**

- Utiliza `useEffect` para sincronizar el estado del modo oscuro con el almacenamiento local.
- Asegúrate de proporcionar una forma de alternar el modo oscuro desde la interfaz de usuario.

```
// Importamos los hooks necesarios
import React, { useEffect, useState } from 'react';

// PASO 1: Creamos nuestro custom hook useDarkMode
const useDarkMode = () => {
    // PASO 2: Definimos un estado local para controlar si estamos en modo oscuro
    const [isDarkMode, setIsDarkMode] = useState(false);

    // PASO 3: Al montar el componente, verificamos si hay una preferencia guardada
    useEffect(() => {
        // PASO 3.1: Intentamos leer la configuración desde localStorage
        const savedMode = localStorage.getItem('darkMode');

        // PASO 3.2: Si existe un valor guardado, lo aplicamos al estado
        if (savedMode !== null) {
            setIsDarkMode(savedMode === 'true');
        } else {
            // PASO 3.3: Como alternativa, podemos detectar la preferencia del sistema
            const prefersDark = window.matchMedia('(prefers-color-scheme: dark)').matches;
            setIsDarkMode(prefersDark);
        }
    }, []);
    // Array vacío significa que solo se ejecuta al montar el componente

    // PASO 4: También podemos sincronizar con cambios en la preferencia del sistema
    useEffect(() => {
        // PASO 4.1: Creamos un detector de cambios en las preferencias del sistema
        const mediaQuery = window.matchMedia('(prefers-color-scheme: dark)');

        // PASO 4.2: Definimos un manejador para cuando cambie la preferencia
        const handleChange = (e) => {
            // Solo actualizamos si no hay configuración manual guardada
            if (localStorage.getItem('darkMode') === null) {
                setIsDarkMode(e.matches);
            }
        };
        // PASO 4.3: Suscribimos el manejador a los cambios
        mediaQuery.addEventListener('change', handleChange);

        // PASO 4.4: Limpiamos el event listener cuando el componente se desmonte
        return () => {
            mediaQuery.removeEventListener('change', handleChange);
        };
    }, []);
    // PASO 5: Creamos una función para alternar el modo oscuro
    const toggleDarkMode = () => {
        // PASO 5.1: Usamos la versión funcional del setter para garantizar el valor
        // correcto
        setIsDarkMode((prevMode) => {
            // PASO 5.2: Calculamos el nuevo modo (invertir el actual)
            return !prevMode;
        });
    };
};

export default useDarkMode;
```

```

const newMode = !prevMode;

// PASO 5.3: Guardamos la preferencia en localStorage para persistencia
localStorage.setItem('darkMode', String(newMode));

// PASO 5.4: También podríamos aplicar clases CSS o atributos al documento
if (newMode) {
  document.documentElement.setAttribute('data-theme', 'dark');
} else {
  document.documentElement.setAttribute('data-theme', 'light');
}

return newMode;
});

};

// PASO 6: Retornamos el estado y la función para cambiarlo
// Seguimos el patrón común de hooks: [valor, setValor]
return [isDarkMode, toggleDarkMode];
};

// PASO 7: Componente que usa nuestro custom hook
const App = () => {
  // PASO 7.1: Usamos el hook como cualquier otro hook de React
  const [isDarkMode, toggleDarkMode] = useDarkMode();

  return (
    <div style={{ background: isDarkMode ? 'black' : 'white', color: isDarkMode ?
'white' : 'black' }}>
      <h1>Hello, World!</h1>
      <button onClick={toggleDarkMode}>Toggle Dark Mode</button>
    </div>
  );
};

```

Custom Hook: useLocalStorage

Un custom hook es una función que usa hooks dentro y encapsula lógica reutilizable.

```

import { useState } from 'react';

function useLocalStorage(key, valorInicial) {
  const [valor, setValor] = useState(() => {
    const itemGuardado = localStorage.getItem(key);
    return itemGuardado ? JSON.parse(itemGuardado) : valorInicial;
  });

  const setValorLS = nuevoValor => {
    setValor(nuevoValor);
    localStorage.setItem(key, JSON.stringify(nuevoValor));
  };

  return [valor, setValorLS];
}

```

```
// Uso del hook
function App() {
  const [nombre, setNombre] = useLocalStorage('nombre', '');

  return (
    <div>
      <input
        value={nombre}
        onChange={e => setNombre(e.target.value)}
        placeholder="Escribe tu nombre"
      />
      <p>Hola, {nombre}</p>
    </div>
  );
}
```

Mini-ejercicio

1. Reloj con `useEffect` que muestra la hora actual cada segundo.
2. Contador avanzado con `useReducer` que permite sumar, restar y resetear.
3. Componente custom : `useToggle` que maneje un valor booleano. Como por ejemplo un botón de Me gusta.

- **Resumen de Hooks**

- `useState`- estado basico o local.
- `useEffect`- efectos secundarios.
- `useContext`- estado global.
- `useReducer`- manejo de estado complejo.
- `useRef`- referencia a elementos del DOM.
- `useMemo`- optimización de rendimiento.
- `useCallback`- memorizar funciones.

Diagramas de flujo y patrones de diseño

- **Objetivos del capítulo**
- Comprender cómo organizar el flujo de datos en React.
- Conocer los patrones de diseño más comunes para estructurar apps.
- Aprender a usar diagramas de flujo para planificar la lógica antes de programar.
- Practicar con ejemplos y mini-ejercicios.

💡 ¿Qué es un diagrama de flujo?

Un diagrama de flujo es una representación visual de cómo fluyen los datos y decisiones en una aplicación. Sirve para:

- Planificar antes de escribir código.
- Entender qué pasa "si ocurre X".
- Comunicar ideas a otros devs o juniors.

Ejemplo de un flujo para manejar un formulario de login:



👉 Este tipo de esquema ayuda a saber dónde colocar la lógica (handleSubmit, validaciones, fetch, etc.).

Flujo de datos

- **Unidireccional:** Los datos fluyen en una sola dirección, desde los componentes padres a los hijos.
- **Props:** Los componentes hijos reciben datos a través de props.
- **State:** El estado se gestiona en componentes padres y se pasa a los hijos según sea necesario.
- **Context:** Para evitar el "prop drilling", se puede utilizar el Context API para compartir datos entre componentes sin tener que pasarlos explícitamente a través de props.

Patrones de diseño

- **Composición de componentes:** Crear componentes reutilizables y componibles que se pueden combinar para formar interfaces más complejas. React fomenta el uso de composición sobre herencia.

```
function Card({ children }) {
  return <div className="card">{children}</div>;
}

function App() {
  return (
    <Card>
      <h2>Hola <img alt="hand icon" style={{ verticalAlign: 'middle' }} /></h2>
      <p>Esto es una tarjeta reutilizable</p>
    </Card>
  );
}
```

☞ La composición permite crear interfaces complejas a partir de piezas simples.

- **Container/Presentational:** Separar lógica y presentación:
- Container → maneja estado, llamadas API, lógica.
- Presentational → solo muestra UI con props.

```
// Presentational
function ListaUsuarios({ usuarios }) {
  return (
    <ul>
      {usuarios.map(u => (
        <li key={u.id}>{u.nombre}</li>
      )))
    </ul>
  );
}

// Container
import { useEffect, useState } from 'react';

function UsuariosContainer() {
  const [usuarios, setUsuarios] = useState([]);

  useEffect(() => {
    fetch('/api/usuarios')
      .then(res => res.json())
      .then(data => setUsuarios(data));
  }, []);

  return <ListaUsuarios usuarios={usuarios} />;
}
```

👉 Esto ayuda a reutilizar la UI en distintos contextos.

- **Render Props:** Permite pasar funciones como hijos para compartir lógica.

```
function FetchData({ url, children }) {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then(r => r.json())
      .then(setData);
  }, [url]);

  return children(data);
}

function App() {
  return (
    <FetchData url="/api/productos">
      {productos =>
        productos ? (
          <ul>
            {productos.map(p => (
              <li key={p.id}>{p.nombre}</li>
            ))}
          </ul>
        ) : (
          <p>Cargando...</p>
        )
      }
    </FetchData>
  );
}
```

👉 Aunque hoy en día muchos prefieren custom hooks, Render Props sigue siendo útil en algunos casos.

- **Higher Order Components (HOCs):** Un HOC es una función que recibe un componente y devuelve otro con funcionalidades extra.
- **Ejemplo de HOC:** proteger rutas de login.

```
function withAuth(Component) {
  return function ProtectedComponent(props) {
    const isLoggedIn = localStorage.getItem('token') !== null;
    if (!isLoggedIn) return <p>No tienes acceso</p>;
    return <Component {...props} />;
  };
}

// Uso
function Dashboard() {
  return <h2>Panel privado</h2>;
}
```

```
export default withAuth(Dashboard);
```

☞ Aunque están en desuso frente a hooks, los HOC siguen apareciendo en librerías.

- **Hooks personalizados(reutilización moderna)**: Hoy en día, el patrón más usado es encapsular lógica en custom hooks.
- **Ejemplo:** manejo de fetch

```
// Importamos los hooks que necesitamos para nuestro custom hook
import { useState, useEffect } from 'react';

// PASO 1: Creamos un custom hook llamado useFetch
// Los custom hooks son funciones que comienzan con "use" y pueden usar otros hooks
function useFetch(url) {
    // PASO 2: Definimos estados para los datos y el estado de carga
    const [data, setData] = useState(null); // Almacena los datos obtenidos de la API
    const [loading, setLoading] = useState(true); // Indica si la petición está en curso

    // PASO 3: Usamos useEffect para ejecutar código cuando el componente se monta
    // o cuando la URL cambia
    useEffect(() => {
        // PASO 4: Realizamos la petición fetch cuando el componente se monta
        fetch(url)
            .then(r => r.json()) // Convertimos la respuesta a JSON
            .then(d => {
                // PASO 5: Actualizamos los estados con los datos obtenidos
                setData(d); // Guardamos los datos
                setLoading(false); // Indicamos que la carga ha terminado
            });
    }, [url]); // La dependencia [url] hace que el efecto se ejecute cuando cambia la URL

    // PASO 6: Devolvemos un objeto con los datos y el estado de carga
    // Este es el "API" de nuestro custom hook
    return { data, loading };
}

// EJEMPLO DE USO DEL CUSTOM HOOK
function Productos() {
    // PASO 7: Usamos nuestro custom hook como cualquier otro hook
    // Renombramos "data" a "productos" con la sintaxis de desestructuración
    const { data: productos, loading } = useFetch('/api/productos');

    // PASO 8: Renderizado condicional basado en el estado de carga
    if (loading) return <p>Cargando...</p>

    // PASO 9: Una vez que tenemos los datos, los renderizamos
    return (
        <ul>
            {/* PASO 10: Iteramos sobre los productos usando map */}
            {productos.map(p => (
```

```
// Cada elemento de la lista necesita una key única
    <li key={p.id}>{p.nombre}</li>
  )}
</ul>
);
}
```

👉 Este enfoque es más limpio y moderno que Render Props o HOC.

Mini-ejercicios guiados

1. Dibuja un diagrama de flujo para un carrito de compras (añadir, eliminar, pagar).
2. Refactoriza un componente con mucha lógica para que siga el patrón Container/Presentational.
3. Crea un custom hook useToggle y compáralo con la solución usando HOC o Render Props.

Resumen

- *. Los diagramas de flujo son clave para planificar antes de programar.
- *. Patrones de diseño en React más usados:
 - *. Composición (recomendado por React).
 - *. Container/Presentational (separa lógica y UI).
 - *. Render Props (compartir lógica a través de children).
 - *. HOCs (extender componentes).
 - *. Custom hooks (la opción más moderna y flexible).

Conexión con backend y fetch

- **Objetivo:** Aprender a conectar una aplicación React con un backend para obtener y enviar datos.
- **Principales conceptos:**
 - Fetch API para peticiones HTTP
 - Estados de carga y error
 - Manejo de respuestas y datos
 - Buenas prácticas de comunicación cliente-servidor
 - Custom hooks para peticiones

Peticiones básicas con fetch

El método más simple para conectar con un backend es usando la API nativa `fetch`:

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

function UserContainer() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('/api/users')
      .then(res => {
        if (!res.ok) throw new Error('Error al cargar usuarios');
        return res.json();
      })
      .then(data => {
        setUsers(data);
        setLoading(false);
      })
      .catch(err => {
        setError(err.message);
        setLoading(false);
      });
  }, []);

  if (loading) return <p>Cargando usuarios...</p>;
  if (error) return <p>Error: {error}</p>;

  return <UserList users={users} />;
}

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

Creando un custom hook para fetch

Para reutilizar la lógica de peticiones, es recomendable crear un custom hook:

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const controller = new AbortController();
    const signal = controller.signal;

    setLoading(true);

    fetch(url, { signal })
      .then(response => {
        if (!response.ok) {
          throw new Error(`Error HTTP: ${response.status}`);
        }
        return response.json();
      })
      .then(data => {
        setData(data);
        setError(null);
        setLoading(false);
      })
      .catch(err => {
        if (err.name === 'AbortError') {
          console.log('Fetch abortado');
        } else {
          setError(err.message);
          setData(null);
          setLoading(false);
        }
      });
  });

  return () => controller.abort();
}, [url]);

return { data, loading, error };
}

// Uso del hook:
function ProductList() {
  const { data, loading, error } = useFetch('/api/products');

  if (loading) return <p>Cargando productos...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <h2>Productos</h2>
      <ul>
        {products?.map(product => (

```

```

        <li key={product.id}>{product.name}</li>
    )}
</ul>
</div>
);
}

```

POST, PUT y DELETE con fetch

Para enviar datos al servidor:

```

function createProduct(product) {
  return fetch('/api/products', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(product),
  }).then(res => res.json());
}

function updateProduct(id, updates) {
  return fetch(`/api/products/${id}`, {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(updates),
  }).then(res => res.json());
}

function deleteProduct(id) {
  return fetch(`/api/products/${id}`, {
    method: 'DELETE',
  }).then(res => res.json());
}

// Uso en un componente:
function ProductForm() {
  const [name, setName] = useState('');
  const [price, setPrice] = useState('');
  const [submitting, setSubmitting] = useState(false);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setSubmitting(true);

    try {
      await createProduct({ name, price: Number(price) });
      setName('');
      setPrice('');
      alert('¡Producto creado!');
    } catch (error) {
      console.error('Error creando producto:', error);
    }
  }
}

```

```
        alert('Error al crear producto');
    } finally {
        setSubmitting(false);
    }
};

return (
    <form onSubmit={handleSubmit}>
        <div>
            <label htmlFor="name">Nombre:</label>
            <input
                id="name"
                value={name}
                onChange={e => setName(e.target.value)}
                required
            />
        </div>
        <div>
            <label htmlFor="price">Precio:</label>
            <input
                id="price"
                type="number"
                value={price}
                onChange={e => setPrice(e.target.value)}
                required
            />
        </div>
        <button type="submit" disabled={submitting}>
            {submitting ? 'Enviando...' : 'Crear Producto'}
        </button>
    </form>
);
}
```

Manejo de autenticación con tokens

Para APIs que requieren autenticación:

```
function fetchWithAuth(url, options = {}) {
    // Obtiene el token de localStorage
    const token = localStorage.getItem('authToken');

    // Prepara los headers
    const headers = {
        ...options.headers,
        'Authorization': `Bearer ${token}`
    };

    // Hace la petición con el token
    return fetch(url, {
        ...options,
        headers
    });
}
```

