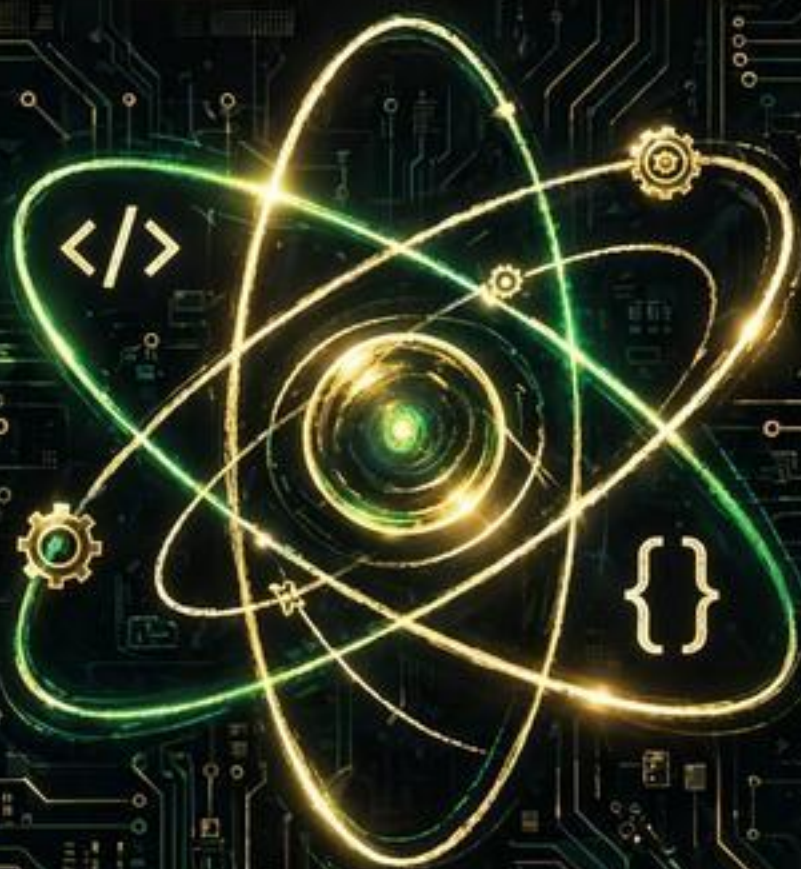


LUCAS EGA ACADEMY

React



Modern Frontend Course • Hooks • State • Performance

Curso Completo de React

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México

lucasegaacademy.com

ÍNDICE

PRÓLOGO.....	4
UNIDAD 1: Bienvenido a React.....	7
UNIDAD 2: Componentes, JSX y Props.....	18
UNIDAD 3: Listas, condicionales y estado con useState.....	56
UNIDAD 4: Eventos e interactividad.....	96
UNIDAD 5: Formularios y efectos secundarios.....	133
UNIDAD 6: Composición de componentes.....	176
UNIDAD 7: Children, estilos y referencias.....	222
UNIDAD 8: Contexto, hooks personalizados y rutas.....	274
UNIDAD 9: Rutas dinámicas, validación y rendimiento.....	331
UNIDAD 10: Proyectos reales y graduación.....	372
EXAMEN FINAL.....	418

PRÓLOGO

Bienvenido a "Curso Completo de React", la guía definitiva de Lucas EGA Academy para aprender React desde cero hasta construir aplicaciones reales de nivel profesional. Este libro nació del curso en video de 30 clases de la academia, pero aquí encontrarás todo explicado con mucho más detalle: conceptos a fondo, analogías, errores comunes, buenas prácticas, ejemplos paso a paso, prácticas con soluciones explicadas y un examen final para medir lo que aprendiste.

React es la librería de JavaScript más usada del mundo para crear interfaces web. La creó Facebook (hoy Meta) y la usan Instagram, Netflix, WhatsApp Web, Airbnb y miles de empresas más. En el mercado laboral, "React" es una de las palabras más pedidas en las ofertas de empleo de desarrollo web, tanto para trabajo remoto como presencial. Aprender React no es solo aprender una herramienta: es abrir la puerta a una de las carreras mejor pagadas del mundo de la tecnología.

Este libro está organizado en 10 unidades, cada una agrupa 3 de las 30 clases originales del curso. Cada unidad sigue la misma estructura para que siempre sepas qué esperar:

- Introducción: qué verás en la unidad y por qué importa.
- Explicación detallada: los conceptos explicados a fondo, con analogías de la vida real, errores comunes que cometen los principiantes y buenas prácticas usadas en la industria.
- Ejemplos paso a paso: código completo que puedes copiar y probar en tu computadora, explicado línea por línea.
- Práctica: ejercicios con soluciones en cada unidad, cada una explicada paso a paso. La programación se aprende programando, no solo leyendo.
- Resumen: los puntos clave para repasar rápido antes de seguir.

Al final del libro hay un EXAMEN FINAL con 50 preguntas: opción múltiple, verdadero o falso, y preguntas abiertas con código. Cada respuesta viene explicada en la sección de soluciones. Si puedes resolver el examen con confianza, estás listo para construir proyectos reales con React.

Cómo estudiar con este libro

No leas este libro como una novela. La programación se aprende con las manos en el teclado. Mi recomendación es la siguiente:

1. Lee la explicación detallada de cada unidad con calma. Subraya (mentalmente o en papel) los conceptos que no entiendas a la primera.
2. Escribe tú mismo cada ejemplo de código en tu computadora. No copies y pegues: escribir el código a mano, aunque sea más lento, hace que tu cerebro lo asimile de verdad.
3. Rompe el código a propósito. Cambia una línea, quita otra, y observa qué pasa. Esa experimentación es donde ocurre el verdadero aprendizaje.
4. Resuelve los ejercicios de práctica ANTES de mirar las soluciones. Si te atorras más de 30 minutos, mira la solución, entiéndela y luego ciérrala y resuelve el ejercicio de nuevo sin mirar.
5. Al terminar cada unidad, lee el resumen y pregúntate: "¿puedo explicar esto con mis palabras?" Si la respuesta es sí, avanza. Si es no, repasa.

Necesitas muy poco para empezar: una computadora (Windows, Mac o Linux), un editor de código como Visual Studio Code (gratis) y Node.js instalado (también gratis). En la Unidad 1 verás la instalación paso a paso. No necesitas saber mucho de JavaScript avanzado, pero sí conviene que conozcas lo básico: variables, funciones, arreglos, objetos y funciones flecha. Si vienes de los cursos de JavaScript de Lucas EGA Academy, tienes todo lo necesario.

Un consejo honesto: React parece fácil al principio y luego se pone retador. Eso es normal. Los hooks, el flujo de datos y el manejo del estado confunden a todos al inicio, incluso a programadores con experiencia. No te desanimes si algo no "hace clic" de inmediato: sigue construyendo cosas pequeñas y un día todo encajará. La diferencia entre quien aprende React y quien no, no es el talento: es la constancia.

Qué vas a construir

Este no es un libro solo de teoría. A lo largo de las unidades construirás: tu primer componente, un contador interactivo, formularios con validación, una app que consume datos de internet, una lista de tareas completa (agregar, tachar, borrar), y en la Unidad 10, un proyecto de nivel portafolio: una tienda en línea con catálogo, detalle de producto y carrito de compras, con navegación entre páginas. Al terminar, tendrás proyectos reales que mostrar en tu portafolio y en entrevistas de trabajo.

Sobre Lucas EGA Academy

Este libro es parte del ecosistema de Lucas EGA Academy (lucasegaacademy.com), donde encontrarás los 22 cursos gratuitos de tecnología, los videos de cada clase y más libros como este. Todo el contenido de la academia es gratis porque creemos que la educación en tecnología no debería tener barreras de dinero. Si este libro te sirve, compártelo: la mejor forma de agradecer es ayudar a que otra persona también aprenda.

Sin más, empecemos. En la Unidad 1 conocerás qué es React, por qué domina el desarrollo web moderno y dejarás tu primer proyecto corriendo en tu computadora. ¡A programar!

Esteban *Lucas EGA Academy - lucasegaacademy.com*

UNIDAD 1: Bienvenido a React

En esta primera unidad conocerás React desde sus cimientos: qué es exactamente, por qué se volvió la librería más usada del mundo para interfaces web, cómo funciona por dentro (sin matemáticas complicadas) y dejarás instalado tu primer proyecto funcionando en tu computadora. Al terminar esta unidad entenderás la filosofía de React y tendrás tu entorno de desarrollo listo para construir.

Las tres clases originales de esta unidad son: Clase 1 (Qué es React), Clase 2 (Instalar React) y Clase 3 (Estructura del proyecto).

Explicación detallada

Qué es React, en palabras simples

React es una librería de JavaScript para construir interfaces de usuario, es decir, todo lo que el usuario ve y con lo que interactúa en una página web o aplicación: botones, formularios, menús, tarjetas de productos, comentarios, notificaciones. La palabra clave aquí es "librería" y no "framework". ¿Cuál es la diferencia? Un framework es como un molde completo: te dice exactamente cómo organizar todo tu proyecto, qué carpetas crear, cómo nombrar las cosas. Una librería, en cambio, es como una caja de herramientas: te da piezas poderosas y tú decides cómo usarlas y cómo organizar tu proyecto. React se enfoca en una sola cosa y la hace muy bien: la interfaz de usuario.

React fue creado en 2013 por Jordan Walke, un ingeniero de Facebook. Facebook tenía un problema enorme: su código de interfaz se había vuelto un caos imposible de mantener. Cada vez que agregaban una función nueva, algo más se rompía. Necesitaban una forma de construir interfaces con piezas pequeñas, predecibles y reutilizables. Así nació React, y en 2015 Facebook lo liberó como código abierto. Hoy lo mantienen Meta y una comunidad gigantesca de millones de desarrolladores en todo el mundo.

La idea central de React es el COMPONENTE. Un componente es una pieza independiente y reutilizable de la interfaz: un botón, una tarjeta de producto, un encabezado, un formulario de login. En vez de escribir una página web como un documento gigante de HTML, la divides en componentes pequeños, cada uno con su propia lógica y apariencia. Luego combinas esos componentes como piezas de LEGO para armar páginas completas. Si necesitas el mismo botón en 20 lugares, lo escribes una vez y lo reutilizas 20 veces. Si hay un error en el botón, lo arreglas en un solo lugar y se corrige en los 20.

La analogía de la fábrica

Imagina que quieres fabricar 10,000 sillas. La forma antigua (HTML tradicional con JavaScript suelto) sería como tener un artesano que construye cada silla desde cero, una por una, a mano. Funciona para pocas sillas, pero es lento y cada silla sale un poco diferente. React es como tener una fábrica con moldes: diseñas el molde del "componente silla" una vez, y la fábrica produce miles de sillas idénticas en segundos. Si quieres cambiar el diseño, cambias el molde y todas las sillas nuevas salen con el cambio. Los componentes son esos moldes: los defines una vez y React los "fabrica" (los dibuja en pantalla) cuantas veces necesites, siempre consistentes.

El DOM virtual: el truco de magia de React

Para entender por qué React es rápido, necesitas saber qué es el DOM. El DOM (Document Object Model) es la representación en memoria que el navegador hace de tu página web: un árbol de elementos (etiquetas HTML) que el navegador dibuja en pantalla. Cada vez que cambias algo en la página con JavaScript tradicional, tocas el DOM directamente, y eso es costoso: el navegador tiene que recalcular estilos, redibujar partes de la pantalla y reorganizar el diseño. Hacer esto cientos de veces por segundo (por ejemplo, al escribir en un buscador con sugerencias en vivo) vuelve lenta la página.

React inventó una solución elegante: el DOM VIRTUAL. Es una copia ligera del DOM real que vive en la memoria de JavaScript (no en el navegador). Cuando algo cambia en tu aplicación (el usuario hace clic, llegan datos nuevos), React primero actualiza su DOM virtual, que es rapidísimo porque es solo un objeto de JavaScript. Luego compara el DOM virtual nuevo con el anterior, detecta EXACTAMENTE qué cambió (un proceso llamado "reconciliación" o "diffing") y solo actualiza esas partes específicas en el DOM real. Es como tener un asistente que, en vez de reescribir todo un documento porque cambiaste una palabra, solo corrige esa palabra.

Esta es la razón por la que aplicaciones enormes como Facebook e Instagram se sienten rápidas: React minimiza el trabajo costoso del navegador. Tú como desarrollador no tienes que pensar en esto: simplemente describes cómo debe verse la interfaz según los datos, y React se encarga de actualizar solo lo necesario. A esto se le llama "programación declarativa": tú declaras QUÉ quieres ver, no CÓMO actualizarlo paso a paso.

Declarativo vs imperativo: un ejemplo de la vida real

Imagina que quieres un café. La forma imperativa sería darle instrucciones paso a paso al barista: "toma una taza, ponla bajo la máquina, presiona el botón, espera 20 segundos, agrega leche, revuelve...". La forma declarativa sería simplemente decir: "quiero un café con leche". Tú declaras el resultado que quieres; el barista (React) se encarga de los pasos. Con JavaScript tradicional (imperativo) tú manipulas el DOM paso a paso: "busca este elemento, cambia su texto, agrega esta clase, oculta este otro". Con React (declarativo) describes: "si el usuario está logueado, muestra su nombre; si no, muestra el botón de login". React se encarga de hacer los cambios necesarios. Este cambio de mentalidad es lo más importante que aprenderás en este libro.

Por qué React domina el mercado

Hay varias razones por las que React se convirtió en el estándar de la industria:

1. Ecosistema gigante: hay librerías React para todo: gráficos, mapas, animaciones, formularios, calendarios. Si necesitas algo, casi seguro ya existe y es gratis.
2. Oportunidades de trabajo: miles de empresas buscan desarrolladores React. Es la habilidad frontend más pedida en ofertas de empleo.
3. React Native: con los mismos conceptos de React puedes crear apps móviles para iOS y Android. Aprendes una vez, aplicas dos veces.
4. Comunidad: si te atorras con algo, hay millones de tutoriales, foros y respuestas en internet. Nunca programarás solo.
5. Respaldo de Meta: no es un proyecto abandonado de un desconocido; lo mantiene una de las empresas de tecnología más grandes del mundo.

Errores comunes de principiantes en esta etapa

Error 1: Pensar que React es un lenguaje nuevo. No lo es: React es JavaScript. Todo lo que sabes de JavaScript (variables, funciones, arreglos, objetos, map, filter) se usa en React todos los días. Si tu JavaScript está flojo, React se sentirá difícil; si tu JavaScript está sólido, React se sentirá natural. Si necesitas repasar, los cursos de JavaScript de Lucas EGA Academy son el complemento perfecto.

Error 2: Querer aprender todo el ecosistema el primer día. React tiene a su alrededor un universo de herramientas: Redux, Next.js, TypeScript, testing, etc. No las necesitas ahora. Este libro te lleva paso a paso: primero los fundamentos, luego los proyectos. El ecosistema avanzado vendrá después, cuando tengas bases sólidas.

Error 3: No instalar las herramientas y solo "leer" el código. React se aprende haciendo. Si llegas al final de esta unidad sin haber instalado Node.js y creado tu primer proyecto, detente y hazlo. Los siguientes capítulos asumen que tienes tu entorno listo.

Buenas prácticas desde el día uno

- Usa Visual Studio Code como editor: es gratis, es el estándar de la industria y tiene extensiones excelentes para React.
- Instala la extensión "ES7+ React/Redux/React-Native snippets" en VS Code: te da atajos para escribir componentes rápido.
- Usa la terminal sin miedo: crear proyectos, instalar paquetes y correr el servidor se hace con comandos simples que verás en esta unidad.
- Acostúmbrate a leer los mensajes de error: React tiene mensajes de error muy descriptivos. El 90% de las veces, el error te dice exactamente qué está mal y dónde.
- Guarda tu código con frecuencia y usa Git para versionarlo. No es tema de este libro, pero es un hábito profesional que te ahorrará dolores de cabeza.

Instalar React: qué está pasando realmente

En la Clase 2 instalas React con un comando que usa Vite. Vamos a desmenuzar qué significa cada parte, porque entender tus herramientas te hace mejor desarrollador.

Node.js es un programa que permite ejecutar JavaScript fuera del navegador, en tu computadora. Lo necesitas porque las herramientas de desarrollo de React (como Vite) están escritas en JavaScript y corren sobre Node.js. Además, Node.js incluye npm (Node Package Manager), que es como una tienda de aplicaciones pero para código: con un comando descargas librerías escritas por otros desarrolladores (como React) y las agregas a tu proyecto.

Vite (se pronuncia "vit", como en francés) es una herramienta de construcción moderna para proyectos web. Antes se usaba "Create React App", pero era lento; Vite es muchísimo más rápido al iniciar el servidor de desarrollo y al recargar tus cambios. Cuando ejecutas el comando de creación, Vite descarga una plantilla con todo lo necesario: React, la configuración, un servidor de desarrollo y una estructura de carpetas lista para trabajar.

El comando completo es:

```
npm create vite@latest mi-app -- --template react
```

Vamos por partes: "npm create" ejecuta un paquete creador; "vite@latest" significa "la versión más reciente de Vite"; "mi-app" es el nombre de la carpeta de tu proyecto (puedes cambiarlo); "-- --template react" le dice a Vite que use la plantilla de React (también hay plantillas para Vue, Svelte, etc.). Después entras a la carpeta (cd mi-app), instalas las dependencias (npm install: descarga React y las librerías que la plantilla necesita) y arrancas el servidor (npm run dev). El servidor te da una dirección local, normalmente <http://localhost:5173>, donde ves tu app funcionando en el navegador. Cada vez que guardas un cambio en el código, la página se recarga sola al instante. Eso se llama "hot reload" y hace el desarrollo rapidísimo y divertido.

La estructura del proyecto, archivo por archivo

Cuando Vite crea tu proyecto, genera varios archivos y carpetas. Al principio parecen muchos, pero solo necesitas entender unos cuantos. Vamos a recorrerlos:

- node_modules/: aquí viven todas las librerías descargadas por npm (React y sus dependencias). Puede tener miles de archivos y pesar cientos de megas. NUNCA edites nada aquí y nunca lo subas a Git (por eso existe el archivo .gitignore). Si algo se rompe feo, puedes borrar esta carpeta y correr "npm install" de nuevo para regenerarla.
- public/: archivos estáticos que se copian tal cual al proyecto final, como el favicon (el iconito de la pestaña del navegador) o imágenes que no necesitan procesamiento.
- src/: AQUÍ VIVIRÁ TU CÓDIGO. Es la carpeta más importante. Todo lo que programes estará aquí dentro.
- main.jsx: el punto de entrada. Es el primer archivo que se ejecuta. Su trabajo es tomar tu componente principal (App) y "montarlo" en el HTML, es decir, decirle a React: "dibuja la aplicación dentro de este elemento de la página".
- App.jsx: tu componente raíz. Toda tu aplicación cuelga de aquí, como las ramas de un árbol cuelgan del tronco. Al inicio contiene un ejemplo que luego reemplazarás con tu propio código.
- index.css y App.css: estilos de ejemplo. Puedes modificarlos o borrarlos cuando uses tus propios estilos.
- index.html: la página HTML base. Contiene un div con id="root" (la "raíz" donde React montará tu app) y carga el archivo main.jsx. Normalmente no necesitas tocarlo, salvo para cambiar el título de la pestaña del navegador.
- package.json: la "identificación" de tu proyecto. Lista el nombre, la versión, los comandos disponibles (como "dev", "build") y las dependencias instaladas. Cuando instales una librería nueva con npm, aparecerá aquí.

- vite.config.js: configuración de Vite. Al inicio no necesitas tocarlo.

La regla de oro: tu código vive en src/. Empieza siempre por App.jsx y desde ahí crea más archivos de componentes conforme tu app crezca.

Preguntas frecuentes de esta unidad

¿React es lo mismo que React Native? No exactamente. React es para interfaces web (navegador). React Native usa los mismos conceptos (componentes, props, estado, hooks) pero en vez de dibujar HTML, dibuja componentes nativos de iOS y Android para crear apps móviles. Si dominas React, aprender React Native es mucho más fácil.

¿Necesito saber HTML y CSS antes de React? Sí, lo básico. React usa JSX que parece HTML, y los estilos se hacen con CSS. No necesitas ser experto, pero debes entender etiquetas, clases y selectores. Los cursos de HTML5 y CSS3 de la academia te preparan perfecto.

¿Puedo usar React sin Vite, solo con un archivo HTML? Sí, se puede cargar React desde un CDN con etiquetas script, y para experimentos rápidos funciona. Pero para proyectos reales, usar Vite (o herramientas similares) es el estándar porque te da un servidor de desarrollo rápido, recarga automática y la capacidad de organizar tu código en múltiples archivos.

¿JavaScript o TypeScript con React? React funciona con JavaScript puro, y así lo aprenderás en este libro. TypeScript (JavaScript con tipos) es muy usado en empresas grandes, pero es un paso posterior. Primero domina React con JavaScript; TypeScript será mucho más fácil después. La academia también tiene curso de TypeScript.

¿Debo aprender primero JavaScript "puro" (vanilla)? Idealmente sí, al menos lo fundamental: variables, funciones, arreglos (map, filter), objetos, desestructuración y funciones flecha. React usa todo eso intensivamente. Si algo de JavaScript no te suena en los ejemplos, repásalo en el curso de JavaScript de la academia.

Ejemplos paso a paso

Ejemplo 1: Instalar Node.js y crear tu primer proyecto

Paso 1: Ve a nodejs.org y descarga la versión LTS (Long Term Support, la estable recomendada) para tu sistema operativo. Instálala aceptando las opciones por defecto.

Paso 2: Abre una terminal y verifica la instalación con:

```
node --version  
npm --version
```

Deberías ver números de versión como v20.x.x y 10.x.x. Si ves los números, Node.js y npm están listos.

Paso 3: En la terminal, ve a la carpeta donde guardas tus proyectos (por ejemplo, Documentos/proyectos) y ejecuta:

```
npm create vite@latest mi-primer-app -- --template react
```

Vite te hará un par de preguntas (puedes aceptar lo sugerido con Enter). Al terminar, verás instrucciones en pantalla.

Paso 4: Entra a la carpeta e instala las dependencias:

```
cd mi-primer-app  
npm install
```

Verás una barra de progreso mientras npm descarga React y las herramientas. Tarda uno o dos minutos la primera vez.

Paso 5: Arranca el servidor de desarrollo:

```
npm run dev
```

La terminal te mostrará algo como "Local: http://localhost:5173/". Abre esa dirección en tu navegador: verás la página de bienvenida de Vite + React. ¡Tu primer proyecto React está corriendo!

Paso 6: Abre la carpeta del proyecto en Visual Studio Code (Archivo > Abrir carpeta). Explora src/App.jsx: ese es el componente que ves en pantalla. Cambia el texto de un encabezado, guarda (Ctrl+S) y mira cómo el navegador se actualiza solo. Acabas de hacer tu primera modificación en React.

Ejemplo 2: Explorando main.jsx

Abre src/main.jsx en tu editor. Verás algo parecido a esto:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

Línea por línea: las tres primeras líneas importan piezas necesarias (StrictMode es un ayudante que te avisa de malas prácticas durante el desarrollo; createRoot es la función que conecta React con el HTML; App es tu componente principal). La última parte busca en el HTML el elemento con id="root" (está en index.html), crea la "raíz" de React ahí y dibuja tu componente App dentro. StrictMode envuelve a App para ayudarte a detectar problemas; no afecta tu app final. Normalmente NO necesitas modificar este archivo: es el puente entre React y la página, y ya funciona.

Ejemplo 3: Tu primer vistazo a App.jsx

Abre src/App.jsx. La plantilla trae un ejemplo con un contador y logos. No te preocupes por entenderlo todo ahora: en la Unidad 2 construirás tus propios componentes desde cero. Por ahora, identifica estas partes:

```
function App() {
  return (
    <>
      <h1>Hola React</h1>
    </>
  )
}

export default App
```

Esto es un componente: una función de JavaScript llamada App que devuelve JSX (ese HTML dentro del return). La última línea (export default App) permite que main.jsx lo importe. Los símbolos <> y </> son un "fragmento": un envoltorio invisible que agrupa elementos sin agregar etiquetas extra al HTML. Todo componente debe devolver UN solo elemento raíz (o un fragmento que agrupe varios).