```
// Uso
function fetchUserData() {
  return fetchWithAuth('/api/user/profile')
    .then(res => {
      if (res.status === 401) {
        // Token inválido o expirado, redirigir a login
        localStorage.removeItem('authToken');
        window.location.href = '/login';
        throw new Error('Sesión expirada');
      }
      return res.json();
    });
}
```

Mejores prácticas

1. **Siempre maneja estados de carga y error** para mejorar UX
2. **Centraliza lógica de API** en servicios o custom hooks
3. **Usa AbortController** para cancelar peticiones pendientes al desmontar componentes
4. **Evita race conditions** (respuestas fuera de orden)
5. **Usa env variables** para URLs de API según entorno (dev/prod)
6. **Implementa retry** para peticiones importantes que fallan
7. **Considera librerías** como Axios o React Query para apps grandes

Ejercicio práctico

Crea un componente <TodoList> que:

1. Cargue tareas desde una API (usa JSONPlaceholder: <https://jsonplaceholder.typicode.com/todos>)
2. Muestre estados de carga y error
3. Permita marcar tareas como completadas
4. Permita eliminar tareas

Este ejercicio te ayudará a practicar lo aprendido sobre conexión con backend en React.

Resumen

- React no tiene método "oficial" para conectar con APIs
- **fetch** es la opción nativa más simple
- Para proyectos grandes, considera SWR o React Query
- Siempre gestiona: carga, errores y cancelación
- Los custom hooks son excelentes para reutilizar lógica de API

Errores comunes y cómo solucionarlos

- **Objetivo:** Identificar y solucionar errores comunes en React relacionados con hooks, props y estado.
- **Errores comunes:**
 - **Hooks:** Llamar a hooks dentro de condicionales o bucles.
 - **Props:** Pasar props incorrectas o no definirlas en el componente hijo.

- **Estado:** Mutar el estado directamente en lugar de usar setState.
- **Efectos:** No limpiar efectos secundarios o dependencias incorrectas en useEffect.
- **Renderizado:** No optimizar el renderizado de componentes, lo que puede llevar a problemas de rendimiento.
- **Accesibilidad:** No tener en cuenta la accesibilidad en los componentes, lo que puede afectar a los usuarios con discapacidades.
- **Pruebas:** No implementar pruebas adecuadas para los componentes, lo que puede llevar a errores no detectados en la aplicación.
- **Rendimiento:** No optimizar el rendimiento de la aplicación, lo que puede llevar a tiempos de carga más largos y una mala experiencia de usuario.
- **Seguridad:** No tener en cuenta las mejores prácticas de seguridad, lo que puede dejar la aplicación vulnerable a ataques.
- **Documentación:** No mantener una documentación adecuada de los componentes y su uso, lo que puede dificultar la colaboración y el mantenimiento del código.
- **Depuración:** No implementar herramientas de depuración adecuadas, lo que puede dificultar la identificación y solución de problemas en la aplicación.

- **Soluciones:**

- **Hooks:** Asegurarse de que los hooks se llamen en el nivel superior del componente y no dentro de condicionales o bucles.
- **Props:** Validar las props en los componentes hijos y proporcionar valores predeterminados si es necesario.
- **Estado:** Utilizar setState para actualizar el estado en lugar de mutarlo directamente.
- **Efectos:** Limpiar los efectos secundarios en useEffect y especificar correctamente las dependencias.
- **Renderizado:** Utilizar técnicas de memoización y optimización para evitar renderizados innecesarios.
- **Accesibilidad:** Seguir las pautas de accesibilidad y realizar pruebas con usuarios que tengan discapacidades.
- **Pruebas:** Implementar pruebas unitarias y de integración para los componentes.
- **Rendimiento:** Utilizar herramientas de análisis de rendimiento y optimizar el código según sea necesario.
- **Seguridad:** Seguir las mejores prácticas de seguridad y realizar auditorías de seguridad regularmente.
- **Documentación:** Mantener una documentación clara y actualizada de los componentes y su uso.
- **Depuración:** Utilizar herramientas de depuración y seguimiento para identificar y solucionar problemas en la aplicación.

- **Ejemplo de error común y solución:**

- **Error:** Llamar a un hook dentro de una condicional.
- **Solución:** Mover la llamada del hook al nivel superior del componente.
- **Ejemplo:** En lugar de esto:

```
if (isLoggedIn) {  
  const user = useUser();  
}
```

Haz esto:

```
const user = isLoggedIn ? useUser() : null;
```

Testing básico y avanzado

Unit tests, integración, E2E, snapshots, mocks con MSW y buenas prácticas

- **Objetivo:** Aprender a escribir pruebas para componentes React utilizando herramientas como Jest y React Testing Library.
- **Pruebas unitarias:** Escribir pruebas para componentes individuales asegurando que funcionen correctamente de forma aislada.
- **Pruebas de integración:** Probar la interacción entre múltiples componentes y su integración con APIs.
- **Pruebas E2E:** Simular el comportamiento del usuario en la aplicación completa para asegurar que todo funcione como se espera.
- **Snapshots:** Capturar el estado de un componente en un momento dado y compararlo en futuras ejecuciones de pruebas.
- **Mocks con MSW:** Utilizar Mock Service Worker para interceptar y simular respuestas de API en pruebas.
- **Buenas prácticas:** Seguir principios como la independencia de pruebas, la claridad y la mantenibilidad.
- **Ejemplo de prueba unitaria:**
 - **Descripción:** Probar un componente de botón que llama a una función al hacer clic.
 - **Código:**

```
import { render, screen, fireEvent } from '@testing-library/react';
import MyButton from './MyButton';

test('llama a la función onClick al hacer clic', () => {
  const handleClick = jest.fn();
  render(<MyButton onClick={handleClick} />);
  fireEvent.click(screen.getByRole('button'));
  expect(handleClick).toHaveBeenCalled();
});
```

Formularios sencillos y complejos

- **Objetivo:** Aprender a manejar formularios en React, desde simples hasta complejos, utilizando librerías populares como React Hook Form y Formik.
- **Formularios simples:** Crear formularios básicos con validaciones simples utilizando React Hook Form.
- **Formularios complejos:** Manejar formularios más complejos con múltiples pasos y validaciones avanzadas utilizando Formik.
- **Manejo de errores:** Implementar un manejo de errores efectivo y mostrar mensajes de error claros para los usuarios.
- **Manejo de estados:** Gestionar el estado de los formularios de manera eficiente, incluyendo estados de carga y éxito.
- **Validaciones:** Implementar validaciones en los formularios para asegurar que los datos ingresados sean correctos y completos.
- **Manejo de efectos secundarios:** Utilizar efectos secundarios para manejar acciones como la validación de formularios y la gestión de estados de carga.
- **Optimización de rendimiento:** Aplicar técnicas de optimización de rendimiento en formularios, como la memoización y la carga diferida.
- **Manejo de accesibilidad:** Asegurar que los formularios sean accesibles para todos los usuarios, incluyendo aquellos con discapacidades.
- **Documentación y ejemplos:** Proporcionar documentación clara y ejemplos prácticos para facilitar la comprensión y el uso de los formularios.
- **Pruebas:** Implementar pruebas para asegurar que los formularios funcionen correctamente y cumplan con los requisitos.
- **Mejora continua:** Fomentar la mejora continua de los formularios a través de la retroalimentación de los usuarios y la revisión del código.
- **Integración con otras librerías:** Asegurar que los formularios se integren bien con otras librerías y herramientas utilizadas en la aplicación.
- **Internacionalización:** Implementar soporte para múltiples idiomas y regiones en los formularios.
- **Ejemplo de formulario simple con React Hook Form:**

```
import React from 'react';
import { useForm } from 'react-hook-form';

const MyForm = () => {
  const { register, handleSubmit, formState: { errors } } = useForm();

  const onSubmit = data => {
    console.log(data);
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <div>
        <label htmlFor="name">Nombre</label>
        <input id="name" {...register('name', { required: true })} />
        {errors.name && <span>Este campo es obligatorio</span>}
      </div>
      <button type="submit">Enviar</button>
    </form>
  );
}
```

```
};

export default MyForm;
```

- **Ejemplo de formulario complejo con Formik:**

```
import React from 'react';
import { Formik, Form, Field, ErrorMessage } from 'formik';

const MyComplexForm = () => {
  return (
    <Formik
      initialValues={{ name: '', email: '' }}
      validate={values => {
        const errors = {};
        if (!values.name) {
          errors.name = 'Requerido';
        }
        if (!values.email) {
          errors.email = 'Requerido';
        } else if (!/\S+@\S+\.\S+/.test(values.email)) {
          errors.email = 'Email inválido';
        }
        return errors;
      }}
      onSubmit={(values, { setSubmitting }) => {
        setTimeout(() => {
          console.log(values);
          setSubmitting(false);
        }, 400);
      }}
    >
    {({ isSubmitting }) => (
      <Form>
        <div>
          <label htmlFor="name">Nombre</label>
          <Field type="text" id="name" name="name" />
          <ErrorMessage name="name" component="span" />
        </div>
        <div>
          <label htmlFor="email">Email</label>
          <Field type="email" id="email" name="email" />
          <ErrorMessage name="email" component="span" />
        </div>
        <button type="submit" disabled={isSubmitting}>
          Enviar
        </button>
      </Form>
    )}
  </Formik>
);
};

export default MyComplexForm;
```

- Ejemplo de formulario sin librerías ni Hooks:

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

const MySimpleForm = () => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [errors, setErrors] = useState({});

  const validate = () => {
    const newErrors = {};
    if (!name) newErrors.name = 'Requerido';
    if (!email) newErrors.email = 'Requerido';
    else if (!/\S+@\S+\.\S+/.test(email)) newErrors.email = 'Email inválido';
    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    if (validate()) {
      console.log({ name, email });
    }
  };
}

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="name">Nombre</label>
      <input
        type="text"
        id="name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      {errors.name && <span>{errors.name}</span>}
    </div>
    <div>
      <label htmlFor="email">Email</label>
      <input
        type="email"
        id="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      {errors.email && <span>{errors.email}</span>}
    </div>
    <button type="submit">Enviar</button>
  </form>
);

export default MySimpleForm;
```

Mejorando performance y caching

- **Objetivo:** Aprender técnicas para mejorar el rendimiento de las aplicaciones React mediante la memoización y el caching de datos.
- **Estrategias:**
 - **Memoización:** Utilizar `React.memo` para componentes funcionales y `useMemo/useCallback` para funciones y valores derivados.
- **Ejemplo:** Un componente que muestra una lista de elementos puede ser memoizado para evitar renders innecesarios.

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

const MyList = ({ items }) => {
  const renderedItems = useMemo(() => {
    return items.map(item => <li key={item.id}>{item.name}</li>);
  }, [items]);

  return <ul>{renderedItems}</ul>;
};
```

- **Caching:** Implementar soluciones de caching como React Query o SWR para manejar datos remotos y optimizar las solicitudes.
- **Ejemplo con React Query:**

```
import React from 'react';
import { useQuery } from 'react-query';

const MyComponent = () => {
  const { data, error, isLoading } = useQuery('myData', fetchMyData);

  if (isLoading) return <div>Cargando...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return (
    <div>
      {data.map(item => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  );
};
```

- **Ejemplo sin React Query:**

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

```
const MyComponent = () => {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('/api/myData');
        if (!response.ok) throw new Error('Network response was not ok');
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };
    fetchData();
  }, []);

  if (isLoading) return <div>Cargando...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return (
    <div>
      {data.map(item => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  );
};
```

Animaciones y UI

- **Framer Motion:** Una biblioteca popular para animaciones en React que permite crear animaciones complejas de manera sencilla.
- **Ejemplo básico con Framer Motion:**

```
import { motion } from 'framer-motion';

const MyComponent = () => {
  return (
    <motion.div
      initial={{ opacity: 0 }}
      animate={{ opacity: 1 }}
      exit={{ opacity: 0 }}
    >
      Contenido animado
    </motion.div>
  );
};
```

- **Transiciones:** Definir transiciones suaves entre estados de los componentes, como al montar o desmontar elementos.

- **Ejemplo de transición:**

```
import { motion } from 'framer-motion';

const MyComponent = () => {
  return (
    <motion.div
      initial={{ opacity: 0 }}
      animate={{ opacity: 1 }}
      exit={{ opacity: 0 }}
    >
      Contenido animado
    </motion.div>
  );
};
```

- **Animaciones de listas:** Manejar animaciones al agregar, eliminar o reordenar elementos en una lista.
- **Ejemplo de animación de lista:**

```
import { motion } from 'framer-motion';

const MyList = ({ items }) => {
  return (
```

```
<ul>
  {items.map(item => (
    <motion.li
      key={item.id}
      initial={{ opacity: 0 }}
      animate={{ opacity: 1 }}
      exit={{ opacity: 0 }}
    >
      {item.name}
    </motion.li>
  )));
</ul>
);
};
```

- **Animaciones de componentes:** Aplicar animaciones a componentes individuales, como botones o tarjetas, para mejorar la interacción del usuario.
- **Ejemplo de animación de componente:**

```
import { motion } from 'framer-motion';

const MyButton = () => {
  return (
    <motion.button
      initial={{ scale: 1 }}
      whileHover={{ scale: 1.1 }}
      whileTap={{ scale: 0.9 }}
    >
      Botón animado
    </motion.button>
  );
};
```

Optimización de carga y lazy loading

- **React.lazy**: Permite la carga diferida de componentes, mejorando el rendimiento al dividir el código.
- **Ejemplo**:

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

const MyComponent = lazy(() => import('./MyComponent'));
```

- **Suspense**: Componente que permite manejar la carga de componentes de manera más elegante, mostrando un fallback mientras se carga el contenido.

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

const App = () => {
  return (
    <Suspense fallback={<div>Cargando...</div>}>
      <MyComponent />
    </Suspense>
  );
};
```

- **Code splitting**: Estrategia para dividir el código en partes más pequeñas y cargarlas bajo demanda.
- **Optimización de bundle**: Técnicas para reducir el tamaño del bundle final, como la eliminación de código muerto y la minificación.

PWA y modo offline

- **Objetivo:** Convertir una aplicación React en una Progressive Web App (PWA) para mejorar la experiencia del usuario, incluso cuando está offline.
- **Service Workers:** Scripts que el navegador ejecuta en segundo plano, separados de una página web, que permiten funcionalidades que no necesitan una página web o interacción del usuario. Son fundamentales para las PWAs, ya que permiten el caching y la carga offline.
- **Ejemplo:**

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js')
      .then(registration => {
        console.log('Service Worker registrado con éxito:', registration);
      })
      .catch(error => {
        console.error('Error al registrar el Service Worker:', error);
      });
  });
}
```

- **Caching estratégico:** Implementar estrategias de caching para almacenar recursos en caché y servirlos rápidamente, incluso sin conexión a Internet.
- **Ejemplo:**

```
const cacheName = 'my-app-cache';
const urlsToCache = [
  '/',
  '/index.html',
  '/styles.css',
  '/script.js',
];

self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(cacheName)
      .then(cache => {
        return cache.addAll(urlsToCache);
      })
  );
});
```

- **Notificaciones push:** Permiten enviar notificaciones a los usuarios incluso cuando la aplicación no está abierta, mejorando la re-engagement.
- **Ejemplo:**

```
if ('serviceWorker' in navigator) {  
  navigator.serviceWorker.ready.then(registration => {  
    registration.showNotification('Hola, mundo!', {  
      body: 'Esta es una notificación push.',  
      icon: '/icon.png'  
    });  
  });  
}
```

- **UX offline-first:** Diseñar la experiencia del usuario teniendo en cuenta el uso offline, asegurando que la aplicación siga siendo funcional y útil sin conexión.
- **Ejemplo:**

```
const App = () => {  
  return (  
    <div>  
      <h1>Mi Aplicación PWA</h1>  
      <p>Funciona sin conexión a Internet.</p>  
    </div>  
  );  
};
```

Internacionalización (i18n)

- **react-i18next**: Biblioteca para la internacionalización en aplicaciones React, que permite la carga de traducciones y la gestión de idiomas de manera sencilla.
- **Ejemplo**:

```
import { useTranslation } from 'react-i18next';

const MyComponent = () => {
  const { t } = useTranslation();

  return <h1>{t('welcome_message')}</h1>;
};
```

- **Carga diferida de idiomas**: Cargar solo los idiomas necesarios en lugar de todos al inicio, mejorando el rendimiento.

- **Ejemplo**:

```
import i18n from 'i18next';

i18n.init({
  lng: 'en',
  resources: {
    en: {
      translation: {
        welcome_message: "Welcome to my app!"
      }
    },
    es: {
      translation: {
        welcome_message: "|Bienvenido a mi aplicación!"
      }
    }
  }
});
```

- **Fallback**: Idioma por defecto que se utiliza si no hay traducción disponible para el idioma seleccionado.
- **Ejemplo**:

```
i18n.init({
  lng: 'en',
  fallbackLng: 'es',
  resources: {
    en: {
      translation: {
        welcome_message: "Welcome to my app!"
      }
    },
    es: {
```

```
        translation: {
          welcome_message: "¡Bienvenido a mi aplicación!"
        }
      }
    });
  
```

- **Pluralización:** Manejo de diferentes formas de una palabra según la cantidad (singular/plural).
- **Integración con backend:** Cargar traducciones desde un servidor o API, permitiendo actualizaciones dinámicas.
- **Ejemplo:**

```
import i18n from 'i18next';

i18n.init({
  lng: 'en',
  backend: {
    loadPath: '/locales/{{lng}}/{{ns}}.json'
  }
});

```javascript
import i18n from 'i18next';

i18n.init({
 lng: 'en',
 backend: {
 loadPath: '/locales/{{lng}}/{{ns}}.json'
 }
});

```

## SEO y accesibilidad

- **React Helmet:** Componente para gestionar el head del documento, permitiendo la modificación dinámica de metadatos como título y descripción.
- **Ejemplo:**

```
import { Helmet } from 'react-helmet';

const MyComponent = () => {
 return (
 <div>
 <Helmet>
 <title>Mi Aplicación</title>
 <meta name="description" content="Descripción de mi aplicación" />
 </Helmet>
 <h1>Hola, mundo!</h1>
 </div>
);
};
```

- **Roles ARIA:** Atributos que mejoran la accesibilidad al proporcionar información sobre el propósito y el estado de los elementos de la interfaz.

- **Ejemplo:**

```
<button aria-label="Cerrar">X</button>
```

- **Focus management:** Estrategias para gestionar el foco en la interfaz, asegurando que los elementos interactivos sean accesibles y fáciles de usar.
- **Ejemplo:**

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

const MyComponent = () => {
 const inputRef = useRef(null);

 useEffect(() => {
 inputRef.current.focus();
 }, []);

 return <input ref={inputRef} type="text" />;
};
```

- **Herramientas axe-core/react-axe:** Librerías para analizar y mejorar la accesibilidad de las aplicaciones React.
- **Ejemplo:**

```
import { axe } from 'jest-axe';

test('should have no accessibility violations', async () => {
 const { container } = render(<MyComponent />);
 const results = await axe(container);
 expect(results).toHaveNoViolations();
});
```

- **PWA SEO:** Técnicas para mejorar el SEO de las aplicaciones PWA, como el uso de metadatos adecuados y la optimización del rendimiento.

- **Ejemplo:**

```
import { Helmet } from 'react-helmet';

const MyComponent = () => {
 return (
 <div>
 <Helmet>
 <title>Mi Aplicación</title>
 <meta name="description" content="Descripción de mi aplicación" />
 </Helmet>
 <h1>Hola, mundo!</h1>
 </div>
);
};
```

## Seguridad básica y avanzada

- **XSS (Cross-Site Scripting)**: Vulnerabilidad que permite a un atacante injectar scripts maliciosos en el contenido de una página web. Para prevenirlo, se deben sanitizar todas las entradas del usuario y utilizar técnicas como Content Security Policy (CSP).
- **Ejemplo:**

```
import DOMPurify from 'dompurify';

const MyComponent = ({ userInput }) => {
 const cleanInput = DOMPurify.sanitize(userInput);
 return <div dangerouslySetInnerHTML={{ __html: cleanInput }} />;
};
```

- **CSRF (Cross-Site Request Forgery)**: Ataque que engaña a un usuario para que realice acciones no deseadas en una aplicación web en la que está autenticado. Se puede mitigar utilizando tokens CSRF y verificando el origen de las solicitudes.

- **Ejemplo:**

```
import { useEffect } from 'react';
import { useLocation } from 'react-router-dom';

const MyComponent = () => {
 const location = useLocation();

 useEffect(() => {
 const token = localStorage.getItem('csrfToken');
 fetch('/api/protected', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 'X-CSRF-Token': token,
 },
 body: JSON.stringify({ data: 'test' }),
 });
 }, [location]);

 return <div>Hola, mundo!</div>;
};
```

- **Manejo de tokens**: Estrategias para gestionar la autenticación y autorización de usuarios mediante tokens (JWT, OAuth). Es importante almacenar los tokens de forma segura y renovarlos periódicamente.
- **Ejemplo:**

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

const MyComponent = () => {
 useEffect(() => {
```

```

const token = localStorage.getItem('authToken');
if (token) {
 fetch('/api/protected', {
 headers: {
 Authorization: `Bearer ${token}`,
 },
 });
}
, []);
};

return <div>Hola, mundo!</div>;
};

```

- **Rutas protegidas:** Implementación de middleware y componentes que restringen el acceso a ciertas rutas de la aplicación según el rol del usuario.
- **Ejemplo:**

```

import { Route, Redirect } from 'react-router-dom';

const PrivateRoute = ({ component: Component, ...rest }) => {
 const isAuthenticated = !!localStorage.getItem('authToken');
 return (
 <Route
 {...rest}
 render={props =>
 isAuthenticated ? <Component {...props} /> : <Redirect to="/login" />
 }
 />
);
};

```

- **Roles:** Definición de diferentes niveles de acceso y permisos para los usuarios en la aplicación.
- **Ejemplo:**

```

import { Route, Redirect } from 'react-router-dom';

const AdminRoute = ({ component: Component, ...rest }) => {
 const userRole = localStorage.getItem('userRole');
 return (
 <Route
 {...rest}
 render={props =>
 userRole === 'admin' ? <Component {...props} /> : <Redirect to="/unauthorized" />
 }
 />
);
};

```

- **Buenas prácticas de producción:** Recomendaciones para asegurar la aplicación en un entorno de producción, como la configuración adecuada de CORS, la gestión de errores y la monitorización de la seguridad.
- **Ejemplo:**

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

const MyComponent = () => {
 useEffect(() => {
 const token = localStorage.getItem('authToken');
 if (token) {
 fetch('/api/protected', {
 headers: {
 Authorization: `Bearer ${token}`,
 },
 });
 }
 }, []);
}

return <div>Hola, mundo!</div>;
};
```

# CI/CD y despliegue

## Objetivos del capítulo

- Aprender a configurar flujos de CI/CD para aplicaciones React
- Conocer las principales plataformas de despliegue para frontend
- Implementar procesos automáticos de testing y deploy
- Conocer estrategias de despliegue para apps React
- Configurar entornos de desarrollo, staging y producción

## Integración Continua con GitHub Actions

GitHub Actions permite automatizar testing y despliegue directamente desde tu repositorio:

```
.github/workflows/ci.yml
name: React CI

on:
 push:
 branches: [main]
 pull_request:
 branches: [main]

jobs:
 test:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3
 - name: Set up Node.js
 uses: actions/setup-node@v3
 with:
 node-version: 18
 cache: 'npm'
 - name: Install dependencies
 run: npm ci
 - name: Run tests
 run: npm test
 - name: Run linter
 run: npm run lint

 build:
 needs: test
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3
 - name: Set up Node.js
 uses: actions/setup-node@v3
 with:
 node-version: 18
 cache: 'npm'
 - name: Install dependencies
 run: npm ci
 - name: Build
```

```
run: npm run build
- name: Upload build artifacts
 uses: actions/upload-artifact@v3
 with:
 name: build
 path: build/
```

## Despliegue en Vercel

Vercel ofrece una experiencia optimizada para apps React y Next.js:

1. Conecta tu repositorio de GitHub/GitLab/Bitbucket
2. Configura variables de entorno
3. Cada push generará un deploy automático

Para desplegar manualmente con la CLI:

```
Instalar Vercel CLI
npm install -g vercel

Login
vercel login

Deploy (desde la carpeta del proyecto)
vercel
```

La configuración se guarda en `vercel.json`:

```
{
 "version": 2,
 "builds": [
 {
 "src": "package.json",
 "use": "@vercel/static-build",
 "config": { "distDir": "build" }
 }
],
 "routes": [
 { "handle": "filesystem" },
 { "src": "./*", "dest": "/index.html" }
]
}
```

## Despliegue en Netlify

Netlify es una excelente alternativa para sitios JAMstack:

1. Conecta tu repositorio
2. Configura comando de build: `npm run build`
3. Directorio de publicación: `build` o `dist`

Para configuración avanzada, usa [netlify.toml](#):

```
[build]
 command = "npm run build"
 publish = "build"

[[redirects]]
 from = "*"
 to = "/index.html"
 status = 200

[context.production.environment]
 REACT_APP_API_URL = "https://api.produccion.com"

[context.deploy-preview.environment]
 REACT_APP_API_URL = "https://api-staging.com"
```

## AWS Amplify

AWS Amplify ofrece una solución end-to-end para apps React:

1. Conecta tu repositorio
2. Configura entornos
3. Define variables de entorno según branch

```
amplify.yml
version: 1
frontend:
 phases:
 preBuild:
 commands:
 - npm ci
 build:
 commands:
 - npm run build
artifacts:
 baseDirectory: build
 files:
 - '**/*'
cache:
 paths:
 - node_modules/**/*
```

## Despliegue con Docker

Para entornos más controlados:

```
Dockerfile
FROM node:18-alpine as build
WORKDIR /app
```

```
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Y un archivo `nginx.conf` para gestionar las rutas:

```
server {
 listen 80;

 location / {
 root /usr/share/nginx/html;
 index index.html;
 try_files $uri $uri/ /index.html;
 }
}
```

## Estrategias de despliegue

1. **Blue/Green Deployment:** Mantén dos entornos idénticos, uno en producción (blue) y otro para la nueva versión (green). Cambia el tráfico al green cuando esté listo.
2. **Canary Releases:** Despliega la nueva versión para un pequeño porcentaje de usuarios primero, y ve incrementando gradualmente.
3. **Feature Flags:** Permite activar/desactivar funcionalidades sin necesidad de redesplegar.

## Buenas prácticas

1. **Automatiza todo:** Tests, builds y despliegues
2. **Variables de entorno:** Usa archivos `.env` para cada entorno
3. **Versionado:** Etiqueta cada release con un número de versión
4. **Rollback plan:** Asegúrate de poder volver a versiones anteriores
5. **Monitoreo:** Configura alertas para detectar problemas post-despliegue
6. **Performance:** Optimiza el bundle con code-splitting y lazy loading

## Ejercicio práctico

Configura un workflow completo de CI/CD para una app React:

1. Crea un repositorio en GitHub
2. Configura GitHub Actions para tests
3. Despliega automáticamente a Netlify o Vercel
4. Implementa diferentes variables de entorno para dev y prod

## 5. Configura una preview para cada PR

### Recursos adicionales

- [GitHub Actions Documentation](#)
  - [Vercel Documentation](#)
  - [Netlify Documentation](#)
  - [AWS Amplify Documentation](#)
  - [Docker Documentation](#)
-

## Referencias rápidas, checklist y mentalidad de depuración

- **Checklist de hooks:** Asegurarse de que los hooks se usen correctamente, siguiendo las reglas de los hooks.
- **Checklist de props:** Verificar que las props se pasen correctamente a los componentes y que se manejen adecuadamente.
- **Checklist de estados:** Asegurarse de que el estado se gestione correctamente y se actualice de manera predecible.
- **Checklist de debugging:** Utilizar herramientas como DevTools y Profiler para identificar y solucionar problemas de rendimiento y comportamiento en la aplicación.
- **Checklist de rendimiento:** Evaluar el rendimiento de la aplicación y aplicar técnicas de optimización cuando sea necesario.
- **Mentalidad de depuración:** Adoptar un enfoque sistemático para identificar y resolver problemas en la aplicación, utilizando técnicas como la reproducción de errores, el análisis de logs y la revisión del código.
- **Ejemplo de uso de React DevTools:** Inspeccionar el árbol de componentes, ver props y estado, y analizar renders.
- **Ejemplo de uso de Profiler:** Medir el rendimiento de los componentes y detectar renders innecesarios.
- **Ejemplo de uso de Error Boundaries:** Capturar errores en componentes hijos y mostrar un mensaje de error amigable.
- **Ejemplo de uso de Suspense:** Cargar componentes de manera asíncrona y mostrar un fallback mientras se cargan.
- **Ejemplo de uso de Portales:** Renderizar componentes en un nodo DOM diferente al de su parente, útil para modales y tooltips.

# Integración con librerías populares

## Objetivos de esta sección

- Aprender a integrar React con las librerías más populares del ecosistema
- Conocer las mejores prácticas de integración
- Entender cuándo usar cada librería según las necesidades del proyecto
- Configurar correctamente cada librería en un proyecto React

## Librerías de UI Components

### Tailwind CSS

Framework de utilidades para diseñar rápidamente interfaces responsivas y personalizables.

#### Instalación y configuración:

```
Instalación
npm install tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

#### Configuración ([tailwind.config.js](#)):

```
/** @type {import('tailwindcss').Config} */
module.exports = {
 content: [
 './src/**/*.{js,jsx,ts,tsx}",
],
 theme: {
 extend: {
 colors: {
 primary: "#0070f3",
 secondary: "#ff4081",
 }
 },
 },
 plugins: [],
}
```

#### Incluir en el proyecto ([index.css](#)):

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

#### Ejemplo de uso:

```
function Button({ children, primary }) {
 return (
 <button className={`${px-4 py-2 rounded-md font-medium ${primary ? 'bg-primary text-white' : 'bg-gray-200 text-gray-800'} } hover:opacity-90 transition-opacity`}>
 {children}
 </button>
);
}
```

## Material UI

Biblioteca de componentes React que implementa el diseño de Google Material Design.

### Instalación:

```
npm install @mui/material @emotion/react @emotion/styled @mui/icons-material
```

### Configuración con tema personalizado:

```
import { createTheme, ThemeProvider } from '@mui/material/styles';

const theme = createTheme({
 palette: {
 primary: {
 main: '#1976d2',
 },
 secondary: {
 main: '#dc004e',
 },
 },
 typography: {
 fontFamily: 'Roboto, Arial, sans-serif',
 button: {
 textTransform: 'none',
 },
 },
});

function App() {
 return (
 <ThemeProvider theme={theme}>
 {/* Tu aplicación aquí */}
 </ThemeProvider>
);
}
```

### Ejemplo de componentes:

```
import { Button, TextField, Card, CardContent, Typography } from '@mui/material';

function LoginForm() {
 return (
 <Card sx={{ maxWidth: 400, mx: 'auto', mt: 4 }}>
 <CardContent>
 <Typography variant="h5" component="h2" gutterBottom>
 Iniciar sesión
 </Typography>
 <TextField
 label="Email"
 variant="outlined"
 fullWidth
 margin="normal"
 />
 <TextField
 label="Contraseña"
 type="password"
 variant="outlined"
 fullWidth
 margin="normal"
 />
 <Button
 variant="contained"
 color="primary"
 fullWidth
 sx={{ mt: 2 }}
 >
 Entrar
 </Button>
 </CardContent>
 </Card>
);
}
```

## Chakra UI

Biblioteca de componentes accesibles y personalizables para React.

### Instalación:

```
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

### Configuración básica:

```
import { ChakraProvider, extendTheme } from '@chakra-ui/react';

const theme = extendTheme({
 colors: {
 brand: {
 100: '#f7fafc',
 }
 }
});
```

```

 500: '#319795',
 900: '#1a202c',
 },
},
fonts: {
 heading: 'Montserrat, sans-serif',
 body: 'Inter, system-ui, sans-serif',
},
});

function App() {
 return (
 <ChakraProvider theme={theme}>
 {/* Tu aplicación aquí */}
 </ChakraProvider>
);
}

```

### Ejemplo de componentes:

```

import { Box, Button, FormControl, FormLabel, Input, VStack } from '@chakra-ui/react';

function ContactForm() {
 return (
 <Box bg="white" p={6} rounded="md" shadow="md">
 <VStack spacing={4}>
 <FormControl isRequired>
 <FormLabel>Nombre</FormLabel>
 <Input placeholder="Tu nombre" />
 </FormControl>

 <FormControl isRequired>
 <FormLabel>Email</FormLabel>
 <Input type="email" placeholder="ejemplo@email.com" />
 </FormControl>

 <FormControl>
 <FormLabel>Mensaje</FormLabel>
 <Input as="textArea" h="100px" placeholder="Tu mensaje" />
 </FormControl>

 <Button colorScheme="teal" width="full">
 Enviar mensaje
 </Button>
 </VStack>
 </Box>
);
}

```

### Enrutamiento con React Router v6

#### Instalación:

```
npm install react-router-dom
```

## Configuración básica:

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

function App() {
 return (
 <BrowserRouter>
 <nav>

 <Link to="/">Inicio</Link>
 <Link to="/about">Acerca de</Link>
 <Link to="/dashboard">Dashboard</Link>

 </nav>

 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/about" element={<About />} />
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="*" element={<NotFound />} />
 </Routes>
 </BrowserRouter>
);
}

}
```

## Rutas anidadas y Layouts:

```
function App() {
 return (
 <BrowserRouter>
 <Routes>
 <Route path="/" element={<Layout />}>
 <Route index element={<Home />} />
 <Route path="about" element={<About />} />
 <Route path="dashboard" element={<Dashboard />} />
 <Route path="products" element={<Products />}>
 <Route index element={<ProductsList />} />
 <Route path=":id" element={<ProductDetail />} />
 </Route>
 <Route path="*" element={<NotFound />} />
 </Route>
 </Routes>
 </BrowserRouter>
);
}

function Layout() {
 return (
 <div>
```

```

<header>
 <nav>/* Links de navegación */</nav>
</header>
<main>
 <Outlet /> /* Aquí se renderizan las rutas hijas */
</main>
<footer>© 2025 Mi Aplicación</footer>
</div>
);
}

```

## Navegación programática y parámetros:

```

import { useNavigate, useParams } from 'react-router-dom';

function ProductDetail() {
 const { id } = useParams();
 const navigate = useNavigate();

 const goBack = () => navigate(-1);
 const goToHome = () => navigate('/');

 return (
 <div>
 <h2>Detalles del producto {id}</h2>
 <button onClick={goBack}>Volver</button>
 <button onClick={goToHome}>Ir a inicio</button>
 </div>
);
}

```

## Gestión de datos remotos

### React Query

Biblioteca para la gestión de estados y la sincronización de datos en aplicaciones React.