Prueba esto: reemplaza todo el contenido de App.jsx con el código de arriba, guarda, y verás "Hola React" en tu navegador. Acabas de escribir tu primer componente React funcional.

Práctica

Ejercicio 1: Instalación completa del entorno

Instala Node.js LTS, Visual Studio Code y crea un proyecto con Vite llamado "mi-portafolio". Arranca el servidor y abre la app en el navegador.

SOLUCIÓN: Descarga Node.js LTS desde nodejs.org e instálalo. Verifica con "node --versión" y "npm --versión" en la terminal. Descarga VS Code desde code.visualstudio.com. En la terminal ejecuta "npm create vite@latest mi-portafolio -- --template react", luego "cd mi-portafolio", "npm install" y "npm run dev". Abre la URL que muestra la terminal (normalmente <http://localhost:5173>). Si ves la página de bienvenida, todo quedó instalado correctamente.

Ejercicio 2: Personaliza el título de la pestaña

Cambia el título que aparece en la pestaña del navegador de tu proyecto (ahora dice algo como "mi-primera-app") por "Mi Portafolio React".

SOLUCIÓN: Abre el archivo index.html en la raíz del proyecto. Busca la etiqueta <title> y cambia su contenido: <title>Mi Portafolio React</title>. Guarda y recarga el navegador: la pestaña ahora muestra tu título personalizado. Esto funciona porque index.html es la página base donde React monta tu aplicación.

Ejercicio 3: Limpia App.jsx y escribe tu presentación

Reemplaza todo el contenido de src/App.jsx con un componente que muestre tu nombre en un h1, una breve descripción en un p, y una lista de 3 tecnologías que quieres aprender.

SOLUCIÓN:

```
function App() {
  return (
    <>
      <h1>Hola, soy [tu nombre]</h1>
      <p>Estoy aprendiendo React en Lucas EGA Academy.</p>
      <ul>
        <li>React</li>
        <li>JavaScript</li>
        <li>CSS</li>
      </ul>
    </>
  )
}

export default App
```

Guarda y verás tu presentación en el navegador. Observa el fragmento `<> </>` que agrupa los tres elementos sin agregar etiquetas extra.

Ejercicio 4: Explora package.json

Abre package.json y responde: ¿cuáles son los 3 scripts disponibles y qué hace cada uno? ¿Qué versión de React está instalada?

SOLUCIÓN: Los scripts típicos son: "dev" (arranca el servidor de desarrollo con npm run dev), "build" (genera la versión optimizada para producción con npm run build, crea la carpeta dist/) y "preview" (sirve localmente la versión construida para probarla). La versión de React aparece en "dependencies" como "react": "^18.x.x" o "^19.x.x" según la fecha. El símbolo ^ significa que npm puede instalar actualizaciones menores automáticamente.

Ejercicio 5: Investiga el DOM virtual con un experimento

En tu App.jsx, agrega un párrafo con la hora actual usando `new Date().toLocaleTimeString()`. Guarda, espera 10 segundos y recarga la página. ¿Cambió la hora sin recargar? ¿Por qué?

SOLUCIÓN: La hora solo cambia al recargar la página manualmente. Esto es porque React dibujó el componente una sola vez con el valor que tenía `new Date()` en ese momento. React NO vuelve a ejecutar tu componente "porque pasó el tiempo": solo lo vuelve a dibujar cuando cambia el estado o las props (lo verás en la Unidad 3 con `useState`). Este experimento te enseña una lección clave: React re-renderiza por cambios de datos, no por el paso del tiempo. Código de prueba:

```
function App() {
  return (
    <>
      <h1>Hora actual</h1>
      <p>{new Date().toLocaleTimeString()}</p>
    </>
  )
}

export default App
```

Resumen de la unidad

- React es una librería de JavaScript (creada por Facebook en 2013) para construir interfaces de usuario con componentes reutilizables.
- Un componente es una pieza independiente de la interfaz; se combinan como piezas de LEGO para armar aplicaciones completas.
- El DOM virtual es una copia ligera del DOM en memoria: React compara versiones y solo actualiza lo que cambió en el DOM real, lo que hace las apps rápidas.
- React es declarativo: describes QUÉ debe verse según los datos, y React se encarga del CÓMO actualizar la pantalla.
- Node.js permite correr JavaScript fuera del navegador e incluye npm, el manejador de paquetes para instalar librerías como React.
- Vite es la herramienta moderna para crear proyectos React: "npm create vite@latest mi-app -- --template react", luego "npm install" y "npm run dev".
- Tu código vive en `src/`: `main.jsx` conecta React con el HTML, `App.jsx` es tu componente raíz, y todo cuelga de ahí.
- React es JavaScript: entre más sólido tu JavaScript, más natural se sentirá React.

UNIDAD 2: Componentes, JSX y Props

Bienvenido a la Unidad 2 del Curso Completo de React de Lucas EGA Academy. Si en la Unidad 1 aprendiste qué es React, por qué existe y cómo preparar tu entorno de trabajo con Vite, ahora si viene lo bueno: vas a conocer las tres piezas que forman el corazón de React. Sin estas tres piezas no hay React que valga, así de simple. Hablamos de los componentes, de JSX y de las props.

Piénsalo así: en la Unidad 1 te entregamos la caja de herramientas y te enseñamos a abrir el taller. En esta Unidad 2 vas a aprender a usar las tres herramientas principales del taller. Todo lo demás que verás en el curso (el estado, los efectos, los hooks, el manejo de formularios, el consumo de APIs) se construye encima de estas tres ideas. Si las entiendes bien, el resto del curso se te va a hacer mucho más fácil. Si las entiendes a medias, el resto del curso se te va a complicar. Así que vale la pena que le pongas atención completa a esta unidad.

Que vas a ver exactamente. Primero, la Clase 4: tu primer componente. Vas a descubrir que un componente no es magia ni una cosa rara: es una función de JavaScript que devuelve JSX. Y vas a aprender a usarlo como si fuera una etiqueta de HTML, algo como `<Saludo />`. Parece poco, pero este es el concepto más importante de todo React. Segundo, la Clase 5: JSX. Esa sintaxis que parece HTML metido dentro de JavaScript. Vas a entender qué es, por qué existe, como las llaves `{}` te permiten ejecutar código JavaScript dentro de tu interfaz, y que por debajo JSX se convierte en JavaScript normal y corriente. Tercero, la Clase 6: las props. Vas a aprender que los componentes reciben datos del exterior, igual que una función recibe parámetros, y que gracias a eso puedes crear un solo componente y usarlo muchas veces con datos distintos. Eso es la reutilización, y es la razón por la que las empresas aman React.

Por qué estos tres temas son el corazón de React. Imagínate que quieres construir una casa. Puedes hacerlo ladrillo por ladrillo, poniendo cada ladrillo a mano, midiendo, cortando y pegando uno por uno. Eso sería como construir una página web escribiendo todo el HTML a mano, repitiendo código una y otra vez. Funciona, pero es lento, es fácil equivocarse y cuando quieres cambiar algo (por ejemplo el color de todas las ventanas) tienes que ir ventana por ventana. React propone otra forma: en lugar de ladrillos sueltos, construyes bloques prefabricados. Un bloque para las ventanas, un bloque para las puertas, un bloque para las paredes. Cada bloque se fabrica una vez y luego lo colocas donde lo necesites, las veces que lo necesites. Esos bloques prefabricados son los componentes. JSX es el lenguaje con el que describes cómo se ve cada bloque, y las props son las opciones de cada bloque: la ventana puede ser grande o chica, de madera o de aluminio, pero sigue siendo el mismo bloque ventana. Con esa idea puedes levantar casas enteras (aplicaciones completas) rapidísimo y sin repetir trabajo.

Otra analogía que ayuda mucho: piensa en una cocina de restaurante. El chef no cocina cada platillo desde cero inventando la receta en el momento. Tiene recetas (componentes) que describen exactamente cómo se prepara cada platillo. La receta de "tacos" dice que lleva tortilla, carne, cebolla y cilantro, y explica el orden de preparación. Esa receta escrita en un lenguaje que mezcla instrucciones con ingredientes es como JSX: mezcla la estructura con la lógica. Y las props son las variaciones que pide el cliente: "quiero mis tacos sin cebolla", "quiero la carne bien cocida", "quiero salsa extra". La receta es la misma, pero cada orden sale distinta según lo que pida cada cliente. Así trabajan los componentes con props: un solo componente `TarjetaProducto` puede mostrar cien productos distintos porque cada vez recibe props distintas.

También puedes pensarlo como piezas de LEGO. Cada pieza tiene una forma definida y un propósito, y las piezas se conectan entre sí para formar cosas más grandes. Un componente pequeño (un botón) se conecta con otros (un formulario) para formar algo más grande (una página de registro), que a su vez se conecta con otras páginas para formar la aplicación completa. Y lo bonito de LEGO es que si una pieza está bien hecha, la puedes usar en cien construcciones distintas. Eso es exactamente lo que buscamos con componentes reutilizables y props.

En esta unidad no solo vas a leer teoría. La estructura está pensada para que aprendas haciendo. Después de la introducción encontrarás una explicación detallada donde desmenuzamos cada concepto con analogías de la vida real, te advertimos sobre los errores más comunes de los principiantes (esos errores que todos cometemos y que te hacen perder horas si nadie te los explica) y te compartimos buenas prácticas de la industria, o sea, como escriben componentes los programadores profesionales en empresas reales. Luego vienen ejemplos paso a paso: cuatro ejemplos completos de código explicados línea por línea, con instrucciones para probarlos en tu proyecto de Vite. Después, la sección de práctica con cinco ejercicios: aquí la regla de oro es que intentes resolver cada ejercicio ANTES de ver la solución. La solución está completa y explicada, pero solo te servirá de verdad si primero lo intentas tú. Equivocarse es parte del aprendizaje. Y cerramos con un resumen de la unidad para repasar los puntos clave.

Un consejo antes de empezar: ten tu proyecto de Vite abierto mientras lees. Cada vez que veas un ejemplo de código, copialo, pégalo en tu proyecto y ejecútalo con `npm run dev`. Mueve cosas, rompe el código a propósito y fíjate qué error sale. Esa curiosidad es lo que separa a quien solo lee de quien realmente aprende a programar. React se aprende con las manos en el teclado, no solo con los ojos en la pantalla.

Otra cosa importante: no te asustes si al principio JSX se te hace raro. A todo el mundo le pasa. Ver HTML dentro de JavaScript se siente antinatural las primeras veces, cómo escribir

con la mano izquierda si eres diestro. Pero después de unos días escribiendo componentes, tu cerebro hace clic y ya no podrás imaginarlo de otra forma. Ten paciencia contigo mismo. Los tres temas de esta unidad (componentes, JSX y props) son cómo aprender a andar en bicicleta: al inicio te tambaleas, pero una vez que agarras el equilibrio, ya no se te olvida.

Para aprovechar esta unidad al máximo necesitas lo siguiente: tu proyecto de React creado con Vite funcionando (lo hiciste en la Unidad 1), un editor de código como Visual Studio Code, y ganas de practicar. No necesitas saber nada más. Todo lo que sigue está explicado desde cero, paso a paso, en español claro y con ejemplos que puedes probar de inmediato.

Algo que vas a notar en esta unidad es que repetimos mucho una idea: los componentes son funciones. No clases complicadas, no objetos misteriosos: funciones de JavaScript normales, de las que ya conoces. Eso es una gran noticia, porque significa que todo lo que sabes de funciones (parámetros, return, variables) te sirve tal cual en React. React no te pide que aprendas un lenguaje nuevo desde cero; te pide que uses tu JavaScript de todos los días de una forma organizada. Los componentes, JSX y las props son simplemente JavaScript bien organizado para construir interfaces.