#### Instalación:

```
npm install react-query
```

#### Configuración básica:

```

import { QueryClient, QueryClientProvider } from 'react-query';
import { ReactQueryDevtools } from 'react-query/devtools';

const queryClient = new QueryClient({
 defaultOptions: {
 queries: {
 refetchOnWindowFocus: false,
 }
 }
});

```

```
 staleTime: 60000, // 1 minuto
 },
},
});

function App() {
return (
<QueryClientProvider client={queryClient}>
/* Tu aplicación aquí */
<ReactQueryDevtools initialIsOpen={false} />
</QueryClientProvider>
);
}
```

### Ejemplo de uso básico:

```
// Importamos los hooks necesarios de react-query
import { useQuery, useMutation, useQueryClient } from 'react-query';

// COMPONENTE DE CONSULTA DE DATOS
function Products() {
// PASO 1: Usamos useQuery para obtener datos
// - 'products' es una clave única para identificar esta consulta en la caché
// - El segundo argumento es una función que devuelve una promesa con los datos
const { isLoading, error, data } = useQuery(
 'products', // Clave única para la consulta
 () => fetch('/api/products').then(res => res.json()) // Función de obtención de datos
);

// PASO 2: Manejo de estados de carga y error
// React Query proporciona automáticamente estados como isLoading y error
if (isLoading) return <p>Cargando...</p>;
if (error) return <p>Error: {error.message}</p>

// PASO 3: Renderizamos los datos cuando estén disponibles
return (
<div>
 <h2>Productos</h2>

 {/* Iteramos sobre los datos obtenidos de la API */}
 {data.map(product => (
 <li key={product.id}>{product.name}
)));

</div>
);
}

// COMPONENTE PARA MUTACIÓN DE DATOS (CREACIÓN, ACTUALIZACIÓN, ELIMINACIÓN)
function AddProduct() {
// PASO 1: Obtenemos la instancia del cliente de React Query
// Esto nos permite interactuar con la caché global
const queryClient = useQueryClient();
```

```

// PASO 2: Configuramos la mutación con useMutation
const mutation = useMutation(
 // Función que realiza la mutación (POST a la API)
 (newProduct) => fetch('/api/products', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(newProduct),
 }).then(res => res.json()),

 // Opciones de configuración de la mutación
{
 // PASO 3: Configuramos comportamiento después del éxito
 onSuccess: () => {
 // Invalidamos la consulta 'products' para que se vuelva a cargar
 // Esto hace que la lista se actualice automáticamente con el nuevo producto
 queryClient.invalidateQueries('products');
 },
 // También podríamos configurar onError, onSettled, etc.
}
);

// PASO 4: Manejador para el envío del formulario
const handleSubmit = (e) => {
 e.preventDefault(); // Evitar el comportamiento predeterminado del formulario
 const name = e.target.name.value; // Obtenemos el valor del input
 mutation.mutate({ name }); // Ejecutamos la mutación con los datos
 e.target.reset(); // Limpiamos el formulario
};

// PASO 5: Renderizamos el formulario
return (
 <form onSubmit={handleSubmit}>
 {/* Campo para el nombre del producto */}
 <input name="name" placeholder="Nombre del producto" />

 {/* Botón que se deshabilita durante la mutación */}
 <button
 type="submit"
 disabled={mutation.isLoading} // Deshabilitamos durante la carga
 >
 {/* Texto dinámico según el estado */}
 {mutation.isLoading ? 'Añadiendo...' : 'Añadir producto'}
 </button>
 </form>
);
}

```

## SWR (Stale-While-Revalidate)

Estrategia de recuperación de datos para React que se centra en la simplicidad y la eficiencia.

### Instalación:

```
npm install swr
```

### Configuración básica:

```
import { SWRConfig } from 'swr';

function App() {
 return (
 <SWRConfig
 value={{
 fetcher: (url) => fetch(url).then(res => res.json()),
 revalidateOnFocus: false,
 dedupingInterval: 10000
 }}
 >
 {/* Tu aplicación aquí */}
 </SWRConfig>
);
}

}
```

### Ejemplo de uso:

```
import useSWR, { useSWRConfig } from 'swr';

function Profile() {
 const { data, error } = useSWR('/api/user');

 if (error) return <div>Error al cargar usuario</div>;
 if (!data) return <div>Cargando...</div>;

 return <div>Hola {data.name}!</div>;
}

// Mutación manual
function UpdateProfileButton() {
 const { mutate } = useSWRConfig();

 const updateProfile = async () => {
 const updatedUser = { name: 'Nuevo nombre' };

 // Actualiza la UI inmediatamente (optimistic UI)
 mutate('/api/user', updatedUser, false);

 // Envía la petición a la API
 await fetch('/api/user', {
 method: 'PUT',
 body: JSON.stringify(updatedUser)
 });

 // Revalida para asegurarse de que los datos están actualizados
 mutate('/api/user');
}
```

```
};

 return <button onClick={updateProfile}>Actualizar perfil</button>;
}
```

## Animaciones con Framer Motion

Biblioteca para animaciones en React que permite crear transiciones fluidas y complejas.

### Instalación:

```
npm install framer-motion
```

### Ejemplos básicos:

```
import { motion } from 'framer-motion';

// Animación simple
function FadeIn() {
 return (
 <motion.div
 initial={{ opacity: 0 }}
 animate={{ opacity: 1 }}
 exit={{ opacity: 0 }}
 transition={{ duration: 0.5 }}
 >
 Este contenido aparece con fade in
 </motion.div>
);
}

// Animación con variantes
function AnimatedList() {
 const container = {
 hidden: { opacity: 0 },
 show: {
 opacity: 1,
 transition: {
 staggerChildren: 0.3
 }
 }
 };

 const item = {
 hidden: { y: 20, opacity: 0 },
 show: { y: 0, opacity: 1 }
 };

 return (
 <motion.ul
 variants={container}
 initial="hidden"
```

```

 animate="show"
 >
 {[1, 2, 3, 4].map(index => (
 <motion.li key={index} variants={item}>
 Item {index}
 </motion.li>
))}
</motion.ul>
);
}

// Gestos de arrastrar
function DraggableItem() {
 return (
 <motion.div
 drag
 dragConstraints={{
 top: -50,
 left: -50,
 right: 50,
 bottom: 50,
 }}
 whileTap={{ scale: 0.9 }}
 whileHover={{ scale: 1.1 }}
 style={{
 width: 100,
 height: 100,
 background: 'blue',
 borderRadius: 20
 }}
 />
);
}

```

## Recomendaciones para elegir librerías

- 1. Evalúa el tamaño del bundle:** Librerías grandes pueden ralentizar la carga.
- 2. Considera el mantenimiento:** Verifica actividad en GitHub, número de issues, estrellas.
- 3. Compatibilidad con React:** Asegúrate de que sea compatible con tu versión de React.
- 4. Documentación:** Una buena documentación facilita la integración y resolución de problemas.
- 5. Comunidad:** Una comunidad activa significa más recursos y soporte.
- 6. Licencia:** Verifica que la licencia sea compatible con tu proyecto.
- 7. Personalización:** Evalúa qué tan fácil es personalizar la librería según tus necesidades.

## Ejercicio práctico

Crea una aplicación que combine:

- Rutas con React Router
- Componentes de UI con Chakra UI o Material UI
- Datos remotos con React Query o SWR
- Al menos una animación con Framer Motion

Este ejercicio te ayudará a entender cómo integrar varias librerías en un mismo proyecto.

## Ejercicios prácticos y mini proyectos guiados

- **Ejercicio 1:** Crear un formulario de inicio de sesión utilizando React Hook Form y Zod para la validación.
  - **Ejercicio 2:** Implementar un sistema de gestión de estado global utilizando Context API y useReducer.
  - **Ejercicio 3:** Crear un componente de lista de tareas utilizando useState y useEffect para gestionar el estado y los efectos secundarios.
  - **Mini-proyecto 1:** Crear una aplicación de lista de tareas utilizando React, con funcionalidades de agregar, eliminar y marcar tareas como completadas.
  - **Mini-proyecto 2:** Implementar un sistema de autenticación utilizando Firebase y React Router.
  - **Mini-proyecto 3:** Crear una aplicación de galería de imágenes utilizando una API pública y mostrar las imágenes en un grid.
-

# Hooks avanzados y patrones

## Introducción a los hooks avanzados

Los hooks avanzados de React permiten implementar patrones sofisticados y optimizar el rendimiento de la aplicación. En esta sección exploraremos hooks como `useReducer`, `useImperativeHandle`, `useDeferredValue` y `useTransition`, junto con patrones avanzados para crear hooks personalizados reutilizables.

### useReducer avanzado

Para gestionar estados complejos con flujos de datos más predecibles:

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

// Tipos de acciones - ayuda a evitar errores tipográficos
const ACTIONS = {
 ADD_TASK: 'add-task',
 TOGGLE_TASK: 'toggle-task',
 DELETE_TASK: 'delete-task',
 UPDATE_FILTER: 'update-filter',
 SET_LOADING: 'set-loading'
};

// Estado inicial complejo
const initialState = {
 tasks: [],
 filter: 'all', // all, active, completed
 isLoading: false,
 error: null
};

// Reducer con manejo de múltiples acciones y validaciones
function taskReducer(state, action) {
 switch (action.type) {
 case ACTIONS.SET_LOADING:
 return { ...state, isLoading: action.payload };

 case ACTIONS.ADD_TASK:
 // Validar que la tarea tenga texto
 if (!action.payload.text.trim()) return state;

 return {
 ...state,
 tasks: [...state.tasks, {
 id: Date.now(),
 text: action.payload.text,
 completed: false,
 createdAt: new Date()
 }]
 };

 case ACTIONS.TOGGLE_TASK:
 return {
 ...state,
```

```
tasks: state.tasks.map(task =>
 task.id === action.payload.id
 ? { ...task, completed: !task.completed }
 : task
)
};

case ACTIONS.DELETE_TASK:
 return {
 ...state,
 tasks: state.tasks.filter(task => task.id !== action.payload.id)
 };

case ACTIONS.UPDATE_FILTER:
 return { ...state, filter: action.payload };

default:
 throw new Error(`Acción no soportada: ${action.type}`);
}

}

function TaskManager() {
 const [state, dispatch] = useReducer(taskReducer, initialState);
 const { tasks, filter, isLoading } = state;

 // Función para añadir una tarea
 const addTask = (text) => {
 dispatch({ type: ACTIONS.ADD_TASK, payload: { text } });
 };

 // Función para cambiar el estado de una tarea
 const toggleTask = (id) => {
 dispatch({ type: ACTIONS.TOGGLE_TASK, payload: { id } });
 };

 // Función para eliminar una tarea
 const deleteTask = (id) => {
 dispatch({ type: ACTIONS.DELETE_TASK, payload: { id } });
 };

 // Función para actualizar el filtro
 const updateFilter = (filter) => {
 dispatch({ type: ACTIONS.UPDATE_FILTER, payload: filter });
 };

 // Filtrar tareas según el filtro activo
 const filteredTasks = tasks.filter(task => {
 if (filter === 'active') return !task.completed;
 if (filter === 'completed') return task.completed;
 return true; // 'all'
 });

 // Simulación de carga de tareas
 useEffect(() => {
 const loadTasks = async () => {
 dispatch({ type: ACTIONS.SET_LOADING, payload: true });
 // Simulación de carga...
 dispatch({ type: ACTIONS.SET_LOADING, payload: false });
 };
 loadTasks();
 });
}
```

```
try {
 const response = await fetch('/api/tasks');
 const data = await response.json();

 // Añadir cada tarea al estado
 data.forEach(task => {
 dispatch({
 type: ACTIONS.ADD_TASK,
 payload: { text: task.text }
 });
 });
} catch (error) {
 console.error('Error cargando tareas:', error);
} finally {
 dispatch({ type: ACTIONS.SET_LOADING, payload: false });
}

};

loadTasks();
}, []));

if (isLoading) return <div>Cargando tareas...</div>

return (
 <div>
 <h2>Gestor de tareas</h2>

 {/* Formulario para añadir tareas */}
 <form onSubmit={(e) => {
 e.preventDefault();
 const text = e.target.task.value;
 addTask(text);
 e.target.reset();
 }}>
 <input name="task" placeholder="Nueva tarea" required />
 <button type="submit">Añadir</button>
 </form>

 {/* Filtros */}
 <div>
 <button
 className={filter === 'all' ? 'active' : ''}
 onClick={() => updateFilter('all')}
 >
 Todas
 </button>
 <button
 className={filter === 'active' ? 'active' : ''}
 onClick={() => updateFilter('active')}
 >
 Activas
 </button>
 <button
 className={filter === 'completed' ? 'active' : ''}
 onClick={() => updateFilter('completed')}
 >
 Completadas
 </button>
 </div>
 </div>
)
```

```

 Completadas
 </button>
 </div>

 /* Lista de tareas */

 {filteredTasks.map(task => (
 <li key={task.id}>
 <input
 type="checkbox"
 checked={task.completed}
 onChange={() => toggleTask(task.id)}
 />
 <span style={{
 textDecoration: task.completed ? 'line-through' : 'none'
 }}>
 {task.text}

 <button onClick={() => deleteTask(task.id)}>Eliminar</button>

))}

 </div>
);
}

```

## useImperativeHandle para referencias avanzadas

Este hook permite personalizar la instancia que se expone cuando se usa `ref`:

```

import { useRef, useImperativeHandle, forwardRef } from 'react';

// Componente hijo con forwardRef
const CustomInput = forwardRef((props, ref) => {
 const inputRef = useRef(null);

 // Exponemos métodos específicos al componente padre
 useImperativeHandle(ref, () => ({
 // Solo exponemos los métodos que queremos
 focus: () => {
 inputRef.current.focus();
 },
 clear: () => {
 inputRef.current.value = '';
 },
 getValue: () => {
 return inputRef.current.value;
 },
 // No exponemos otros métodos nativos del input
 }));

 return <input ref={inputRef} {...props} />;
});

```

```
// Componente padre que usa la ref personalizada
function FormWithCustomInput() {
 const inputRef = useRef(null);

 const handleSubmit = (e) => {
 e.preventDefault();
 alert(`Valor: ${inputRef.current.getValue()}`);
 inputRef.current.clear(); // Limpia el input
 };

 return (
 <form onSubmit={handleSubmit}>
 <CustomInput ref={inputRef} placeholder="Escribe algo" />
 <button type="button" onClick={() => inputRef.current.focus()}>
 Enfocar
 </button>
 <button type="submit">Enviar</button>
 </form>
);
}
```

## useDeferredValue para performance

Este hook permite retrasar la actualización de valores costosos para evitar bloquear la interfaz:

```
import { useState, useDeferredValue, useMemo } from 'react';

function SearchResults() {
 const [query, setQuery] = useState('');
 // El valor diferido se actualiza después de renderizar lo prioritario
 const deferredQuery = useDeferredValue(query);

 // Lista de elementos (simulada)
 const allItems = new Array(10000).fill().map((_, i) => ({
 id: i,
 name: `Item ${i + 1}`
 }));

 // Filtrar usando el valor diferido (operación costosa)
 const filteredItems = useMemo(() => {
 console.log(`Filtrando con query: "${deferredQuery}"`);
 if (!deferredQuery) return allItems.slice(0, 100); // Solo mostrar los primeros
 100 si no hay búsqueda

 return allItems.filter(item =>
 item.name.toLowerCase().includes(deferredQuery.toLowerCase())
);
 }, [deferredQuery, allItems]);

 // Muestra visualmente si el valor diferido está retrasado
 const isStale = query !== deferredQuery;

 return (
 <div>
```

```

<input
 value={query}
 onChange={e => setQuery(e.target.value)}
 placeholder="Buscar...">
/>

{isStale && <p>Cargando resultados...</p>}

 {filteredItems.map(item => (
 <li key={item.id}>{item.name}
)))

</div>
);
}

```

## useTransition para UI responsiva

Este hook permite marcar actualizaciones de estado como no urgentes:

```

import { useState, useTransition } from 'react';

function TabSelector() {
 // PASO 1: Inicializamos el hook useTransition que nos da:
 // - isPending: un booleano que indica si hay una transición en curso
 // - startTransition: función para marcar actualizaciones como menos prioritarias
 const [isPending, startTransition] = useTransition();

 // PASO 2: Definimos dos estados:
 // - tab: almacena la pestaña actualmente seleccionada
 // - tabContent: almacena el contenido que se muestra (esto cambiará con transición)
 const [tab, setTab] = useState('home');
 const [tabContent, setTabContent] = useState('Contenido del Home');

 // PASO 3: Datos simulados para cada pestaña
 // En una aplicación real, estos podrían venir de una API
 const tabData = {
 home: 'Contenido del Home',
 profile: 'Información del perfil del usuario',
 settings: 'Configuración de la aplicación',
 dashboard: 'Panel con estadísticas y gráficos'
 };

 // PASO 4: Función que maneja el cambio de pestañas sin bloquear la interfaz
 const selectTab = (newTab) => {
 // PASO 4.1: Actualizamos inmediatamente la pestaña seleccionada
 // Esta actualización es prioritaria y debe reflejarse de inmediato en la UI
 setTab(newTab);

 // PASO 4.2: Envolvemos la actualización del contenido en startTransition
 // Esto marca la actualización como menos prioritaria, permitiendo que la UI siga
 // respondiendo
 startTransition(() => {

```

```
// PASO 4.3: Simulamos una operación pesada que normalmente bloquearía la UI
// En un caso real, esto podría ser una consulta a API o un cálculo complejo
const start = performance.now();
while (performance.now() - start < 1000) {
 // Bloqueamos el hilo por 1 segundo intencionalmente para simular carga pesada
}

// PASO 4.4: Finalmente actualizamos el contenido de la pestaña
// Esta actualización ocurrirá después de que React termine con tareas más
prioritarias
 setTabContent(tabData[newTab]);
};

return (
 <div>
 {/* PASO 5: Renderizamos los botones para cada pestaña */}
 <div style={{ display: 'flex', marginBottom: '1rem' }}>
 {Object.keys(tabData).map(tabName => (
 <button
 key={tabName}
 onClick={() => selectTab(tabName)} // PASO 5.1: Manejador para cambiar de
pestaña
 disabled={isPending} // PASO 5.2: Deshabilitamos botones durante la
transición
 style={{
 fontWeight: tab === tabName ? 'bold' : 'normal', // PASO 5.3: Destacamos
la pestaña activa
 margin: '0 0.5rem'
 }}
 >
 {tabName}
 </button>
)))
 </div>
 {/* PASO 6: Mostramos un indicador de carga o el contenido según el estado de la
transición */}
 {isPending ? (
 <div>Cargando contenido...</div> // PASO 6.1: Feedback visual durante la
transición
) : (
 <div>{tabContent}</div> // PASO 6.2: Contenido de la pestaña cuando no hay
transición
)}
 </div>
);
}
```

## Patrones avanzados para custom hooks

### 1. Hook Compuesto

Crear una familia de hooks relacionados que trabajen juntos:

```
// useAuth.js - Hook compuesto para manejo de autenticación
import { createContext, useContext, useState, useEffect } from 'react';

// PASO 1: Creamos un contexto para almacenar y proveer el estado de autenticación
const AuthContext = createContext(null);

// PASO 2: Componente proveedor que envuelve nuestra aplicación
export function AuthProvider({ children }) {
 // PASO 2.1: Obtenemos toda la lógica de autenticación del hook interno
 const auth = useProvideAuth();

 // PASO 2.2: Proveemos los valores de autenticación a todos los componentes hijos
 return (
 <AuthContext.Provider value={auth}>
 {children}
 </AuthContext.Provider>
);
}

// PASO 3: Hook público que los componentes usarán para acceder al contexto
export function useAuth() {
 // PASO 3.1: Obtenemos el contexto de autenticación
 const context = useContext(AuthContext);

 // PASO 3.2: Verificamos que el hook se use dentro del Provider
 if (!context) {
 throw new Error('useAuth debe usarse dentro de AuthProvider');
 }

 return context;
}

// PASO 4: Hook interno que implementa toda la lógica de autenticación
function useProvideAuth() {
 // PASO 4.1: Estados para el usuario autenticado y estado de carga
 const [user, setUser] = useState(null);
 const [isLoading, setLoading] = useState(true);

 // PASO 4.2: Función para iniciar sesión
 const login = async (email, password) => {
 setLoading(true);
 try {
 // PASO 4.2.1: Realizamos petición a la API de login
 const response = await fetch('/api/login', {
 method: 'POST',
 body: JSON.stringify({ email, password })
 });

 // PASO 4.2.2: Procesamos la respuesta y actualizamos el estado
 const data = await response.json();
 setUser(data.user);
 return data.user;
 } catch (error) {
 // PASO 4.2.3: Manejamos errores de autenticación
 console.error('Error al iniciar sesión:', error);
 }
 };
}
```

```

 throw error;
 } finally {
 // PASO 4.2.4: Terminamos el estado de carga independientemente del resultado
 setLoading(false);
 }
};

// PASO 4.3: Función para cerrar sesión
const logout = async () => {
 await fetch('/api/logout');
 setUser(null);
};

// PASO 4.4: Efecto para verificar si el usuario está autenticado al cargar la
// aplicación
useEffect(() => {
 // PASO 4.4.1: Petición a la API para obtener el usuario actual
 fetch('/api/me')
 .then(res => res.json())
 .then(data => {
 // PASO 4.4.2: Actualizamos el estado con los datos del usuario
 setUser(data.user || null);
 setLoading(false);
 })
 .catch(() => {
 // PASO 4.4.3: En caso de error, asumimos que no hay usuario autenticado
 setUser(null);
 setLoading(false);
 });
}, []);

return {
 user,
 isLoading,
 isAuthenticated: !!user,
 login,
 logout
};
}

```

Uso del hook compuesto:

```

// PASO 5: Ejemplo de uso del patrón de hook compuesto

// PASO 5.1: Componente raíz que proporciona el contexto de autenticación a toda la
// app
function App() {
 return (
 // PASO 5.1.1: Envolvemos toda la aplicación con elAuthProvider
 <AuthProvider>
 <AppContent />
 </AuthProvider>
);
}

```

```
// PASO 5.2: Componente que consume el contexto de autenticación
function AppContent() {
 // PASO 5.2.1: Extraemos los valores y funciones que necesitamos del hook
 const { user, isAuthenticated, login, logout } = useAuth();

 // PASO 5.2.2: Renderizado condicional basado en el estado de autenticación
 if (!isAuthenticated) {
 // PASO 5.2.3: Si no está autenticado, mostramos el formulario de login
 return <LoginForm onLogin={login} />;
 }

 // PASO 5.2.4: Si está autenticado, mostramos la interfaz de usuario
 return (
 <div>
 <p>Bienvenido, {user.name}</p>
 <button onClick={logout}>Cerrar sesión</button>
 </div>
);
}
```

## 2. Hook con máquina de estados

Usar useReducer para implementar una máquina de estados:

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

// Estados posibles
const STATES = {
 IDLE: 'idle',
 LOADING: 'loading',
 SUCCESS: 'success',
 ERROR: 'error'
};

// Acciones posibles
const ACTIONS = {
 START_FETCH: 'START_FETCH',
 FETCH_SUCCESS: 'FETCH_SUCCESS',
 FETCH_ERROR: 'FETCH_ERROR',
 RESET: 'RESET'
};

// Reducer para la máquina de estados
function fetchReducer(state, action) {
 switch (action.type) {
 case ACTIONS.START_FETCH:
 return {
 status: STATES.LOADING,
 data: null,
 error: null
 };
 case ACTIONS.FETCH_SUCCESS:
 return {
 status: STATES.SUCCESS,
 data: action.payload,
 error: null
 };
 case ACTIONS.FETCH_ERROR:
 return {
 status: STATES.ERROR,
 data: null,
 error: action.error
 };
 case ACTIONS.RESET:
 return {
 status: STATES.IDLE,
 data: null,
 error: null
 };
 }
}
```

```
 return {
 status: STATES.SUCCESS,
 data: action.payload,
 error: null
 };

 case ACTIONS.FETCH_ERROR:
 return {
 status: STATES.ERROR,
 data: null,
 error: action.payload
 };

 case ACTIONS.RESET:
 return {
 status: STATES.IDLE,
 data: null,
 error: null
 };

 default:
 throw new Error(`Acción desconocida: ${action.type}`);
 }
}

// Hook que encapsula la máquina de estados
function useFetchMachine(fetchFn) {
 const [state, dispatch] = useReducer(fetchReducer, {
 status: STATES.IDLE,
 data: null,
 error: null
 });

 const { status, data, error } = state;

 const executeFetch = async (...args) => {
 dispatch({ type: ACTIONS.START_FETCH });

 try {
 const result = await fetchFn(...args);
 dispatch({
 type: ACTIONS.FETCH_SUCCESS,
 payload: result
 });
 return result;
 } catch (error) {
 dispatch({
 type: ACTIONS.FETCH_ERROR,
 payload: error.message
 });
 throw error;
 }
 };

 const reset = () => {
 dispatch({ type: ACTIONS.RESET });
 };
}
```

```
};

return {
 isIdle: status === STATES.IDLE,
 isLoading: status === STATES.LOADING,
 isSuccess: status === STATES.SUCCESS,
 isError: status === STATES.ERROR,
 data,
 error,
 executeFetch,
 reset
};
}
```

Uso del hook con máquina de estados:

```
function UserProfile({ userId }) {
 const fetchUser = async (id) => {
 const response = await fetch(`/api/users/${id}`);
 if (!response.ok) throw new Error('Error al cargar usuario');
 return response.json();
 };

 const {
 isIdle,
 isLoading,
 isSuccess,
 isError,
 data: user,
 error,
 executeFetch,
 reset
 } = useFetchMachine(fetchUser);

 useEffect(() => {
 if (userId) {
 executeFetch(userId);
 } else {
 reset();
 }
 }, [userId]);

 if (isIdle) return <p>Selecciona un usuario</p>;
 if (isLoading) return <p>Cargando...</p>;
 if (isError) return <p>Error: {error}</p>;

 return (
 <div>
 <h2>{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>Rol: {user.role}</p>
 </div>
);
}
```

## Ejercicio práctico: Crear un hook avanzado

Crea un hook personalizado `usePaginatedData` que:

1. Obtenga datos paginados de una API
2. Maneje estados de carga y error
3. Controle la navegación entre páginas
4. Permita configurar el tamaño de página
5. Implemente un caché básico para evitar peticiones repetidas

Utiliza hooks como `useReducer`, `useMemo` y `useCallback` para optimizar su funcionamiento.

## Consejos para crear hooks avanzados

1. **Divide y vencerás:** Separa la lógica en hooks más pequeños y específicos
2. **Expón solo lo necesario:** No sobrecargues la API del hook con detalles de implementación
3. **Usa TypeScript:** Define interfaces claras para la entrada y salida del hook
4. **Documenta con JSDoc:** Añade comentarios explicativos sobre parámetros y retorno
5. **Prueba de forma aislada:** Escribe tests unitarios para validar el comportamiento del hook
6. **Reutiliza entre proyectos:** Considera publicar tus hooks como paquetes npm

## Recursos adicionales

- [Documentación oficial de Hooks avanzados](#)
  - [Patrones de composición con hooks](#)
  - [Máquinas de estado con useReducer](#)
-

## 💡 Avanzado: Optimización, Concurrent Mode y patrones críticos

Esta sección cubre técnicas avanzadas que te permitirán construir aplicaciones React de alto rendimiento y escalables, dominando patrones críticos que separan a los desarrolladores experimentados de los principiantes.

### Error Boundaries: Manejo elegante de errores

Los Error Boundaries permiten capturar errores en cualquier parte de la jerarquía de componentes y mostrar una UI alternativa en lugar de que toda la aplicación falle.

```
// PASO 1: Creamos un componente de clase para manejar errores
import React, { Component } from 'react';

class ErrorBoundary extends Component {
 // PASO 2: Inicializamos el estado con hasError en false
 constructor(props) {
 super(props);
 this.state = { hasError: false, errorMessage: '' };
 }

 // PASO 3: Método que se ejecuta cuando ocurre un error en cualquier componente hijo
 // Este método actualiza el estado para reflejar que ocurrió un error
 static getDerivedStateFromError(error) {
 // Actualizamos el estado para mostrar la UI alternativa
 return { hasError: true };
 }

 // PASO 4: Método para manejar efectos secundarios del error (logging, analytics)
 componentDidCatch(error, errorThrown) {
 // Podemos enviar el error a un servicio como Sentry o t6t666
 console.error('Error capturado por boundary:', error, errorThrown);
 this.setState({ errorMessage: error.toString() });

 // En producción enviariámos este error a nuestro servicio de monitoreo
 // logErrorToService(error, errorThrown);
 }

 // PASO 5: Renderizamos la UI alternativa o los componentes hijos
 render() {
 // Si hay un error, mostramos la UI alternativa
 if (this.state.hasError) {
 return (
 <div className="error-container">
 <h2>¡Ups! Algo salió mal</h2>
 <p>Estamos trabajando para solucionarlo lo antes posible.</p>
 <details>
 <summary>Detalles técnicos</summary>
 <p>{this.state.errorMessage}</p>
 </details>
 <button onClick={() => window.location.reload()}>
 Intentar de nuevo
 </button>
 </div>
);
 }
 }
}
```

```

);
}

// Si no hay error, renderizamos normalmente los componentes hijos
return this.props.children;
}
}

// PASO 6: Ejemplo de uso del Error Boundary
function App() {
 return (
 <div className="app">
 <h1>Mi Aplicación</h1>

 {/* Envolvemos componentes críticos con ErrorBoundary */}
 <ErrorBoundary>
 <UserProfile userId={1} />
 </ErrorBoundary>

 {/* Otros componentes críticos pueden tener su propio boundary */}
 <ErrorBoundary>
 <ShoppingCart />
 </ErrorBoundary>

 {/* Los errores en este componente no afectarán a los anteriores */}
 <Footer />
 </div>
);
}
}

```

**💡 Nota para juniors:** Los Error Boundaries solo capturan errores en componentes hijos durante el renderizado, métodos del ciclo de vida y constructores. No capturan errores en manejadores de eventos, código asíncrono o funciones de utilidad fuera de React.