Vamos a arrancar. En la siguiente sección desarmamos pieza por pieza que es un componente, cuáles son sus reglas, que es JSX por dentro, cómo se compila, cómo se usan las llaves, por qué `className` existe, que son las props, cómo se desestructuran, que es `children`, por qué los datos fluyen en una sola dirección y por qué nunca debes modificar las props. También veremos los errores típicos (como olvidar la mayúscula en el nombre del componente, que es el error número uno de los principiantes) y las buenas prácticas que usan los profesionales. Al final de la unidad habrás construido tus primeros componentes reales, los habrás alimentado con props y entenderás por dentro cómo React convierte tu JSX en lo que el navegador muestra. Manos a la obra.

Explicación detallada

Que es un componente

Un componente en React es una función de JavaScript que devuelve JSX. Punto. Esa es toda la definición, y conviene que te la grabes porque es la base de todo. Mira este ejemplo, el más pequeño posible:

```
function Saludo() {  
  return <h1>Hola, mundo</h1>;  
}
```

Eso es un componente. Una función llamada Saludo que devuelve un `<h1>`. No hay más misterio. Cuando React encuentra `<Saludo />` en tu código, lo que hace es ejecutar la función Saludo y colocar en la pantalla lo que esa función devolvio. Así de directo.

Fíjate en lo poderoso que es esto: como un componente es una función, puedes usar dentro de el todo lo que sabes de JavaScript. Variables, condicionales if, ciclos, funciones auxiliares, calculos, arreglos. Todo vale. Por ejemplo:

```
function Saludo() {  
  const nombre = "Esteban";  
  const hora = new Date().getHours();  
  const mensaje = hora < 12 ? "Buenos dias" : "Buenas tardes";  
  return <h1>{mensaje}, {nombre}</h1>;  
}
```

Aquí el componente usa variables, un objeto Date, un operador ternario y luego devuelve JSX que muestra el resultado. La función hace su lógica en JavaScript normal y al final devuelve la descripción de lo que se debe ver en pantalla. Esa es la esencia: lógica arriba, descripción visual abajo en el return.

Ahora, cómo se usa un componente. Una vez que lo defines, lo usas como si fuera una etiqueta de HTML. Si tu componente se llama Saludo, lo escribes así: `<Saludo />`. Observa la diagonal al final: como normalmente no lleva contenido adentro, se escribe como etiqueta autocerrada, igual que `<img />` o `<br />` en HTML. Y lo puedes usar las veces que quieras:

```
function App() {  
  return (  
    <div>  
      <Saludo />  
      <Saludo />  
      <Saludo />  
    </div>  
  );  
}
```

Esto mostraría "Hola, mundo" tres veces. App también es un componente (la función principal de tu aplicación), y dentro de él estamos usando el componente Saludo tres veces. Esto ilustra la gran idea: los componentes se componen entre sí, o sea, se combinan como piezas para formar cosas más grandes. App es un componente que usa a Saludo. Mañana harás un componente Página que use a Encabezado, a ListaDeProductos y a PieDePagina. Y así se construyen aplicaciones completas: componentes dentro de componentes, como muñecas rusas.

Analogía de la vida real: un componente es como una receta de cocina escrita en una tarjeta. La tarjeta dice el nombre del platillo (el nombre de la función), la lista de ingredientes que necesita (las props, que veremos más adelante) y los pasos de preparación (el cuerpo de la función). Cada vez que quieres servir ese platillo, sacas la tarjeta y sigues la receta. Puedes preparar el mismo platillo cien veces con la misma tarjeta. Y puedes combinar recetas: la receta de "comida completa" usa la tarjeta de "sopa", la tarjeta de "plato fuerte" y la tarjeta de "postre". En React pasa igual: defines el componente una vez y lo usas donde quieras, combinado con otros.

Otra analogía útil: piensa en los sellos de goma de la papelería. Tallan el sello una vez (definen el componente) y luego lo pueden estampar cien veces en cien hojas distintas (usar el componente muchas veces). Cada estampado sale igual porque viene del mismo sello. Si quieren cambiar el diseño, retallan el sello una sola vez y todos los estampados futuros salen con el nuevo diseño. Así funcionan los componentes: cambias el componente en un solo archivo y todas las partes de tu aplicación donde lo usaste se actualizan automáticamente. Imagínate lo útil que es esto para un botón que aparece en cincuenta pantallas: lo cambias una vez y queda cambiado en las cincuenta.

Reglas de los componentes

Los componentes tienen reglas sencillas pero estrictas. Si las rompes, React te lo hará saber con errores. Mejor aprenderlas desde ahora.

Regla 1: el nombre del componente debe empezar con mayúscula. Esta es la regla número uno y el error número uno de los principiantes. En React, las etiquetas que empiezan con minúscula se interpretan como etiquetas HTML normales (div, p, h1, span), y las que empiezan con mayúscula se interpretan como componentes. Mira la diferencia:


```
function saludo() {
  return <h1>Hola</h1>;
}

function App() {
  return (
    <div>
      <saludo />
    </div>
  );
}
```

Esto NO funciona como esperas. React ve `<saludo />` con minúscula y piensa: "esto debe ser una etiqueta HTML llamada saludo", y como esa etiqueta no existe en HTML, no se muestra nada o sale un error raro. En cambio, si lo llamas Saludo con mayúscula, React entiende: "esto es un componente, voy a ejecutar la función Saludo". La mayúscula es la señal que le dice a React que ejecute tu función. Grábatelo: componentes siempre con mayúscula inicial. Saludo, TarjetaProducto, ListaDeUsuarios, BotonPrincipal. Si usas varias palabras, se acostumbra el estilo PascalCase: cada palabra empieza con mayúscula y van pegadas, como en TarjetaProducto o FormularioDeContacto.

Regla 2: el componente debe devolver un solo elemento raíz. Esto significa que todo lo que devuelvas en el return tiene que estar envuelto en una sola etiqueta padre. Esto está mal:

```
function Tarjeta() {
  return (
    <h2>Titulo</h2>
    <p>Descripcion</p>
  );
}
```

React se queja porque el return tiene dos elementos hermanos (h2 y p) sin un padre que los envuelva. Es como si una función quisiera devolver dos cosas a la vez: no se puede, solo puede devolver una. La solución es envolverlos en un div o en un fragmento:

```
function Tarjeta() {
  return (
    <div>
      <h2>Titulo</h2>
      <p>Descripcion</p>
    </div>
  );
}
```

Ahora el return devuelve una sola cosa: el div, que por dentro contiene lo demás. Problema resuelto.

Pero a veces no quieres meter un div extra solo para cumplir la regla, porque ese div de más puede arruinar tu diseño o tu CSS. Para esos casos existen los fragmentos (fragments). Un fragmento es una envoltura invisible: agrupa elementos sin agregar ninguna etiqueta real al HTML final. Se escribe así:

```
function Tarjeta() {
  return (
    <>
      <h2>Titulo</h2>
      <p>Descripcion</p>
    </>
  );
}
```

Los símbolos <> y </> son la forma corta de escribir un fragmento. Agrupan el h2 y el p para cumplir la regla del elemento único, pero en el HTML final no aparece ningún div extra: solo verás el h2 y el p. Es como una bolsa transparente: junta las cosas pero no se ve. Los fragmentos son muy usados por los profesionales porque mantienen el HTML limpio.

Regla 3: un componente es una función pura en cuanto a su renderizado. Esto suena técnico pero es simple: dado los mismos datos de entrada (las props), el componente siempre debe devolver lo mismo en pantalla. No debe hacer cosas raras como modificar variables globales o cambiar las props que recibe. De esto hablaremos más cuando veamos que las props son inmutables.

Regla 4 (buena práctica fuerte): un componente por archivo. Aunque JavaScript te permite definir varios componentes en el mismo archivo, la convención en la industria es que cada componente viva en su propio archivo, y que el archivo se llame igual que el componente. Por ejemplo, el componente Saludo vive en Saludo.jsx, el componente TarjetaProducto vive en

TarjetaProducto.jsx. Esto hace que los proyectos grandes sean fáciles de navegar: buscas el archivo con el nombre del componente y ahí está. Cuando un proyecto tiene cientos de componentes, esta organización te salva la vida.

JSX a fondo

JSX significa JavaScript XML. Es una extensión de la sintaxis de JavaScript que te permite escribir algo parecido a HTML dentro de tu código JavaScript. Mira de nuevo:

```
const elemento = <h1>Hola, {nombre}</h1>;
```

Esa línea mezcla HTML (<h1>) con JavaScript (la variable nombre entre llaves). Los navegadores no entienden JSX directamente. Entonces, cómo funciona. La respuesta: las herramientas de construcción (como Vite, que usa esbuild por debajo) convierten tu JSX en llamadas de JavaScript normal antes de que el código llegue al navegador. Esa conversión se llama compilación o transpilación.

¿Qué es lo que realmente se ejecuta. Cuando escribes:

```
const elemento = <h1>Hola, {nombre}</h1>;
```

La herramienta lo convierte en algo parecido a esto:

```
const elemento = React.createElement("h1", null, "Hola, ", nombre);
```

React.createElement es una función que crea un objeto que describe el elemento: qué tipo de etiqueta es ("h1"), qué atributos tiene (null porque no tiene) y qué contenido lleva ("Hola, " y el valor de nombre). A ese objeto se le llama elemento de React. React luego usa esos objetos para construir y actualizar la página de forma eficiente. Tu nunca escribes React.createElement a mano (sería tedioso), pero es útil saber que existe, porque explica por qué JSX funciona: no es magia, es azúcar sintáctica, o sea, una forma más cómoda de escribir lo mismo.

Analogía: JSX es cómo escribir una carta usando una plantilla con espacios en blanco. La plantilla dice "Hola, _____, bienvenido a _____". Tu rellenas los espacios con los datos (el nombre, la academia). La plantilla en sí no es la carta final: alguien (la herramienta de compilación) toma tu plantilla, rellena los espacios y produce la carta terminada. JSX es la plantilla cómoda para ti; React.createElement es el trabajo mecánico que ocurre por debajo.

Las llaves {} ejecutan código JavaScript. Esta es la característica más importante de JSX. Todo lo que pongas entre llaves dentro de JSX se evalúa como JavaScript normal. Puedes poner variables, operaciones, llamadas a funciones, ternarios, acceso a propiedades de objetos:

```
function Perfil() {
  const usuario = { nombre: "Ana", edad: 28 };
  const esMayor = usuario.edad >= 18;
  return (
    <div>
      <h1>{usuario.nombre}</h1>
      <p>Tienes {usuario.edad} años</p>
      <p>2 + 2 = {2 + 2}</p>
      <p>{esMayor ? "Eres mayor de edad" : "Eres menor de edad"}</p>
      <p>{usuario.nombre.toUpperCase()}</p>
    </div>
  );
}
```

Fíjate en cada caso: {usuario.nombre} muestra el valor de la variable, {2 + 2} calcula y muestra 4, el ternario elige un texto según la condición, y {usuario.nombre.toUpperCase()} ejecuta un método. Todo eso es JavaScript corriendo dentro de las llaves. Las llaves son como pequeñas ventanas por donde tu JavaScript se asoma a tu HTML.

Hay límites para lo que puedes meter entre llaves: tiene que ser una expresión, o sea, algo que produce un valor. Puedes usar ternarios, pero no puedes usar un if completo con llaves de bloque ni un ciclo for con su sintaxis de sentencia, porque esas son sentencias, no expresiones. Si necesitas lógica más compleja, la haces arriba del return (en el cuerpo de la función) y en el JSX solo muestras el resultado con llaves. Este patrón (lógica arriba, JSX abajo) es el más común y el más limpio:

```
function Lista() {
  const frutas = ["manzana", "pera", "uva"];
  const elementos = frutas.map((fruta) => <li>{fruta}</li>);
  return <ul>{elementos}</ul>;
}
```

Aquí el .map se ejecuta arriba y genera un arreglo de elementos . En el JSX solo ponemos {elementos} para mostrarlos. Limpio y fácil de leer.