## Portales: Renderizado fuera del DOM padre

Los portales permiten renderizar componentes en nodos DOM fuera de la jerarquía del componente padre, perfecto para modales, tooltips y diálogos.

```

// PASO 1: Importamos la API de createPortal de React DOM
import { createPortal } from 'react-dom';
import { useState } from 'react';

// PASO 2: Creamos un componente Modal que usa portal
function Modal({ isOpen, onClose, children }) {
 // Si el modal no está abierto, no renderizamos nada
 if (!isOpen) return null;

 // PASO 3: Usamos createPortal para renderizar el componente
 // fuera de su jerarquía normal, directamente en el body
 return createPortal(
 // Este es el contenido que se renderizará en el portal
 <div className="modal-overlay">
 <div className="modal-content">

```

```

 <button className="close-button" onClick={onClose}>
 ×
 </button>
 {children}
 </div>
 </div>,
 // Segundo argumento: el nodo DOM donde se renderizará
 // En este caso, lo hacemos directamente en el body del documento
 document.body
);
}

// PASO 4: Componente principal que usa el modal
function AppWithModal() {
 const [isModalOpen, setIsModalOpen] = useState(false);

 return (
 <div className="app-container">
 <h1>Aplicación con Portal Modal</h1>
 <button onClick={() => setIsModalOpen(true)}>
 Abrir Modal
 </button>

 {/*
 Aunque el Modal está declarado aquí como hijo,
 se renderizará directamente en el body gracias al portal
 */}
 <Modal
 isOpen={isModalOpen}
 onClose={() => setIsModalOpen(false)}
 >
 <h2>Contenido del Modal</h2>
 <p>Este modal se renderiza fuera de la jerarquía del componente padre.</p>
 <p>Es ideal para modales, tooltips, y cualquier UI que necesite "flotar" sobre
 el contenido.</p>
 </Modal>
 </div>
);
}

```

**💡 Tip para juniors:** Los portales son útiles cuando necesitas que un componente escape el contexto de su parent, especialmente cuando hay problemas con CSS como `overflow: hidden` o `z-index` que afectarían a modales o tooltips.

## Suspense avanzado: Carga de datos elegante

Suspense permite coordinar estados de carga y crear experiencias de usuario fluidas al trabajar con contenido asíncrono.

```

// PASO 1: Importamos los componentes y hooks necesarios
import { Suspense, useState, useTransition } from 'react';

// PASO 2: Componente que carga datos y "suspende" mientras esperamos
function UserData({ userId }) {

```

```
// Este es un recurso que implementa la interfaz de Suspense
// En un caso real, usarías una biblioteca como React Query o SWR
// que tengan soporte para Suspense
const user = fetchUserData(userId);

// Si el recurso aún está cargando, "lanzará" una promesa
// causando que React suspenda este componente

// Cuando los datos estén disponibles, renderizamos normalmente
return (
 <div className="user-card">
 <h3>{user.name}</h3>
 <p>Email: {user.email}</p>
 <p>Rol: {user.role}</p>
 </div>
);
}

// PASO 3: Creamos un componente con pestañas que usa Suspense y useTransition
function UserDashboard() {
 const [selectedUserId, setSelectedUserId] = useState(1);
 const [isPending, startTransition] = useTransition();

 // Lista de usuarios para las pestañas
 const users = [
 { id: 1, name: "Ana" },
 { id: 2, name: "Carlos" },
 { id: 3, name: "Elena" }
];

 // PASO 4: Función para cambiar de usuario con transición
 const selectUser = (userId) => {
 // Envolvemos el cambio de estado en una transición para
 // que la UI siga siendo receptiva
 startTransition(() => {
 setSelectedUserId(userId);
 });
 };

 return (
 <div className="dashboard">
 {/* PASO 5: Pestañas de usuarios */}
 <div className="tabs">
 {users.map(user => (
 <button
 key={user.id}
 onClick={() => selectUser(user.id)}
 className={user.id === selectedUserId ? "active" : ""}
 // Aplicamos estilo si está en transición
 style={{ opacity: isPending ? 0.7 : 1 }}
 >
 {user.name}
 </button>
)));
 </div>
 </div>
);
}
```

```
{/* PASO 6: Indicador de transición pendiente */}
{isPending && <div className="loading-indicator">Cambiando usuario...</div>}

{/* PASO 7: Suspense para el contenido que puede tardar en cargar */}
<Suspense fallback={<div className="loader">Cargando datos de usuario...</div>}>
 <UserData userId={selectedUserId} />
</Suspense>

{/* PASO 8: Contenido adicional que también puede "suspenderse" */}
<Suspense fallback={<div className="loader">Cargando actividad reciente...
</div>}>
 <UserActivity userId={selectedUserId} />
</Suspense>
</div>
);
}
```

💡 **Para juniors:** Suspense combinado con `useTransition` es poderoso para crear UIs fluidas que cargan datos de forma asíncrona. El código puede parecer síncrono y limpio aunque esté manejando operaciones asíncronas complejas.

## Ξ Arquitectura de aplicaciones grandes y cheatsheets

La arquitectura adecuada puede hacer la diferencia entre una aplicación escalable y un proyecto imposible de mantener.

### Modularización y organización de código

```
// File: src/features/auth/components/LoginForm.jsx
import { useState } from 'react';
import { useAuth } from '../hooks/useAuth';
import { Button } from '@/components/ui/Button';
import { TextField } from '@/components/ui/TextField';
import { validateEmail, validatePassword } from '../utils/validators';

// PASO 1: Componente aislado con responsabilidad única
export function LoginForm({ onSuccess, redirectPath = '/dashboard' }) {
 const [formData, setFormData] = useState({ email: '', password: '' });
 const [errors, setErrors] = useState({});
 const { login, isLoading } = useAuth();

 // PASO 2: Manejadores de eventos encapsulados en el componente
 const handleChange = (e) => {
 const { name, value } = e.target;
 setFormData(prev => ({ ...prev, [name]: value }));
 }

 // Validación en tiempo real
 if (name === 'email') {
 setErrors(prev => ({
 ...prev,
 email: validateEmail(value) ? '' : 'Email inválido'
 }));
 }

 if (name === 'password') {
 setErrors(prev => ({
 ...prev,
 password: validatePassword(value) ? '' : 'La contraseña debe tener al menos 8 caracteres'
 }));
 }
};

// PASO 3: Lógica de negocio clara y separada
const handleSubmit = async (e) => {
 e.preventDefault();

 // Validación completa antes de enviar
 const emailValid = validateEmail(formData.email);
 const passwordValid = validatePassword(formData.password);

 if (!emailValid || !passwordValid) {
 setErrors({
 email: emailValid ? '' : 'Email inválido',
 password: passwordValid ? '' : 'La contraseña debe tener al menos 8
```

```
caracteres'
 });
 return;
}

try {
 await login(formData.email, formData.password);
 onSuccess?.(redirectPath);
} catch (error) {
 setErrors({
 form: error.message || 'Error al iniciar sesión'
 });
}
};

return (
 <form onSubmit={handleSubmit} className="login-form">
 <h2>Iniciar sesión</h2>

 {errors.form && (
 <div className="error-message">{errors.form}</div>
)}

 <div className="form-group">
 <TextField
 label="Email"
 type="email"
 name="email"
 value={formData.email}
 onChange={handleChange}
 error={errors.email}
 required
 />
 </div>

 <div className="form-group">
 <TextField
 label="Contraseña"
 type="password"
 name="password"
 value={formData.password}
 onChange={handleChange}
 error={errors.password}
 required
 />
 </div>

 <Button
 type="submit"
 disabled={isLoading}
 variant="primary"
 fullWidth
 >
 {isLoading ? 'Iniciando sesión...' : 'Iniciar sesión'}
 </Button>
 </form>
)
```

```
);
}
```

## Feature-Based Architecture

La arquitectura basada en características organiza el código según las funcionalidades del negocio, no por tipos de archivos.

```
src/
 └── features/ # Módulos organizados por funcionalidad
 ├── auth/ # Todo lo relacionado con autenticación
 ├── api/ # Llamadas a API específicas de auth
 ├── components/ # Componentes UI específicos de auth
 ├── hooks/ # Hooks personalizados para auth
 ├── store/ # Estado relacionado con auth
 ├── utils/ # Utilidades para auth
 └── index.js # Punto de entrada público de la feature

 ├── products/ # Todo lo relacionado con productos
 ├── checkout/ # Todo lo relacionado con el proceso de compra
 └── user-profile/ # Todo lo relacionado con perfil de usuario

 └── components/ # Componentes compartidos y reutilizables
 ├── ui/ # Componentes de UI genéricos (botones, inputs...)
 └── layout/ # Componentes de estructura (header, footer...)

 ├── hooks/ # Hooks compartidos en toda la app
 ├── utils/ # Utilidades globales
 ├── services/ # Servicios globales (API client, analítica...)
 ├── styles/ # Estilos globales
 └── App.jsx # Punto de entrada de la aplicación
```

**💡 Tip de arquitectura:** La modularización basada en features permite que los equipos trabajen en paralelo con menos conflictos y hace más fácil eliminar o añadir funcionalidades completas sin afectar al resto del código.

## Nomenclatura y convenciones coherentes

```
// Buenas prácticas de nomenclatura en React

// PASO 1: Componentes con PascalCase
function UserProfileCard({ user }) {
 // ...
}

// PASO 2: Hooks personalizados con prefijo "use"
function useWindowSize() {
 // ...
}

// PASO 3: HOCs con prefijo "with"
```

```

function withAuth(Component) {
 // ...
}

// PASO 4: Nombrar archivos igual que el componente exportado
// UserProfileCard.jsx exporta UserProfileCard

// PASO 5: Archivos de test con sufijo .test.js o .spec.js
// UserProfileCard.test.jsx

// PASO 6: Context API con sufijo Context
const UserContext = createContext();

// PASO 7: Providers con sufijo Provider
function ThemeProvider({ children }) {
 // ...
}

// PASO 8: Eventos con prefijo handle + verbo en infinitivo
const handleSubmit = (e) => {
 // ...
}

// PASO 9: Props para renderizar funciones con prefijo render
<Table renderRow={(item) => <Row {...item} />} />

// PASO 10: Estado booleano con prefijos is/has/should
const [isLoading, setIsLoading] = useState(false);
const [hasError, setError] = useState(false);

```

## Cheatsheets rápidas

### Cheatsheet de Hooks

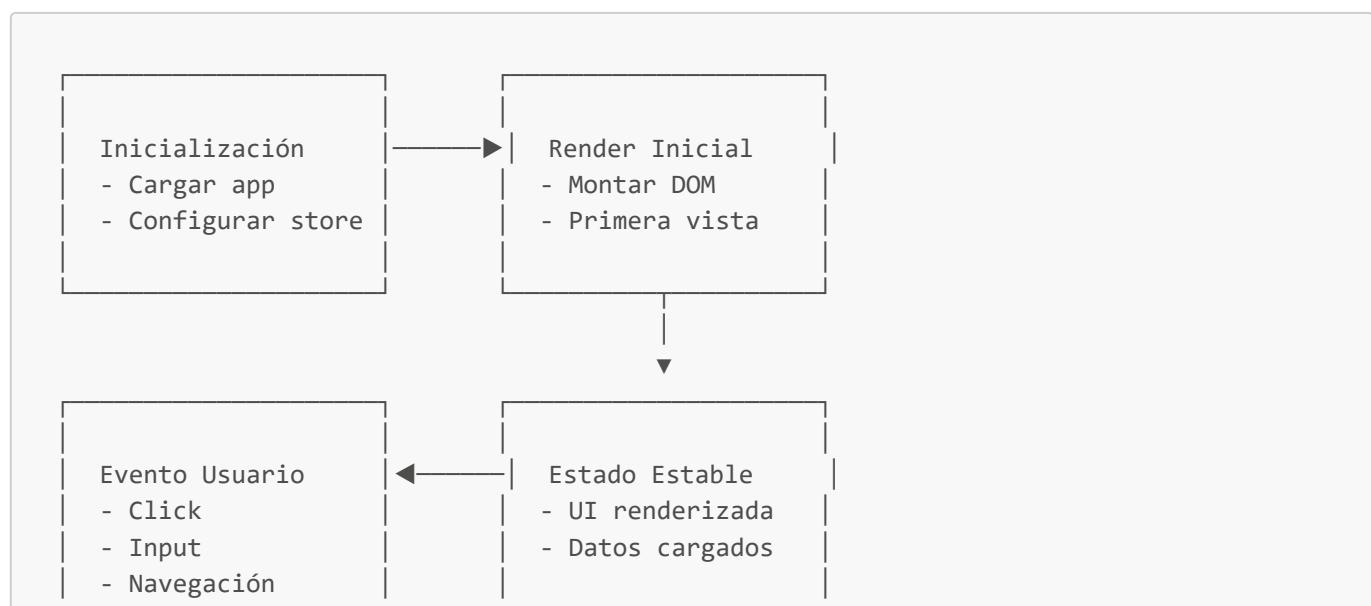
Hook	Uso principal	Alternativas	Ejemplo simple
useState	Estado local	Class component state	<code>const [count, setCount] = useState(0)</code>
useEffect	Efectos secundarios	componentDidMount/Update/Unmount	<code>useEffect(() =&gt; { document.title = count }, [count])</code>
useContext	Acceder al contexto	Context.Consumer	<code>const theme = useContext(ThemeContext)</code>
useReducer	Estado complejo	Redux	<code>const [state, dispatch] = useReducer(reducer, initialState)</code>
useMemo	Memoizar valores	shouldComponentUpdate	<code>const total = useMemo(() =&gt; calcTotal(items), [items])</code>

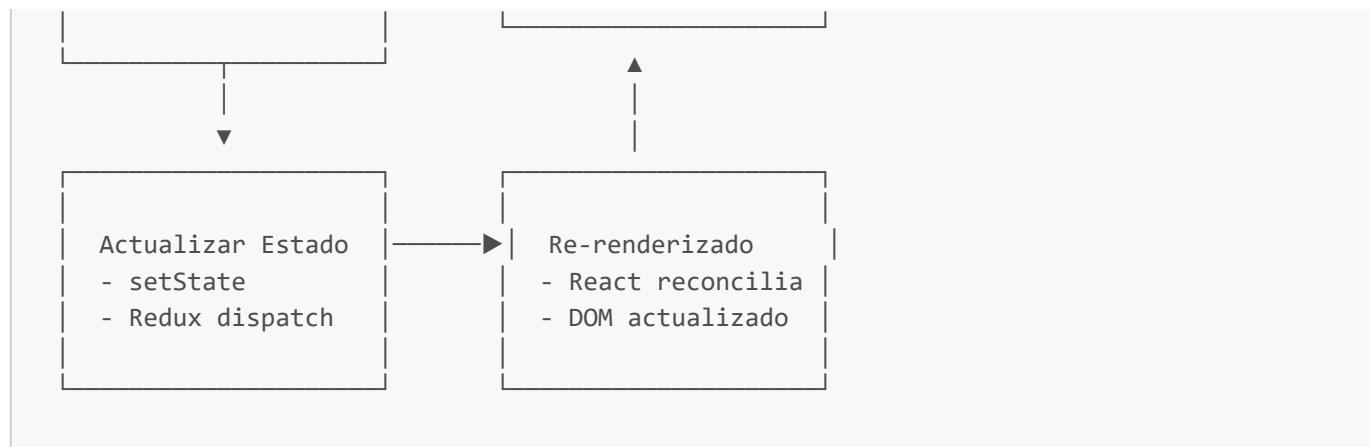
Hook	Uso principal	Alternativas	Ejemplo simple
useCallback	Memoizar funciones	-	<pre>const handleClick = useCallback(() =&gt; setCount(c =&gt; c+1), [])</pre>
useRef	Referencia persistente	createRef	<pre>const inputRef = useRef(null)</pre>
useLayoutEffect	Efectos sincrónicos	-	<pre>useLayoutEffect(() =&gt; { measureHeight() }, [])</pre>
useImperativeHandle	API imperativa	forwardRef	<pre>useImperativeHandle(ref, () =&gt; ({ focus }))</pre>

## 🔗 Cheatsheet de Patrones

Patrón	Uso principal	Ejemplo
Compound Components	Componentes relacionados	<pre>&lt;Menu&gt;&lt;MenuItem/&gt;&lt;MenuItem/&gt;&lt;/Menu&gt;</pre>
Render Props	Compartir lógica	<pre>&lt;List render={item =&gt; &lt;ListItem {...item} /&gt;} /&gt;</pre>
Custom Hooks	Reutilizar lógica	<pre>const { width, height } = useWindowSize()</pre>
HOC	Componentes mejorados	<pre>export default withAuth(Dashboard)</pre>
Provider Pattern	Estado compartido	<pre>&lt;ThemeProvider&gt;&lt;App /&gt;&lt;/ThemeProvider&gt;</pre>
State Reducer	Control externo	<pre>&lt;Counter reducer={(state, action) =&gt; {...}} /&gt;</pre>
Controlled Components	Estado controlado	<pre>&lt;Input value={value} onChange={setValue} /&gt;</pre>

## Diagrama de Flujo de Aplicación React





## 🔒 Optimización avanzada, seguridad y performance en producción

La producción requiere consideraciones especiales para garantizar rendimiento, seguridad y estabilidad de tu aplicación React.