Atributos en JSX. Las etiquetas JSX aceptan atributos parecido a HTML, pero con diferencias importantes. En HTML escribes class para las clases de CSS; en JSX debes escribir

className. Por qué. Porque class es una palabra reservada de JavaScript (se usa para definir clases), y como JSX vive dentro de JavaScript, usar class causaría conflictos. Así que la regla es: en JSX, class se convierte en className:

```
function Boton() {  
  return <button className="boton-principal">Haz clic</button>;  
}
```

Esto genera un botón con la clase CSS "botón-principal". Si por costumbre escribes class, React te mostrara una advertencia en la consola. Otro atributo que cambia es for (el de las etiquetas label en formularios), que en JSX se escribe htmlFor, por la misma razón: for es palabra reservada en JavaScript (se usa en los ciclos for).

```
function Campo() {  
  return (  
    <div>  
      <label htmlFor="nombre">Nombre:</label>  
      <input id="nombre" type="text" />  
    </div>  
  );  
}
```

Los atributos que reciben valores dinámicos usan llaves. Por ejemplo, si la imagen depende de una variable:

```
function Avatar() {  
  const url = "https://ejemplo.com/foto.jpg";  
  const descripcion = "Foto de perfil";  
  return <img src={url} alt={descripcion} />;  
}
```

Aquí src={url} le pasa el valor de la variable url al atributo src. Sin llaves, src="url" pasaría el texto literal "url", que no es lo que queremos. Regla práctica: texto fijo entre comillas, valores dinámicos entre llaves.

Los eventos también son atributos en JSX, y usan llaves porque reciben funciones: <button onClick={saludar}>Haz clic</button>. Aquí onClick recibe la función saludar (sin paréntesis, porque no la estamos ejecutando, solo se la estamos pasando al botón para que la ejecute cuando le den clic). De los eventos hablaremos a fondo en otra unidad, pero conviene que lo veas desde ahora porque es un uso típico de llaves con funciones.

Comentarios en JSX. En JavaScript normal los comentarios se escriben con `//` o con `/* */`. Dentro de JSX eso no funciona directamente, porque JSX espera etiquetas o expresiones. Para comentar dentro de JSX debes envolver el comentario en llaves, así:

```
function Tarjeta() {
  return (
    <div>
      {/* Este es un comentario dentro de JSX */}
      <h2>Titulo</h2>
      {/* <p>Este parrafo esta comentado y no se muestra</p> */}
    </div>
  );
}
```

El truco es `{/* */}`: las llaves le dicen a JSX "aquí va JavaScript", y dentro va un comentario de JavaScript `/* */`. Así puedes dejar notas o desactivar partes de tu interfaz temporalmente. Fuera del JSX (en el cuerpo de la función, arriba del `return`) los comentarios normales `//` funcionan sin problema.

Otra diferencia con HTML: en JSX todas las etiquetas deben cerrarse. En HTML puedes escribir `<br>` o `<img>` sin cerrar; en JSX debes escribir `<br />` y `<img />`. Si olvidas la diagonal de cierre, la compilación falla con un error. Es una regla estricta pero fácil de acostumbrarse.

También los atributos booleanos: en HTML escribes `<input disabled>`; en JSX puedes escribir `<input disabled={true} />` o la forma corta `<input disabled />`. Y los estilos en línea no son texto como en HTML (`style="color: red"`), sino un objeto de JavaScript: `style={{ color: "red", fontSize: "20px" }}`. Fíjate en las dobles llaves: las de afuera son las llaves de JSX ("aquí va JavaScript") y las de adentro son las llaves del objeto. Además, las propiedades CSS con guion se escriben en camelCase: `font-size` se vuelve `fontSize`, `background-color` se vuelve `backgroundColor`. Es un detalle que al inicio se olvida mucho.

Props a fondo

Las props (abreviatura de properties, propiedades) son los datos que recibe un componente desde afuera. Si un componente es una función, las props son sus parámetros. Mira:

```
function Saludo({ nombre }) {
  return <h1>Hola, {nombre}</h1>;
}
```

Este componente Saludo recibe una prop llamada nombre y la muestra. Y así se usa, pasándole el dato:

```
function App() {
  return (
    <div>
      <Saludo nombre="Ana" />
      <Saludo nombre="Luis" />
      <Saludo nombre="Maria" />
    </div>
  );
}
```

El resultado en pantalla: "Hola, Ana", "Hola, Luis", "Hola, Maria". Un solo componente, tres resultados distintos. Esa es la magia de las props: convierten un componente fijo en un componente reutilizable. Sin props, Saludo siempre diría lo mismo y no serviría de mucho. Con props, el mismo componente sirve para saludar a cualquier persona.

Analogía de la vida real: las props son como los ingredientes que le pasas a la receta. La receta de "licuado" dice: toma fruta, leche y azúcar, licualo todo. Pero cada vez que la preparas le pasas ingredientes distintos: hoy fruta = fresa, mañana fruta = plátano. La receta (el componente) no cambia; lo que cambia son los ingredientes (las props) que recibe cada vez. Otro ejemplo: una fábrica de playeras. La máquina que stampa playeras es siempre la misma (el componente), pero cada orden le indica un diseño, una talla y un color distintos (las props). Misma máquina, productos distintos según los datos de entrada.

Como llegan las props al componente. Cuando escribes `<Saludo nombre="Ana" />`, React junta todos los atributos que le pasaste y los mete en un solo objeto. Ese objeto es el primer parámetro de tu función componente. Tienes dos formas de recibirlo.

Forma 1: recibir el objeto completo y acceder con punto:

```
function Saludo(props) {
  return <h1>Hola, {props.nombre}</h1>;
}
```

Aquí props es el objeto `{ nombre: "Ana" }`, y accedemos con `props.nombre`. Esta forma deja muy claro que los datos vienen de las props, pero si usas muchas props se vuelve repetitivo escribir props. esto y props. lo otro.

Forma 2: desestructuración, la favorita de los profesionales:

```
function Saludo({ nombre }) {  
  return <h1>Hola, {nombre}</h1>;  
}
```

La desestructuración es una característica de JavaScript que extrae propiedades de un objeto directamente en los parámetros. En lugar de recibir props y luego escribir props.nombre, extraes nombre desde el inicio y lo usas directo. Cuando un componente recibe varias props, la desestructuración mantiene el código limpio:

```
function TarjetaProducto({ nombre, precio, descripcion }) {  
  return (  
    <div>  
      <h2>{nombre}</h2>  
      <p>{descripcion}</p>  
      <p>Precio: ${precio}</p>  
    </div>  
  );  
}
```

Y se usa así:

```
<TarjetaProducto nombre="Laptop" precio={999} descripcion="Computadora portatil  
potente" />
```

Observa que precio={999} usa llaves porque 999 es un número (JavaScript), mientras que nombre="Laptop" usa comillas porque es texto fijo. Si escribieras precio="999", llegaría como texto "999" y no como número 999. Para textos simples usa comillas; para números, arreglos, objetos, booleanos o variables usa llaves. También puedes pasar arreglos y objetos completos como props:

```
<ListaDeUsuarios usuarios=[["Ana", "Luis", "Maria"]] />  
<Perfil datos={{ nombre: "Ana", edad: 28 }} />
```

Las props pueden ser cualquier valor de JavaScript: texto, números, booleanos, arreglos, objetos e incluso funciones. Esa flexibilidad es lo que hace a los componentes tan poderosos.

Props por defecto. A veces un componente se usa sin pasarle alguna prop, y no quieres que se rompa o que muestre "undefined" en pantalla. Para eso existen los valores por defecto en la desestructuración:


```
function Saludo({ nombre = "visitante" }) {  
  return <h1>Hola, {nombre}</h1>;  
}
```

Si usas `<Saludo />` sin pasar nombre, mostrara "Hola, visitante". Si pasas `<Saludo nombre="Ana" />`, mostrara "Hola, Ana". El valor por defecto solo se usa cuando la prop no llega o llega como undefined. Es una forma elegante de hacer tus componentes más robustos. Los profesionales usan valores por defecto para las props opcionales, o sea, las que el componente puede funcionar sin ellas.

children: la prop especial. Hay una prop que no se pasa como atributo, sino como contenido entre la etiqueta de apertura y la de cierre. Se llama children (hijos). Mira:

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

Y se usa así:

```
<Tarjeta>  
  <h2>Mi titulo</h2>  
  <p>Este contenido va dentro de la tarjeta.</p>  
</Tarjeta>
```

Todo lo que pongas entre `<Tarjeta>` y `</Tarjeta>` llega al componente como la prop children. El componente Tarjeta decide donde colocar esos hijos dentro de su estructura (en este caso, dentro del div con clase "tarjeta"). Esto es utilísimo para crear componentes contenedores: tarjetas, modales, paneles, secciones. El contenedor define el marco (bordes, fondo, padding) y quien lo usa decide que contenido va adentro. Es como un marco de fotos: el marco es siempre el mismo, pero cada quien mete la foto que quiera.

Fíjate que children te permite dos estilos de uso: `<Tarjeta />` autocerrado cuando no hay contenido, y `<Tarjeta>...</Tarjeta>` con contenido cuando si lo hay. Ambos son validos y se eligen según lo que necesites.

Flujo unidireccional: los datos van del padre al hijo. En React, las props siempre fluyen en una sola dirección: del componente padre al componente hijo. El padre es quien usa al hijo y le pasa datos; el hijo los recibe y los muestra, pero no puede cambiarlos ni devolverlos al padre por el mismo camino. Es como una cascada: el agua baja, nunca sube.

```
function App() {
  const nombreUsuario = "Esteban";
  return (
    <div>
      <Saludo nombre={nombreUsuario} />
    </div>
  );
}
```

Aquí App es el padre y Saludo es el hijo. App tiene el dato (nombreUsuario) y se lo pasa a Saludo mediante la prop nombre. Saludo lo muestra. Si Saludo quisiera cambiar el nombre, no puede hacerlo modificando la prop; tendría que pedirle al padre que lo cambie por otro mecanismo (el estado, que verás en la Unidad 3). Esta restricción no es un capricho: hace que las aplicaciones sean predecibles. Siempre sabes de donde viene cada dato: viene del padre. Cuando algo se muestra mal en pantalla, sabes exactamente donde buscar: en el componente padre que paso la prop. En aplicaciones grandes, esta predictibilidad vale oro.

Props inmutables: nunca modifiques las props. Esta es una regla de oro. Las props son de solo lectura. Tu componente las recibe y las muestra, pero jamás debe cambiarlas. Esto está prohibido:

```
function Saludo({ nombre }) {
  nombre = nombre.toUpperCase();
  return <h1>Hola, {nombre}</h1>;
}
```

Reasignar el parámetro nombre es mutar la prop, y aunque JavaScript lo permita técnicamente, en React es una mala práctica grave. Por qué. Porque rompe la predictibilidad: el padre cree que paso "Ana" y el hijo muestra "ANA" sin que el padre lo sepa. Si necesitas transformar el dato, crea una variable nueva:

```
function Saludo({ nombre }) {
  const nombreMayusculas = nombre.toUpperCase();
  return <h1>Hola, {nombreMayusculas}</h1>;
}
```

Así la prop original queda intacta y tu transformación vive en una variable local. Recuerda la regla de las funciones puras: con las mismas props de entrada, el componente siempre muestra lo mismo. Si mutas las props, esa garantía se rompe y aparecen errores difíciles de rastrear.

Errores comunes de principiantes

Vamos con los errores que casi todos cometemos al empezar. Conocerlos de antemano te ahorra horas de frustración.

Error 1: olvidar la mayúscula en el nombre del componente. Ya lo vimos en las reglas, pero lo repetimos porque es el error más común. Escribes `function saludo()` y luego `<saludo />` y no se muestra nada. React lo trata como etiqueta HTML desconocida. Solución: siempre PascalCase. Saludo, TarjetaProducto, MiBoton. Si algo no se renderiza y no hay error claro, lo primero que debes revisar es la mayúscula.

Error 2: devolver varios elementos sin envolverlos. Escribes dos etiquetas hermanas en el return sin un padre y la compilación falla con un mensaje sobre "adjacent JSX elements". Solución: envuelve todo en un `div` o usa un fragmento `<>`. Con la práctica, envolver se vuelve automático.

Error 3: mutar las props. Reasignar una prop dentro del componente o modificar un objeto que llegó como prop. Los síntomas son comportamientos raros: datos que cambian sin explicación, componentes que muestran cosas distintas con las mismas props. Solución: trata las props como solo lectura; si necesitas otro valor, crea variables nuevas.

Error 4: llaves mal usadas. Dos variantes: olvidar las llaves cuando el valor es dinámico (escribir `src="url"` en lugar de `src={url}`, con lo que pasas el texto "url" en vez de la variable), y poner llaves de más o usar comillas con llaves (escribir `nombre={"Ana"}` funciona pero es innecesario; peor es escribir `nombre="{nombre}"` que pasa el texto literal "{nombre}"). Regla: texto fijo con comillas, todo lo demás con llaves. Y nunca mezcles: o comillas o llaves, no ambas.

Error 5: usar `class` en lugar de `className`. Por costumbre de HTML escribes `<div class="caja">` y React te lanza una advertencia. No se rompe del todo, pero es incorrecto y la consola se llena de warnings. Solución: en JSX siempre `className`. Lo mismo con `for`, que debe ser `htmlFor`.

Error 6: olvidar cerrar las etiquetas. Escribes `<img>` sin la diagonal y la compilación falla. En HTML el navegador lo perdona; en JSX no hay perdón. Solución: toda etiqueta sin contenido se autocierra: `<img />`, `<br />`, `<input />`, `<Saludo />`.

Error 7: intentar usar sentencias dentro de las llaves. Escribes algo como `{ if (x) { ... } }` o un `for` dentro del JSX y falla, porque entre llaves solo van expresiones. Solución: haz la lógica arriba del return y en el JSX solo muestra variables o ternarios. Este patrón te acostumbra a separar lógica de presentación, que es una buena práctica en sí misma.

Error 8: pasar props al componente equivocado o con el nombre mal escrito. Escribes `<Saludo Nombre="Ana" />` con mayúscula en "Nombre" pero el componente espera nombre con minúscula. JavaScript distingue mayusculas de minusculas, así que la prop no llega y el componente muestra undefined. Solución: revisa que el nombre de la prop sea identico en el uso y en la definición. Este error es traicionero porque no siempre lanza un error visible: simplemente se muestra undefined o el valor por defecto.

Error 9: confundir el objeto props con las props desestructuradas. Defines function `Saludo({ nombre })` pero luego escribes `props.nombre` dentro del JSX, y falla porque props no existe como variable (solo existe nombre). O al reves: defines function `Saludo(props)` y escribes `{nombre}` directo, y falla porque nombre no existe (solo props.nombre). Solución: se consistente. Si desestructuras, usa los nombres directos. Si no, usa props.algo.

Error 10: no pasar la key al renderizar listas (adelanto). Cuando uses `.map` para generar listas de componentes, cada elemento necesita una prop especial llamada key. De esto hablaremos a fondo en su unidad, pero que no se te olvide: sin key, React te advierte y el rendimiento y el comportamiento de la lista pueden fallar.

Buenas prácticas de la industria

Como escriben componentes los profesionales en empresas reales. Estas prácticas no son obligatorias para que el código funcione, pero separan el código de juguete del código profesional.

Componentes pequenos y con una sola responsabilidad. Cada componente debe hacer una sola cosa y hacerla bien. Un componente `TarjetaProducto` muestra una tarjeta de producto; no debe además manejar el carrito de compras, ni hacer la petición a la API, ni mostrar el menu. Si un componente crece demasiado (más de 50 o 100 líneas es una senal de alerta), dividelo en componentes más chicos. Los componentes pequenos son más fáciles de entender, de probar y de reutilizar. Piensa en piezas de LEGO pequenas: con muchas piezas chicas puedes construir lo que sea; con pocas piezas gigantes tus opciones se reducen.

Un componente por archivo, y el archivo se llama igual que el componente. `Saludo.jsx` contiene al componente `Saludo`. Esto es un estandar en la industria. Cuando alguien nuevo entra al proyecto, encuentra cada componente en segundos. Además, los editores de código y las herramientas están optimizados para este patrón.

Nombres descriptivos. El nombre del componente debe decir que hace o que muestra: `TarjetaProducto`, `FormularioDeContacto`, `ListaDeTareas`, `BotonPrincipal`. Evita nombres genericos como `Cosa`, `Componente1`, `MiDiv` o `Test`. Un buen nombre es documentación gratis: quien lea `<TarjetaProducto />` entiende que ahí se muestra una tarjeta de producto sin tener que abrir el archivo.

Desestructura las props en los parámetros. Como vimos, `function Saludo({ nombre })` es más limpio que `function Saludo(props)` con `props.nombre` por todos lados. Además, la desestructuración funciona como documentación: con solo ver los parámetros sabes que datos necesita el componente. Algunos equipos incluso ordenan las props alfabeticamente cuando son muchas.

Define valores por defecto para las props opcionales. Si una prop puede no llegar, dale un valor por defecto en la desestructuración. Así tu componente nunca muestra `undefined` por accidente y quien lo usa no esta obligado a pasar todas las props siempre.

No repitas código: si ves el mismo bloque de JSX dos o tres veces, es señal de que necesitas un componente. La duplicación es el enemigo. Cada vez que copias y pegas JSX, preguntate: "esto debería ser un componente con props". Casi siempre la respuesta es sí.

Manten la lógica fuera del JSX cuando sea compleja. El JSX debe ser declarativo y fácil de leer: describir que se muestra, no cómo se calcula. Los calculos, filtros y transformaciones van arriba del `return`, en variables con nombres claros. Un `return` limpio se lee como una historia; un `return` lleno de lógica anidada es un dolor de cabeza.

Usa `children` para componentes contenedores. Si haces un componente que envuelve contenido (tarjeta, modal, panel, sección), recibe `children` y colocalo donde corresponda. Así separas el marco del contenido y tu componente sirve para cualquier contenido, no solo para uno fijo.

Ejemplos paso a paso

En esta sección vamos a construir cuatro ejemplos completos, de menos a más complejidad. Cada ejemplo esta explicado línea por línea y trae instrucciones para probarlo en tu proyecto de Vite. La mecánica para probar todos es la misma: abre tu proyecto, crea un archivo nuevo en la carpeta `src` con el nombre indicado, pega el código, importa el componente en `src/App.jsx`, guarda y mira el resultado en el navegador con `npm run dev`. Si el proyecto no está corriendo, abre la terminal en la carpeta del proyecto y ejecuta `npm run dev`.

Antes de empezar, recuerda cómo se ve un App.jsx típico de Vite. Lo vas a modificar en cada ejemplo para mostrar tu componente. Un App.jsx mínimo se ve así:

```
function App() {  
  return (  
    <div>  
      <h1>Mi app</h1>  
    </div>  
  );  
}  
  
export default App;
```

La línea `export default App` es la que permite que `main.jsx` importe tu componente `App`. Cuando crees tus propios componentes en archivos separados, también necesitarán su `export default` para poder importarlos. Lo verás en cada ejemplo.

Ejemplo 1: componente Saludo

El ejemplo más simple posible: un componente que saluda. Sirve para practicar la definición de un componente, su uso como etiqueta y las reglas básicas.

Paso 1: crea el archivo `src/Saludo.jsx` con este contenido:

```
function Saludo() {  
  return <h1>Hola, bienvenido a React</h1>;  
}  
  
export default Saludo;
```

Explicación línea por línea. Línea 1: `function Saludo()` define una función de JavaScript llamada `Saludo`. El nombre empieza con mayúscula, que es la regla número uno: así React sabe que es un componente y no una etiqueta HTML. No recibe parámetros porque este primer ejemplo no usa props. Línea 2: `return <h1>Hola, bienvenido a React</h1>;` devuelve JSX: un encabezado `h1` con un texto. Todo componente debe devolver JSX (o algo que React pueda mostrar). Línea 3: la llave cierra la función. Línea 5: `export default Saludo` permite que otros archivos importen este componente. Sin esta línea, `App.jsx` no podría usarlo.

Paso 2: usa el componente en `src/App.jsx`:

```
import Saludo from "./Saludo";

function App() {
  return (
    <div>
      <Saludo />
      <Saludo />
    </div>
  );
}

export default App;
```

Explicación. Línea 1: `import Saludo from "./Saludo";` trae el componente del archivo `Saludo.jsx`. El `"./"` significa "en la misma carpeta". El nombre `Saludo` debe coincidir con el `export default`. Línea 3: `function App()` define el componente principal. Línea 4 a 9: el `return` devuelve un `div` que envuelve todo (recuerda la regla del elemento único raíz). Líneas 6 y 7: `<Saludo />` usa el componente dos veces, como etiqueta autocerrada. Cada uso ejecuta la función `Saludo` y muestra el saludo. Si quisieras mostrarlo diez veces, pondrías diez líneas `<Saludo />`. Así de reutilizable es desde el día uno.

Paso 3: guarda los archivos y revisa el navegador. Deberías ver "Hola, bienvenido a React" dos veces. Si no se ve nada, revisa: que el nombre del archivo sea `Saludo.jsx` exactamente, que la importación diga `"./Saludo"`, que el componente empiece con mayúscula y que `npm run dev` este corriendo sin errores en la terminal.

Experimento sugerido: cambia el texto del `h1` en `Saludo.jsx`, guarda y mira como cambian los dos saludos al mismo tiempo. Esa es la ventaja de definir una sola vez y usar muchas veces: un cambió, muchos lugares actualizados.

Ejemplo 2: tarjeta de producto con props

Ahora subimos de nivel: un componente que recibe props y muestra datos distintos cada vez. Es el ejemplo clásico de reutilización.

Paso 1: crea el archivo `src/TarjetaProducto.jsx`:

```
function TarjetaProducto({ nombre, precio, descripcion }) {
  return (
    <div className="tarjeta">
      <h2>{nombre}</h2>
      <p>{descripcion}</p>
      <p>
        Precio: ${precio}
      </p>
    </div>
  );
}

export default TarjetaProducto;
```

Explicación línea por línea. Línea 1: `function TarjetaProducto({ nombre, precio, descripción })` define el componente y desestructura tres props directamente en los parámetros: `nombre`, `precio` y `descripción`. Con solo leer esta línea sabes que datos necesita el componente. Línea 2: `return (` abre el `return`; el paréntesis permite escribir el JSX en varias líneas. Línea 3: `<div className="tarjeta">` es el elemento raíz único que envuelve todo. Usa `className` (no `class`) para asignar la clase CSS "tarjeta". Línea 4: `<h2>{nombre}</h2>` muestra la prop `nombre` dentro de un encabezado. Las llaves insertan el valor de la variable. Línea 5: `<p>{descripción}</p>` muestra la descripción. Líneas 6 a 8: un párrafo que muestra el texto "Precio: \$" seguido del valor de la prop `precio`. Observa que el signo \$ está como texto normal y `{precio}` entre llaves inserta el número. Línea 9: se cierra el `div`. Línea 13: `export default TarjetaProducto` lo hace importable.

Paso 2: usalo en `src/App.jsx` pasando props distintas:


```

import TarjetaProducto from "../TarjetaProducto";

function App() {
  return (
    <div>
      <TarjetaProducto
        nombre="Laptop Gamer"
        precio={15999}
        descripcion="Laptop potente para videojuegos y diseno."
      />
      <TarjetaProducto
        nombre="Teclado Mecanico"
        precio={1299}
        descripcion="Teclado con switches azules y luz RGB."
      />
      <TarjetaProducto
        nombre="Mouse Inalambrico"
        precio={599}
        descripcion="Mouse ergonómico con batería de larga duración."
      />
    </div>
  );
}

export default App;

```

Explicación. Cada `<TarjetaProducto ... />` es un uso del mismo componente con props distintas. `nombre="Laptop Gamer"` pasa texto con comillas. `precio={15999}` pasa un número con llaves (sin llaves llegaría como texto). `descripcion="..."` pasa el texto descriptivo. El resultado: tres tarjetas con la misma estructura pero datos distintos. Si mañana tienes cien productos, no escribes cien tarjetas a mano: usas el componente cien veces, o mejor aun, lo generas con `.map` sobre un arreglo (eso lo verás en otra unidad).

Paso 3: guarda y mira el navegador. Verás tres bloques con título, descripción y precio. Para que se vean como tarjetas de verdad, agrega estos estilos en `src/index.css` o en tu archivo de estilos:

```
.tarjeta {  
  border: 1px solid #ccc;  
  border-radius: 8px;  
  padding: 16px;  
  margin: 12px;  
  max-width: 300px;  
}
```

Experimento sugerido: agrega un cuarto producto, o cambia el precio de uno solo. Observa que cada tarjeta es independiente: cambiar las props de una no afecta a las demás. Así funcionan las props: datos de entrada por uso, sin interferencia entre usos.