### Lazy loading y code splitting

```
// PASO 1: Importar lazy y Suspense de React
import React, { lazy, Suspense, useState } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

// PASO 2: Importar componentes básicos de forma normal
import Navbar from './components/Navbar';
import HomePage from './pages/HomePage';
import Footer from './components/Footer';

// PASO 3: Importar componentes pesados con lazy loading
// Estos componentes solo se cargarán cuando se necesiten
const Dashboard = lazy(() => import('./pages/Dashboard'));
const UserProfile = lazy(() => import('./pages/UserProfile'));
const ProductCatalog = lazy(() => import('./pages/ProductCatalog'));
const Analytics = lazy(() => import('./pages/Analytics'));

// PASO 4: Crear un componente de carga elegante
function LoadingFallback() {
 return (
 <div className="page-loading">
 <div className="spinner"></div>
 <p>Cargando contenido...</p>
 </div>
);
}

// PASO 5: Implementar la aplicación con lazy loading
function App() {
 return (
 <BrowserRouter>
 <Navbar />

 {/* PASO 6: Envolver rutas que usan lazy loading en Suspense */}
 <Suspense fallback={<LoadingFallback />}>
 <Routes>
 {/* La página de inicio se carga inmediatamente */}
 <Route path="/" element={<HomePage />} />

 {/* Estas rutas cargarán sus componentes bajo demanda */}
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/profile" element={<UserProfile />} />
 <Route path="/products" element={<ProductCatalog />} />
 <Route path="/analytics" element={<Analytics />} />
 </Routes>
 </Suspense>

 <Footer />

```

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

 **Mejora de rendimiento:** El code splitting puede reducir el tamaño del bundle inicial hasta en un 60-70%, lo que resulta en tiempos de carga iniciales mucho más rápidos.

## Estrategias de caché efectivas

```
// PASO 1: Implementar un hook personalizado para caché de datos
import { useState, useEffect } from 'react';

function useCachedData(key, fetchFn, ttl = 300000) { // ttl: 5 minutos por defecto
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 // PASO 2: Comprobar si hay datos en caché y si son válidos
 const cachedItem = localStorage.getItem(`cache_${key}`);

 if (cachedItem) {
 try {
 const { value, timestamp } = JSON.parse(cachedItem);
 const isValid = Date.now() - timestamp < ttl;

 if (isValid) {
 // PASO 3: Usar datos del caché si son válidos
 setData(value);
 setLoading(false);
 return;
 }
 } catch (e) {
 // Si hay error al parsear, ignorar el caché
 console.warn('Error al leer caché:', e);
 localStorage.removeItem(`cache_${key}`);
 }
 }
 });

 // PASO 4: Si no hay caché válido, hacer la petición
 const fetchData = async () => {
 try {
 setLoading(true);
 const result = await fetchFn();

 // PASO 5: Guardar en caché los nuevos datos
 localStorage.setItem(
 `cache_${key}`,
 JSON.stringify({
 value: result,
 timestamp: Date.now()
 })
);
 };
 };
}
```

```
 setData(result);
 setError(null);
 } catch (e) {
 setError(e.message);
 console.error('Error fetching data:', e);
 } finally {
 setLoading(false);
 }
};

fetchData();
}, [key, fetchFn, ttl]);

// PASO 6: Función para refrescar datos manualmente
const reloadData = async () => {
 try {
 setLoading(true);
 const result = await fetchFn();

 localStorage.setItem(
 `cache_${key}`,
 JSON.stringify({
 value: result,
 timestamp: Date.now()
 })
);
 }

 setData(result);
 setError(null);
} catch (e) {
 setError(e.message);
} finally {
 setLoading(false);
}
};

return { data, loading, error, reloadData };
}

// PASO 7: Ejemplo de uso en un componente
function ProductList() {
 const { data, loading, error, reloadData } = useCachedData(
 'products',
 () => fetch('/api/products').then(res => res.json()),
 60000 * 15 // 15 minutos de TTL
);

 if (loading) return <p>Cargando productos...</p>;
 if (error) return <p>Error: {error}</p>;

 return (
 <div>
 <button onClick={reloadData}>Actualizar productos</button>

 {products.map(product => (
 <li key={product.id}>{product.name}
))}

 </div>
);
}
}
```

```
 })}

</div>
);
}
```

## Seguridad en producción

```
// PASO 1: Componente de entrada segura contra XSS
import React, { useState } from 'react';
import DOMPurify from 'dompurify'; // Importamos librería para sanitización

// PASO 2: Función para escapar HTML (protección básica)
function escapeHTML(str) {
 return str
 .replace(/&/g, '&')
 .replace(/</g, '<')
 .replace(/>/g, '>')
 .replace(/"/g, '"')
 .replace(/\'/g, ''');
}

// PASO 3: Hook personalizado para input seguro
function useSecureInput(initialValue = '') {
 const [value, setValueInternal] = useState(initialValue);

 // Sanitizamos el valor antes de guardarlo en el estado
 const setValue = (newValue) => {
 const sanitized = typeof newValue === 'string'
 ? DOMPurify.sanitize(newValue)
 : newValue;
 setValueInternal(sanitized);
 };

 return [value, setValue];
}

// PASO 4: Componente de comentario seguro
function SecureCommentForm() {
 const [comment, setComment] = useSecureInput('');
 const [comments, setComments] = useState([]);

 const handleSubmit = (e) => {
 e.preventDefault();

 // PASO 5: Validar datos antes de procesarlos
 if (!comment.trim()) return;

 // Añadimos el comentario a la lista
 setComments([...comments, comment]);
 setComment('');
 };

 // PASO 6: Nunca usar dangerouslySetInnerHTML sin sanitizar
}
```

```
// O mejor aún, evitarlo completamente
return (
 <div>
 <form onSubmit={handleSubmit}>
 <textarea
 value={comment}
 onChange={(e) => setComment(e.target.value)}
 placeholder="Escribe un comentario"
 />
 <button type="submit">Publicar comentario</button>
 </form>

 <div className="comments-list">
 <h3>Comentarios ({comments.length})</h3>
 {comments.map((text, index) => (
 <div key={index} className="comment">
 {/* Mostramos el texto como text content, no como HTML */}
 <p>{text}</p>
 </div>
))}
 </div>
 </div>
);
}

// PASO 7: Componente con protección CSRF para formularios
function SecureLoginForm() {
 const [formData, setFormData] = useState({
 email: '',
 password: ''
 });

 // Al montar, obtenemos un token CSRF del servidor
 const [csrfToken, setCsrfToken] = useState('');

 useEffect(() => {
 // Obtener token CSRF de una cookie o endpoint específico
 fetch('/api/csrf-token')
 .then(res => res.json())
 .then(data => setCsrfToken(data.token));
 }, []);

 const handleSubmit = async (e) => {
 e.preventDefault();

 // PASO 8: Enviar token CSRF con la petición
 try {
 const response = await fetch('/api/login', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 'X-CSRF-Token': csrfToken
 },
 body: JSON.stringify(formData),
 credentials: 'include' // Importante para cookies de sesión
 });
 }
 };
}
```

```
if (!response.ok) throw new Error('Error de autenticación');
// Procesar respuesta...

} catch (error) {
 console.error('Error:', error);
}

};

return (
 <form onSubmit={handleSubmit}>
 {/* Incluir token CSRF como campo oculto también es una buena práctica */}
 <input type="hidden" name="csrf" value={csrfToken} />
 {/* Resto del formulario... */}
 </form>
);
}
```

 **Consejos de seguridad:** Nunca almacenes información sensible como tokens JWT en localStorage; usa cookies HttpOnly para tokens de sesión y no confíes en datos provenientes del cliente sin validación previa.

## ♿ Accesibilidad avanzada (a11y) y buenas prácticas

La accesibilidad no es una característica adicional, sino un requisito fundamental para construir aplicaciones inclusivas y cumplir con normativas legales.

### Implementación correcta de roles ARIA

```
// PASO 1: Importaciones necesarias
import { useState, useRef, useEffect } from 'react';

// PASO 2: Creamos un componente de menú desplegable accesible
function AccessibleDropdown({ label, options, onSelect }) {
 const [isOpen, setIsOpen] = useState(false);
 const [activeIndex, setActiveIndex] = useState(-1);
 const dropdownRef = useRef(null);
 const menuRef = useRef(null);

 // PASO 3: Manejamos el teclado para navegación accesible
 const handleKeyDown = (e) => {
 switch (e.key) {
 case 'ArrowDown':
 e.preventDefault(); // Evitamos scroll de página
 setActiveIndex(prev =>
 prev < options.length - 1 ? prev + 1 : 0
);
 break;
 case 'ArrowUp':
 e.preventDefault();
 setActiveIndex(prev =>
 prev > 0 ? prev - 1 : options.length - 1
);
 break;
 case 'Enter':
 case 'Space':
 e.preventDefault();
 if (isOpen && activeIndex >= 0) {
 onSelect(options[activeIndex]);
 setIsOpen(false);
 setActiveIndex(-1);
 } else {
 setIsOpen(true);
 }
 break;
 case 'Escape':
 setIsOpen(false);
 setActiveIndex(-1);
 break;
 default:
 break;
 }
 };
}

// PASO 4: Gestión del enfoque con useEffect
useEffect(() => {
```

```
// Cuando el menú se abre, movemos el foco al primer elemento
if (isOpen && menuRef.current) {
 const items = menuRef.current.querySelectorAll('[role="menuitem"]');
 if (items.length && activeIndex >= 0) {
 items[activeIndex].focus();
 }
}
}, [isOpen, activeIndex]);

// PASO 5: Cerrar el menú si se hace clic fuera
useEffect(() => {
 const handleClickOutside = (event) => {
 if (dropdownRef.current && !dropdownRef.current.contains(event.target)) {
 setIsOpen(false);
 }
 };
 document.addEventListener('mousedown', handleClickOutside);
 return () => {
 document.removeEventListener('mousedown', handleClickOutside);
 };
}, []);

return (
 <div
 ref={dropdownRef}
 className="dropdown-container"
 >
 {/* PASO 6: Botón con atributos ARIA correctos */}
 <button
 aria-haspopup="true"
 aria-expanded={isOpen}
 aria-controls="dropdown-menu"
 onClick={() => setIsOpen(!isOpen)}
 onKeyDown={handleKeyDown}
 className="dropdown-toggle"
 >
 {label}
 </button>

 {/* PASO 7: Menú con roles ARIA adecuados */}
 {isOpen && (
 <ul
 id="dropdown-menu"
 ref={menuRef}
 role="menu"
 aria-label={`${label} options`}
 className="dropdown-menu"
 >
 {options.map((option, index) => (
 <li
 key={option.id}
 role="menuitem"
 tabIndex={-1}
 onClick={() => {
 onSelect(option);
 }}
 >
 {option.name}

))}

)}
 </div>
);
```

```

 setIsOpen(false);
)}
onKeyDown={(e) => {
 if (e.key === 'Enter' || e.key === ' ') {
 e.preventDefault();
 onSelect(option);
 setIsOpen(false);
 }
}}
className={index === activeIndex ? 'active' : ''}
>
{option.label}

))}

)}
</div>
);
}

// PASO 8: Uso del componente en la aplicación
function App() {
 const options = [
 { id: 1, label: 'Opción 1' },
 { id: 2, label: 'Opción 2' },
 { id: 3, label: 'Opción 3' }
];

 const handleSelect = (option) => {
 console.log('Opción seleccionada:', option);
 };

 return (
 <div className="app">
 <h1>Menú accesible con ARIA</h1>
 <AccessibleDropdown
 label="Selecciona una opción"
 options={options}
 onSelect={handleSelect}
 />
 </div>
);
}
}

```

## Gestión avanzada del foco

```

// PASO 1: Implementamos un sistema de gestión de foco para modales accesibles
import { useEffect, useRef } from 'react';
import FocusTrap from 'focus-trap-react'; // Recomendado para gestión profesional

function AccessibleModal({ isOpen, onClose, title, children }) {
 const previousFocusRef = useRef(null);

 // PASO 2: Guardamos el elemento que tenía el foco antes de abrir el modal

```

```
useEffect(() => {
 if (isOpen) {
 previousFocusRef.current = document.activeElement;

 // PASO 3: Bloqueamos el scroll del body cuando el modal está abierto
 document.body.style.overflow = 'hidden';
 }

 // PASO 4: Restauramos el foco y el scroll al cerrar
 return () => {
 if (isOpen && previousFocusRef.current) {
 previousFocusRef.current.focus();
 }
 document.body.style.overflow = '';
 };
}, [isOpen]);

// Si no está abierto, no renderizamos nada
if (!isOpen) return null;

return (
 <div
 className="modal-backdrop"
 role="presentation"
 onClick={(e) => {
 // PASO 5: Cerrar al hacer clic en el fondo
 if (e.target === e.currentTarget) onClose();
 }}
 >
 {/* PASO 6: Usamos FocusTrap para mantener el foco dentro del modal */}
 <FocusTrap>
 <div
 role="dialog"
 aria-modal="true"
 aria-labelledby="modal-title"
 className="modal"
 >
 <div className="modal-header">
 <h2 id="modal-title">{title}</h2>
 <button
 onClick={onClose}
 aria-label="Cerrar"
 className="close-button"
 >
 ×
 </button>
 </div>

 <div className="modal-content">
 {children}
 </div>

 <div className="modal-footer">
 <button onClick={onClose}>Cerrar</button>
 </div>
 </div>
 </FocusTrap>
)
```

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

// PASO 7: Formulario accesible con validación y feedback
function AccessibleForm() {
 const [formData, setFormData] = useState({
 name: '',
 email: ''
 });
 const [errors, setErrors] = useState({});
 const nameInputRef = useRef(null);

 // PASO 8: Manejar errores de forma accesible
 const validateForm = () => {
 const newErrors = {};

 if (!formData.name) {
 newErrors.name = 'El nombre es requerido';
 }

 if (!formData.email) {
 newErrors.email = 'El email es requerido';
 } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
 newErrors.email = 'El formato del email es inválido';
 }

 setErrors(newErrors);
 return Object.keys(newErrors).length === 0;
 };

 const handleSubmit = (e) => {
 e.preventDefault();

 if (validateForm()) {
 // Procesar formulario...
 alert('Formulario enviado con éxito');
 } else {
 // PASO 9: Mover el foco al primer campo con error
 const firstErrorField = Object.keys(errors)[0];
 if (firstErrorField === 'name' && nameInputRef.current) {
 nameInputRef.current.focus();
 }
 }
 };
};

return (
 <form onSubmit={handleSubmit} noValidate>
 <div className="form-group">
 <label htmlFor="name">
 Nombre:
 {errors.name && (

 {errors.name}

)}
 </label>
 </div>
 </form>
);
```

```

)}
 </label>
 <input
 ref={nameInputRef}
 id="name"
 type="text"
 value={formData.name}
 onChange={(e) => setFormData({...formData, name: e.target.value})}
 aria-invalid={!errors.name}
 aria-describedby={errors.name ? "name-error" : undefined}
 />
 </div>

 <div className="form-group">
 <label htmlFor="email">
 Email:
 {errors.email && (

 {errors.email}

)}
 </label>
 <input
 id="email"
 type="email"
 value={formData.email}
 onChange={(e) => setFormData({...formData, email: e.target.value})}
 aria-invalid={!errors.email}
 aria-describedby={errors.email ? "email-error" : undefined}
 />
 </div>

 <button type="submit">Enviar</button>
 </form>
);
}

```

**Nota sobre accesibilidad:** Según WebAIM, el 97.8% de las páginas principales tienen errores de accesibilidad detectables. Implementar estas técnicas te coloca automáticamente por encima de la mayoría de sitios web.

## Herramientas de validación y buenas prácticas

```

// Herramientas recomendadas para verificar accesibilidad:

// 1. ESLint con plugin de a11y
// En .eslintrc:
/*
{
 "extends": [
 "react-app",
 "plugin:jsx-a11y/recommended"
],
 "plugins": [

```