Ejemplo 3: lista de usuarios con props

Este ejemplo muestra como un componente padre le pasa datos a un componente hijo, y como el hijo los muestra. Es el patrón padre -> hijo en acción.

Paso 1: crea el archivo src/TarjetaUsuario.jsx (el hijo):

```
function TarjetaUsuario({ nombre, edad, ciudad }) {  
  return (  
    <div className="tarjeta">  
      <h3>{nombre}</h3>  
      <p>Edad: {edad} anos</p>  
      <p>Ciudad: {ciudad}</p>  
    </div>  
  );  
}  
  
export default TarjetaUsuario;
```

El hijo es simple: recibe nombre, edad y ciudad, y los muestra. No sabe de donde vienen los datos ni le importa: su único trabajo es mostrar lo que recibe. Esa es la mentalidad correcta para un componente hijo.

Paso 2: crea el archivo src/ListaDeUsuarios.jsx (el padre):

```
import TarjetaUsuario from "../TarjetaUsuario";

function ListaDeUsuarios({ usuarios }) {
  return (
    <div>
      <h2>Usuarios registrados</h2>
      <TarjetaUsuario nombre="Ana" edad={28} ciudad="Ciudad de Mexico" />
      <TarjetaUsuario nombre="Luis" edad={35} ciudad="Guadalajara" />
      <TarjetaUsuario nombre="Maria" edad={22} ciudad="Monterrey" />
    </div>
  );
}

export default ListaDeUsuarios;
```

Explicación. Línea 1: el padre importa al hijo. Línea 3: el padre recibe una prop usuarios (en este ejemplo no la usamos a fondo, pero está lista para crecer). Líneas 6 a 8: el padre usa al hijo tres veces, pasándole props distintas cada vez. Fíjate en el flujo de datos: los datos están escritos en el padre (ListaDeUsuarios) y viajan hacia abajo al hijo (TarjetaUsuario) mediante props. El hijo nunca pide los datos: los recibe. Si el padre cambia un dato, el hijo muestra el cambio automáticamente.

Paso 3: muestra la lista en src/App.jsx:

```
import ListaDeUsuarios from "../ListaDeUsuarios";

function App() {
  return (
    <div>
      <h1>Mi aplicacion</h1>
      <ListaDeUsuarios />
    </div>
  );
}

export default App;
```

Aquí hay tres niveles: App usa a ListaDeUsuarios, y ListaDeUsuarios usa a TarjetaUsuario. Componentes dentro de componentes dentro de componentes. Así se construyen las aplicaciones reales: árboles de componentes donde los datos fluyen de arriba hacia abajo.

Paso 4: guarda y revisa el navegador. Verás el título "Mi aplicación", luego "Usuarios registrados" y las tres tarjetas de usuario.

Experimento sugerido: en ListaDeUsuarios, cambia la ciudad de Luis y guarda. Verás que solo su tarjeta cambia. Luego intenta modificar la prop dentro del hijo (por ejemplo, agrega ciudad = "Otra" antes del return en TarjetaUsuario) y observa que es una mala idea: funciona técnicamente pero rompe la regla de inmutabilidad. Mejor crea una variable nueva si necesitas transformar el dato.

Ejemplo 4: componente Botón reutilizable

El ejemplo más útil para la vida real: un botón configurable que puedes usar en toda tu aplicación con textos y estilos distintos.

Paso 1: crea el archivo src/Botón.jsx:

```
function Boton({ texto = "Aceptar", color = "azul", alHacerClic }) {
  const estilos = {
    azul: { backgroundColor: "blue", color: "white" },
    rojo: { backgroundColor: "red", color: "white" },
    verde: { backgroundColor: "green", color: "white" },
  };
  const estiloElegido = estilos[color] || estilos.azul;
  return (
    <button style={estiloElegido} onClick={alHacerClic}>
      {texto}
    </button>
  );
}

export default Boton;
```

Explicación línea por línea. Línea 1: el componente recibe tres props. texto y color tienen valores por defecto ("Aceptar" y "azul"): si quien usa el botón no los pasa, se usan esos. alHacerClic no tiene valor por defecto porque es opcional: si no se pasa, el botón simplemente no hace nada al darle clic. Líneas 2 a 6: un objeto estilos que relaciona cada color con su estilo en línea. Los estilos en línea en JSX son objetos: { backgroundColor: "blue", color: "white" }. Fíjate en backgroundColor en camelCase, no background-color. Línea 7: estilos[color] busca el estilo según la prop color; si el color no existe, el || usa el estilo azul como respaldo. Líneas 8 a 12: el return devuelve un <button>. style={estiloElegido} aplica el objeto de estilos con llaves (valor dinámico). onClick={alHacerClic} le pasa la función que debe ejecutarse al dar clic (sin paréntesis: se pasa la función, no se ejecuta). {texto} muestra el texto del botón. Si no se paso alHacerClic, onClick recibe undefined y no pasa nada al dar clic, sin errores.

Paso 2: usalo en src/App.jsx:

```

import Boton from "./Boton";

function App() {
  function guardar() {
    alert("Datos guardados");
  }

  function eliminar() {
    alert("Elemento eliminado");
  }

  return (
    <div>
      <h1>Panel de control</h1>
      <Boton texto="Guardar" color="verde" alHacerClic={guardar} />
      <Boton texto="Eliminar" color="rojo" alHacerClic={eliminar} />
      <Boton texto="Cancelar" color="azul" />
      <Boton />
    </div>
  );
}

export default App;

```

Explicación. Líneas 4 a 10: App define dos funciones normales de JavaScript, guardar y eliminar, que muestran una alerta. Línea 14: un botón verde que dice "Guardar" y ejecuta guardar al darle clic. La prop `alHacerClic={guardar}` pasa la función (observa: sin paréntesis). Línea 15: un botón rojo que dice "Eliminar" y ejecuta eliminar. Línea 16: un botón azul que dice "Cancelar" sin función de clic: no hace nada al presionarlo, pero no se rompe. Línea 17: `<Botón />` sin props: usa los valores por defecto y muestra un botón azul que dice "Aceptar". Cuatro botones distintos con un solo componente. Imagina una aplicación con cincuenta botones: los cambias todos desde el archivo Botón.jsx.

Paso 3: guarda y prueba en el navegador. Dale clic a los botones y verifica las alertas. El botón "Cancelar" y el botón por defecto no hacen nada al darles clic, lo cual es correcto porque no recibieron función.

Experimento sugerido: agrega un nuevo color al objeto estilos (por ejemplo, morado) y usalo en un quinto botón. Luego cambia el valor por defecto de texto a "Continuar" y observa como el botón sin props ahora dice "Continuar". Así se siente el poder de los valores por defecto.

Con estos cuatro ejemplos ya practicaste lo esencial: definir componentes, usarlos como etiquetas, pasarles props de todo tipo (texto, números, funciones), usar valores por defecto,

anidar componentes padre e hijo y manejar children si lo deseas. El siguiente paso natural es intentarlo tu solo con los ejercicios de la sección de práctica.

Práctica

Llegó la hora de ensuciarse las manos. Aquí tienes cinco ejercicios ordenados de fácil a intermedio. La regla de oro: intenta resolver CADA ejercicio por tu cuenta ANTES de ver la solución. Lee el enunciado, abre tu editor, escribe el código, pruebalo en el navegador y solo cuando ya lo intentaste (aunque no te salga perfecto) revisa la solución. Equivocarte es parte del proceso: cada error que corriges tu solo te hace más fuerte. Las soluciones están completas y explicadas paso a paso para que compares y aprendas.

Para todos los ejercicios usa tu proyecto de Vite con npm run dev corriendo. Crea un archivo por ejercicio en la carpeta src y muéstralo desde App.jsx, como hiciste en los ejemplos.

Ejercicio 1: componente Perfil

Enunciado: crea un componente llamado Perfil que muestre información de una persona: su nombre, su edad y su ocupación. El componente debe recibir esos tres datos como props (nombre, edad, ocupación) y mostrarlos con buena presentación: el nombre en un h2, la edad y la ocupación en párrafos. Luego usalo en App.jsx tres veces con tres personas distintas.

Pistas: recuerda la estructura básica `function Perfil({ nombre, edad, ocupación }) { return (...); }`, no olvides el `export default`, y en App.jsx importa el componente y pasale las props. La edad es un número, así que usa llaves.

Intentalo ahora antes de seguir leyendo. En serio: escribe el código tu primero.

Solución:

Archivo src/Perfil.jsx:

```
function Perfil({ nombre, edad, ocupacion }) {
  return (
    <div className="tarjeta">
      <h2>{nombre}</h2>
      <p>Edad: {edad} años</p>
      <p>Ocupacion: {ocupacion}</p>
    </div>
  );
}

export default Perfil;
```

Explicación paso a paso. Línea 1: definimos el componente Perfil y desestructuramos las tres props que necesita: nombre, edad y ocupación. Los nombres de las props los elegimos nosotros; lo importante es usar los mismos nombres al pasar los datos. Línea 2: abrimos el return con paréntesis para escribir el JSX en varias líneas. Línea 3: el div con className="tarjeta" es el elemento raíz único que envuelve todo (puedes reutilizar los estilos .tarjeta de los ejemplos). Línea 4: <h2>{nombre}</h2> muestra el nombre en grande. Las llaves insertan el valor de la prop. Línea 5: <p>Edad: {edad} años</p> combina texto fijo ("Edad: " y " años") con la prop edad entre llaves. Línea 6: lo mismo con la ocupación. Línea 10: export default Perfil para poder importarlo.

Archivo src/App.jsx:

```
import Perfil from "../Perfil";

function App() {
  return (
    <div>
      <h1>Directorio de personas</h1>
      <Perfil nombre="Carlos" edad={30} ocupacion="Ingeniero" />
      <Perfil nombre="Sofia" edad={25} ocupacion="Disenadora" />
      <Perfil nombre="Diego" edad={41} ocupacion="Maestro" />
    </div>
  );
}

export default App;
```

Explicación. Importamos Perfil y lo usamos tres veces con datos distintos. Observa que edad={30} usa llaves porque es número, mientras que nombre="Carlos" usa comillas porque es texto. El resultado: tres tarjetas con tres personas distintas, todas generadas por el mismo

componente. Si tu solución se ve así, vas por excelente camino. Si usaste `props.nombre` sin desestructurar, también es válido: la desestructuración es la forma preferida por los profesionales, pero ambas funcionan.

Ejercicio 2: componente TarjetaProducto con props y precio con descuento

Enunciado: crea un componente `TarjetaProducto` que reciba las props `nombre`, `precio` y `enOferta` (un booleano). Debe mostrar el nombre en un `h2` y el precio en un párrafo. Si la prop `enOferta` es `true`, debe mostrar además un párrafo que diga "En oferta!" y el precio debe verse con un descuento del 20 por ciento ya aplicado. Si `enOferta` es `false` o no se pasa, muestra el precio normal sin el mensaje. Usalo en `App.jsx` con al menos tres productos: uno en oferta, uno sin oferta y uno sin pasar la prop `enOferta` (para comprobar el valor por defecto).

Pistas: dale a `enOferta` el valor por defecto `false` en la desestructuración: `{ nombre, precio, enOferta = false }`. Para el precio con descuento calcula `precio * 0.8` arriba del `return`. Para mostrar el mensaje solo cuando hay oferta usa un ternario o el operador `&&` dentro del JSX.

Intentalo tu primero. Piensa donde va la lógica (arriba del `return`) y que va en el JSX (solo mostrar).