```
"jsx-a11y"
]
}
*/
/*

import React from 'react';
import { render } from '@testing-library/react';
import { axe, toHaveNoViolations } from 'jest-axe';

expect.extend(toHaveNoViolations);

test('Button component has no accessibility violations', async () => {
 const { container } = render(<Button>Click me</Button>);
 const results = await axe(container);
 expect(results).toHaveNoViolations();
});
*/
/*

// 3. React Testing Library ayuda a escribir tests accesibles
// Los selectores estimulan el uso de etiquetas y roles accesibles
/*
test('Modal can be closed with escape key', () => {
 render(<Modal isOpen={true} onClose={mockClose} />);

 // Prueba la interacción por teclado
 fireEvent.keyDown(document, { key: 'Escape' });

 expect(mockClose).toHaveBeenCalled();
});
*/
/*

// PASO 1: Componente de lista accesible con buenas prácticas
function AccessibleList({ items }) {
 return (
 <div>
 <h2 id="list-heading">Elementos disponibles</h2>

 {/* PASO 2: Usar elementos semánticos correctos */}
 <ul
 aria-labelledby="list-heading"
 className="items-list"
 >
 {items.map((item) => (
 <li key={item.id}>
 {/* PASO 3: Botones con texto descriptivo */}
 <button
 onClick={() => alert(`Seleccionaste: ${item.name}`)}
 aria-label={`Ver detalles de ${item.name}`}
 >
 {item.name}
 </button>

 {/* PASO 4: Íconos con texto alternativo */}
 <button>
```

```
 onClick={() => alert(`Eliminando: ${item.name}`)}
 aria-label={`Eliminar ${item.name}`}
 className="delete-button"
 >
 ☒
 </button>

))

/* PASO 5: Si la lista está vacía, mostrar mensaje apropiado */
{items.length === 0 && (
 <p>No hay elementos disponibles.</p>
)
</div>
);
}

// PASO 6: Lista de verificación a11y para desarrolladores React
/*
✓ Usar elementos semánticos (button en lugar de div onClick)
✓ Proporcionar alternativas de texto para imágenes e iconos
✓ Asegurar suficiente contraste de color (mínimo 4.5:1)
✓ Implementar navegación completa por teclado
✓ Añadir atributos ARIA solo cuando sea necesario
✓ Manejar estados de foco visibles
✓ Usar etiquetas descriptivas para formularios
✓ Proporcionar feedback para errores de validación
✓ Verificar orden lógico de tabulación
✓ Probar con lectores de pantalla
*/
```

 **Checklist esencial:** Al finalizar cada componente, asegúrate de que: (1) sea navegable completamente por teclado, (2) transmita toda la información visualmente disponible a usuarios de lectores de pantalla, y (3) mantenga un enfoque visual claro durante la navegación.

## Monitoreo y logging en producción

- **Ejemplo de uso de Sentry:** Implementar Sentry para el seguimiento de errores en producción y obtener informes detallados.
  - **Ejemplo de uso de LogRocket:** Utilizar LogRocket para grabar sesiones de usuario y reproducir errores en el contexto de la aplicación.
  - **Ejemplo de estrategias de alertas:** Configurar alertas para notificar al equipo sobre errores críticos y problemas de rendimiento.
  - **Ejemplo de logs de errores:** Implementar un sistema de logging para capturar y almacenar errores en producción.
  - **Ejemplo de métricas:** Utilizar herramientas de monitoreo para recopilar métricas de rendimiento y uso de la aplicación.
- 

## Patrones avanzados de hooks personalizados

- **Ejemplo de uso de useInfiniteScroll:** Implementar un hook para cargar más datos a medida que el usuario se desplaza hacia abajo en la página.
  - **Ejemplo de uso de useFormValidation:** Crear un hook para gestionar la validación de formularios de manera eficiente.
  - **Ejemplo de uso de useAuth:** Implementar un hook para gestionar la autenticación de usuarios en la aplicación.
  - **Ejemplo de uso de useThemeSwitcher:** Crear un hook para permitir a los usuarios cambiar entre temas claros y oscuros.
  - **Ejemplo de cómo construir hooks escalables:** Seguir buenas prácticas para crear hooks que sean fáciles de mantener y escalar.
- 

## Ejemplos de apps completas

- **Ejemplo de mini-proyecto e-commerce:** Crear una tienda en línea con carrito de compras, gestión de productos y pasarela de pago.
  - **Ejemplo de mini-proyecto dashboard:** Desarrollar un panel de control con gráficos, estadísticas y gestión de usuarios.
  - **Ejemplo de mini-proyecto chat:** Implementar una aplicación de chat en tiempo real con autenticación y gestión de mensajes.
  - **Incluir arquitectura, hooks, lazy loading, optimización y testing:** Asegurarse de que cada mini-proyecto siga buenas prácticas de desarrollo, incluyendo una arquitectura clara, el uso adecuado de hooks, la implementación de lazy loading para mejorar el rendimiento, la optimización del código y la escritura de pruebas para garantizar la calidad.
-

## Plantillas y snippets listos para usar

- **Ejemplo de plantilla de componente:** Crear una plantilla básica para componentes React con manejo de errores y optimizaciones.
  - **Ejemplo de plantilla de formulario:** Diseñar una plantilla para formularios con validación y manejo de estados.
  - **Ejemplo de plantilla de página:** Crear una plantilla para páginas con diseño responsivo y optimización de rendimiento.
  - **Ejemplo de plantilla de layout:** Implementar una plantilla de layout con navegación, pie de página y diseño adaptable.
  - **Ejemplo de plantilla de servicio:** Crear una plantilla para servicios con manejo de errores y optimización de rendimiento.
  - **Ejemplo de snippet de fetch:** Proporcionar un snippet para realizar solicitudes HTTP con manejo de errores y estados de carga.
- 

## Errores comunes de novatos y cómo evitarlos

- **Hooks mal usados:** Asegurarse de seguir las reglas de los hooks, como no llamarlos condicionalmente y mantener su uso consistente.
  - **Renders infinitos:** Identificar y solucionar dependencias incorrectas en useEffect o useMemo para evitar bucles de renderizado.
  - **Props undefined:** Validar las props recibidas en los componentes y proporcionar valores predeterminados cuando sea necesario.
  - **Estado no controlado:** Asegurarse de que todos los estados sean controlados y se actualicen correctamente para evitar inconsistencias en la UI.
  - **Efectos secundarios no gestionados:** Limpiar los efectos secundarios en useEffect para evitar fugas de memoria y comportamientos inesperados.
  - **Dependencias faltantes en useEffect:** Incluir todas las dependencias necesarias en el array de dependencias de useEffect para evitar comportamientos inesperados.
  - **Actualizaciones de estado asincrónicas:** Tener cuidado con las actualizaciones de estado que dependen de valores anteriores, utilizando la función de actualización de estado para garantizar que se utilicen los valores más recientes.
  - **Referencias a valores obsoletos:** Evitar referencias a valores que pueden haber cambiado entre renders, utilizando useRef o asegurando que las dependencias de useEffect estén correctamente configuradas.
  - **Manejo inadecuado de errores:** Implementar un manejo de errores adecuado en los componentes y hooks para evitar que errores no controlados afecten la experiencia del usuario.
  - **Falta de pruebas:** Asegurarse de que los componentes y hooks estén bien probados para evitar errores en producción.
- 

## Pensamiento Reactivo y mentalidad de desarrollo

- **Composición de componentes:** Fomentar la creación de componentes pequeños y reutilizables que se puedan combinar para construir interfaces complejas.
- **Inmutabilidad:** Adoptar un enfoque inmutable para el estado y las props, utilizando técnicas como el spread operator y métodos de array inmutables para evitar mutaciones directas.
- **Reactividad:** Aprovechar la reactividad de React para actualizar la UI en respuesta a cambios en el estado y las props, utilizando hooks como useState y useEffect.

- **Visualización de renders:** Utilizar herramientas como React DevTools para visualizar y analizar los renders de los componentes, identificando posibles problemas de rendimiento.
  - **Mentalidad de desarrollo:** Adoptar una mentalidad de desarrollo centrada en la calidad del código, la mantenibilidad y la colaboración en equipo, siguiendo buenas prácticas y patrones de diseño.
  - **Enfoque en la experiencia del usuario:** Priorizar la experiencia del usuario en el desarrollo de componentes y aplicaciones, asegurando que sean intuitivos, accesibles y fáciles de usar.
  - **Iteración y mejora continua:** Fomentar un enfoque de iteración y mejora continua en el desarrollo, aprendiendo de la retroalimentación y los errores para mejorar constantemente la calidad del producto.
  - **Colaboración y comunicación:** Fomentar la colaboración y la comunicación efectiva entre los miembros del equipo, utilizando herramientas y prácticas que faciliten el trabajo en conjunto.
-

## 💡 Preguntas frecuentes (FAQ)

### Hooks y estado

- **¿Cuáles son los errores más comunes al usar hooks?**

Debes seguir las reglas de los hooks: no los llames dentro de bucles, condiciones o funciones anidadas.

Siempre úsalos en el nivel superior de tu componente o en otros hooks personalizados.

- **¿Cómo puedo evitar el problema de "dependencias omitidas" en useEffect?**

Incluye todas las variables externas utilizadas dentro del efecto en el array de dependencias. Si necesitas evitar re-ejecuciones, considera usar useCallback o useMemo para estabilizar las dependencias.

- **¿Por qué mi componente se renderiza infinitamente con useEffect?**

Probablemente estás modificando un estado dentro del useEffect sin una condición de parada o sin las dependencias correctas. Verifica que tus dependencias sean las adecuadas y considera usar condiciones dentro del efecto.

### Peticiones y APIs

- **¿Cuál es la mejor manera de manejar llamadas API en React?**

Usa useEffect para realizar llamadas a APIs, asegurando que las dependencias estén correctamente configuradas. Para casos más complejos, considera usar React Query o SWR que manejan caché, recarga y estados de error.

- **¿Cómo puedo manejar errores en mis peticiones fetch?**

Implementa bloques try-catch con async/await o usa el manejo de errores con promesas (.catch()). Mantén estados separados para datos, carga y errores para reflejar correctamente el estado de la petición en la UI.

### Formularios y validación

- **¿Cuál es la mejor forma de manejar formularios en React?**

Para formularios simples, usa el patrón de componente controlado con useState. Para formularios complejos, considera bibliotecas como Formik o React Hook Form que manejan validación, errores y estado del formulario.

- **¿Cómo puedo validar formularios eficientemente?**

Puedes implementar validación personalizada con useState o useReducer, o usar bibliotecas como Yup o Zod junto con Formik/React Hook Form para esquemas de validación declarativos.

### Testing y Calidad

- **¿Qué tests debo escribir para mis componentes React?**

Escribe tests unitarios para lógica aislada, tests de integración para componentes que interactúan entre sí, y tests end-to-end para flujos completos. Usa Jest y React Testing Library para testing de componentes.

- **¿Cómo puedo mejorar el rendimiento de mi aplicación React?**

Identifica re-renderizados innecesarios con React DevTools, utiliza React.memo, useCallback y useMemo para componentes y funciones costosas, implementa virtualización para listas largas, y considera el code splitting con React.lazy.

### Arquitectura y patrones

- **¿Cuándo debo usar Redux vs Context API?**

Usa Context API para estado global simple en aplicaciones pequeñas o medianas. Considera Redux para aplicaciones grandes con estado complejo, cuando necesites middleware avanzado o cuando la depuración del flujo de datos sea crucial.

- **¿Cómo organizo una aplicación React escalable?**

Implementa una arquitectura por características o módulos en lugar de por tipos de archivo. Separa la lógica de UI de la lógica de negocio mediante custom hooks, y considera patrones como presentational/container o clean architecture.

## Despliegue y DevOps

- **¿Cuáles son las mejores prácticas para desplegar aplicaciones React?**

Utiliza un CDN para servir assets estáticos, implementa CI/CD para automatizar pruebas y despliegues, configura monitoring y error tracking, y considera soluciones JAMstack como Netlify o Vercel para sitios estáticos.

- **¿Cómo configuro correctamente mi aplicación para diferentes entornos?**

Usa variables de entorno con create-react-app o archivos .env para configuraciones específicas por entorno. Considera herramientas como dotenv para manejar estas variables y abstrae la configuración en un módulo separado.

## Accesibilidad e Internacionalización

- **¿Cómo hago que mi aplicación React sea accesible?**

Usa elementos semánticos de HTML, asegura que todos los elementos interactivos sean accesibles por teclado, implementa ARIA roles y atributos cuando sea necesario, y haz testing con lectores de pantalla.

- **¿Cómo implemento múltiples idiomas en mi aplicación?**

Usa bibliotecas como react-i18next o react-intl para internacionalización. Separa los textos en archivos de traducción, implementa detección de idioma, y considera aspectos culturales como formatos de fecha, moneda y dirección del texto.

---

## Documentación y JSDoc/TSDoc

### Mejores prácticas de documentación

- **¿Cómo debo documentar mis componentes React?**

Documenta cada componente con JSDoc o TSDoc incluyendo: propósito, props con sus tipos y descripciones, ejemplos de uso, y comportamientos especiales. Considera crear un Storybook para visualizar componentes en diferentes estados.

- **¿Cómo puedo documentar mis hooks personalizados?**

Documenta cada hook detallando: propósito, parámetros, valor de retorno, ejemplos de uso, y posibles efectos secundarios. Explica claramente las dependencias internas y cualquier regla especial para su uso correcto.

- **¿Cuál es la mejor manera de documentar funciones y utilidades?**

Usa JSDoc/TSDoc para cada función documentando: propósito, parámetros, valor de retorno, excepciones potenciales y ejemplos de uso. Agrupa utilidades relacionadas en secciones lógicas de la documentación.

- **¿Cómo debería documentar mis pruebas?**

Organiza pruebas con descripciones claras usando "describe" e "it" que expliquen el comportamiento esperado. Incluye comentarios para casos complejos y documentación sobre mocks o configuraciones especiales.

- **¿Qué debe incluir la documentación de arquitectura?**

Crea diagramas que muestren el flujo de datos y la relación entre componentes. Documenta decisiones arquitectónicas en archivos ADR (Architecture Decision Records) y mantén un README que explique la estructura general del proyecto.

- **¿Cómo documento la configuración del entorno?**

Incluye un archivo README.md con instrucciones paso a paso para configurar entornos de desarrollo, prueba y producción. Documenta variables de entorno necesarias, comandos de inicio, y cualquier requisito especial como Docker.

- **¿Cuál es la mejor práctica para documentar una API?**

Usa estándares como OpenAPI/Swagger para documentar endpoints. Incluye métodos HTTP, parámetros requeridos y opcionales, formatos de respuesta, códigos de estado, y ejemplos completos de solicitudes y respuestas.

---

## 🔗 Conclusión

Esta guía es **la referencia definitiva de React moderno**, útil para novatos, juniors y expertos, abarcando **todas las necesidades de aprendizaje, desarrollo y producción**.

A lo largo de este documento, has aprendido:

- **Fundamentos sólidos** de React y sus componentes básicos
- **Hooks modernos** y cómo aprovechar todo su potencial
- **Gestión de estado** desde soluciones simples hasta complejas
- **Routing y navegación** para construir SPAs profesionales
- **Patrones avanzados** que usan los desarrolladores senior
- **Optimización de rendimiento** para aplicaciones fluidas y rápidas
- **Técnicas de seguridad** para proteger tus aplicaciones
- **Prácticas de accesibilidad** para crear productos inclusivos

La diferencia entre un desarrollador junior y uno senior no está solo en conocer la API de React, sino en dominar estos conceptos avanzados y, sobre todo, en saber **cuándo y cómo aplicarlos correctamente**.

¿Qué hacer ahora?

1. **Practica constantemente** - Crea pequeños proyectos aplicando estos conceptos
2. **Analiza código de calidad** - Estudia proyectos open source bien estructurados
3. **Refactoriza código antiguo** - Aplica estos patrones a código existente
4. **Comparte tu conocimiento** - Enseñar es la mejor manera de consolidar lo aprendido

Recuerda que el camino del desarrollador React es un aprendizaje continuo. Las APIs evolucionan, surgen nuevas bibliotecas, pero los principios fundamentales permanecen.

¡Ahora estás equipado para construir aplicaciones React robustas, mantenibles y de alta calidad!

**"El buen código no solo funciona, también es claro, mantenable y sigue los principios de React."**