Solución:

Archivo `src/TarjetaProducto.jsx`:

```
function TarjetaProducto({ nombre, precio, enOferta = false }) {
  const precioFinal = enOferta ? precio * 0.8 : precio;
  return (
    <div className="tarjeta">
      <h2>{nombre}</h2>
      <p>Precio: ${precioFinal}</p>
      {enOferta && <p>En oferta!</p>}
    </div>
  );
}

export default TarjetaProducto;
```

Explicación paso a paso. Línea 1: desestructuramos las tres props y le damos a `enOferta` el valor por defecto `false`. Así, si alguien usa `<TarjetaProducto nombre="X" precio={100} />` sin pasar `enOferta`, el componente no se rompe: asume `false`. Línea 2: la lógica va arriba del `return`. Con un ternario calculamos `precioFinal`: si `enOferta` es `true`, aplicamos el 20 por ciento de descuento (multiplicar por 0.8); si no, el precio queda igual. Esta variable es local: no

mutamos la prop precio, creamos un valor nuevo. Buena práctica. Línea 6: mostramos el precio final con el signo \$ como texto y {precioFinal} entre llaves. Línea 7: {enOferta && <p>En oferta!</p>}} es renderizado condicional con el operador &&. Funciona así: si enOferta es true, se muestra el párrafo; si es false, no se muestra nada. Es un patrón muy usado en React para mostrar cosas solo cuando se cumple una condición.

Archivo src/App.jsx:

```
import TarjetaProducto from "../TarjetaProducto";

function App() {
  return (
    <div>
      <h1>Tienda</h1>
      <TarjetaProducto nombre="Audifonos" precio={2000} enOferta={true} />
      <TarjetaProducto nombre="Cargador" precio={500} enOferta={false} />
      <TarjetaProducto nombre="Funda" precio={300} />
    </div>
  );
}

export default App;
```

Explicación. El primer producto pasa enOferta={true} con llaves porque es booleano (si escribieras enOferta="true" llegaría como texto, y el texto "false" también sería verdadero porque cualquier texto no vacío es verdadero en JavaScript: cuidado con eso). El segundo pasa false explícito. El tercero no pasa la prop y usa el valor por defecto false. Resultado: los audifonos muestran precio 1600 (2000 * 0.8) y el mensaje "En oferta!"; los demás muestran su precio normal sin mensaje.

Ejercicio 3: componente ListaDeTareas estatica con props

Enunciado: crea dos componentes. Primero, Tarea, que recibe las props título y completada (booleano, con valor por defecto false) y muestra el título en un párrafo. Si completada es true, el texto debe verse tachado (usa un estilo en línea con textDecoration). Segundo, ListaDeTareas, que usa el componente Tarea cinco veces con títulos y estados distintos, con un h2 que diga "Mis tareas". Muestralo todo desde App.jsx.

Pistas: el estilo en línea para tachar es style={{ textDecoration: completada ? "line-through" : "none" }}. Recuerda las dobles llaves: las de afuera son de JSX y las de adentro son del objeto. ListaDeTareas es el padre y Tarea es el hijo: los datos fluyen de arriba hacia abajo.

Hazlo tu antes de ver la solución. Este ejercicio combina props booleanas, valores por defecto, estilos en línea y composición padre-hijo.

Solución:

Archivo src/Tarea.jsx:

```
function Tarea({ titulo, completada = false }) {  
  const estilo = {  
    textDecoration: completada ? "line-through" : "none",  
  };  
  return <p style={estilo}>{titulo}</p>;  
}  
  
export default Tarea;
```

Explicación. Línea 1: el componente recibe título y completada (por defecto false). Líneas 2 a 4: creamos el objeto de estilo arriba del return. textDecoration vale "line-through" si la tarea esta completada y "none" si no. Fíjate en textDecoration en camelCase, como exige JSX. Línea 5: devolvemos un parrafo con style={estilo} (llaves porque es un valor dinámico) y el título entre llaves. Es un componente pequeno con una sola responsabilidad: mostrar una tarea. Así deben ser los componentes.

Archivo src/ListaDeTareas.jsx:

```
import Tarea from "../Tarea";  
  
function ListaDeTareas() {  
  return (  
    <div>  
      <h2>Mis tareas</h2>  
      <Tarea titulo="Aprender componentes" completada={true} />  
      <Tarea titulo="Practicar JSX" completada={true} />  
      <Tarea titulo="Dominar las props" completada={false} />  
      <Tarea titulo="Hacer los ejercicios" />  
      <Tarea titulo="Construir mi primer proyecto" />  
    </div>  
  );  
}  
  
export default ListaDeTareas;
```

Explicación. El padre importa al hijo y lo usa cinco veces. Las dos primeras tareas están completadas (se verán tachadas), la tercera pasa false explícito, y las dos últimas no pasan la prop, así que usan el valor por defecto false. Observa el flujo unidireccional: los datos (título y completada) están en el padre y viajan al hijo mediante props. El hijo solo muestra.

Archivo src/App.jsx:

```
import ListaDeTareas from "../ListaDeTareas";

function App() {
  return (
    <div>
      <h1>Mi organizador</h1>
      <ListaDeTareas />
    </div>
  );
}

export default App;
```

Tres niveles de componentes: App -> ListaDeTareas -> Tarea. Cada nivel tiene su responsabilidad: App es la página, ListaDeTareas agrupa las tareas, Tarea muestra una tarea. Así se organiza el código profesional.

Ejercicio 4: tarjeta con children

Enunciado: crea un componente llamado Panel que reciba la prop children y una prop título. El Panel debe mostrar el título en un h3 y luego mostrar children dentro de un div con className="panel". Luego, en App.jsx, usa Panel dos veces: una vez con un párrafo y un botón adentro, y otra vez con una lista de tres elementos adentro. Agrega estilos para la clase .panel en tu CSS (borde, fondo claro, padding).

Pistas: la sintaxis es function Panel({ título, children }). El contenido entre <Panel> y </Panel> llega como children. No olvides que al usarlo con contenido debes usar etiqueta de apertura y cierre, no la autocerrada.

Intentalo primero. Este ejercicio te enseña el patrón de componentes contenedores, muy usado en la industria para modales, tarjetas y paneles.

Solución:

Archivo src/Panel.jsx:

```
function Panel({ titulo, children }) {
  return (
    <div className="panel">
      <h3>{titulo}</h3>
      <div>{children}</div>
    </div>
  );
}

export default Panel;
```

Explicación. Línea 1: desestructuramos título y children. children es la prop especial que contiene lo que el usuario del componente ponga entre las etiquetas. Línea 3: el div con clase "panel" es el marco del contenedor. Línea 4: el título siempre se muestra arriba. Línea 5: {children} coloca el contenido que paso quien uso el componente. El Panel no sabe ni le importa que contenido es: puede ser texto, botones, listas o hasta otros componentes. Esa es la gracia de children: el contenedor define el marco y el usuario define el contenido.

Archivo src/App.jsx:

```
import Panel from "../Panel";

function App() {
  return (
    <div>
      <h1>Paneles</h1>
      <Panel titulo="Aviso importante">
        <p>Recuerda guardar tu trabajo antes de salir.</p>
        <button>Entendido</button>
      </Panel>
      <Panel titulo="Lista de compras">
        <ul>
          <li>Tortillas</li>
          <li>Queso</li>
          <li>Salsa</li>
        </ul>
      </Panel>
    </div>
  );
}

export default App;
```

Explicación. Usamos Panel dos veces con contenidos totalmente distintos: el primero lleva un párrafo y un botón; el segundo lleva una lista. El mismo componente marco sirve para ambos casos gracias a children. Fíjate en que aquí NO usamos la forma autocerrada `<Panel />`: como hay contenido adentro, usamos `<Panel>...</Panel>`. Si lo escribieras autocerrado, el contenido quedaría fuera y children llegaría vacío.

Estilos sugeridos en tu CSS:

```
.panel {
  border: 2px solid #333;
  border-radius: 8px;
  background-color: #f5f5f5;
  padding: 16px;
  margin: 12px;
  max-width: 400px;
}
```

Ejercicio 5: galeria de productos con precio formateado (reto integrador)

Enunciado: este ejercicio junta todo lo de la unidad. Crea un componente Producto que reciba nombre, precio y moneda (con valor por defecto "MXN"). Debe mostrar el nombre en un h2 y el precio formateado como "\$1,599.00 MXN" (usa `precio.toLocaleString()` para los separadores de miles). Luego crea un componente Galeria que muestre un h1 con "Catálogo" y use Producto cuatro veces con datos distintos, uno de ellos sin pasar la moneda para probar el valor por defecto. Muestralo desde App.jsx.

Pistas: `(1599).toLocaleString()` devuelve "1,599". Puedes combinar texto y llaves: `${precio.toLocaleString()}.00 {moneda}`. La lógica del formateo va arriba del return en una variable, por ejemplo `const precioTexto = ...`. Este ejercicio practica: componentes, props, valores por defecto, llaves con expresiones y composición.

Es el reto final: hazlo completo tu solo antes de mirar. Si te atorras, repasa el Ejemplo 2.

Solución:

Archivo `src/Producto.jsx`:

```
function Producto({ nombre, precio, moneda = "MXN" }) {
  const precioTexto = "$" + precio.toLocaleString() + ".00 " + moneda;
  return (
    <div className="tarjeta">
      <h2>{nombre}</h2>
      <p>{precioTexto}</p>
    </div>
  );
}

export default Producto;
```

Explicación paso a paso. Línea 1: el componente recibe nombre, precio y moneda con valor por defecto "MXN". Línea 2: la lógica arriba del return. `precio.toLocaleString()` convierte el número a texto con separadores de miles según tu region: 1599 se vuelve "1,599". Luego concatenamos el signo \$, el ".00" de los centavos y la moneda. Creamos una variable nueva (`precioTexto`) en lugar de mutar la prop `precio`: buena práctica. Línea 6: mostramos el texto ya formateado entre llaves. El JSX queda limpio y legible porque el calculo se hizo arriba.

Archivo `src/Galeria.jsx`:

```
import Producto from "../Producto";

function Galeria() {
  return (
    <div>
      <h1>Catalogo</h1>
      <Producto nombre="Laptop" precio={15999} moneda="MXN" />
      <Producto nombre="Teclado" precio={1299} moneda="MXN" />
      <Producto nombre="Monitor" precio={4500} />
      <Producto nombre="Silla gamer" precio={8999} moneda="USD" />
    </div>
  );
}

export default Galeria;
```

Explicación. `Galeria` es el padre que compone cuatro productos. El tercero no pasa moneda y usa el valor por defecto "MXN": verás "\$4,500.00 MXN". El cuarto pasa "USD" para mostrar que la misma prop acepta valores distintos. Resultado: un catálogo de cuatro productos con precios bien formateados, todo generado por dos componentes pequenos y reutilizables.

Archivo `src/App.jsx`:

```
import Galeria from "../Galeria";

function App() {
  return (
    <div>
      <Galeria />
    </div>
  );
}

export default App;
```

Si completaste los cinco ejercicios (aunque hayas consultado las soluciones después de intentarlo), ya dominas lo esencial de la unidad: definir componentes, usar JSX con expresiones, pasar props de todos los tipos, usar valores por defecto, componer padres e hijos y aprovechar children. Lo que sigue en el curso (estado, eventos, efectos) se construye sobre esta base, así que cada minuto que invertiste aquí se multiplica después.

Resumen de la unidad

Llegamos al final de la Unidad 2. Vamos a repasar los puntos clave, como si fuera un acordeon para tu examen mental. Si puedes explicar cada uno de estos puntos con tus propias palabras, la unidad esta dominada.

Componentes: un componente es una función de JavaScript que devuelve JSX. Es la pieza básica de construcción en React: defines el componente una vez y lo usas cuantas veces quieras, como si fuera una etiqueta de HTML (<Saludo />). Los componentes se combinan entre si (composición): componentes pequenos forman componentes más grandes, y esos forman páginas, y las páginas forman la aplicación completa. Analogía para no olvidar: los componentes son como piezas de LEGO o como recetas de cocina: se definen una vez y se reutilizan muchas veces.

Reglas de los componentes, las tres innegociables: primera, el nombre empieza con mayúscula (PascalCase como TarjetaProducto), porque así React distingue tus componentes de las etiquetas HTML normales. Segunda, el return debe devolver un solo elemento raíz: envuelve todo en un div o usa un fragmento <>...</> si no quieres etiquetas extra en el HTML. Tercera, cada componente vive en su propio archivo con el mismo nombre (Saludo.jsx para el componente Saludo) y se comparte con export default.

JSX: es una sintaxis que mezcla HTML con JavaScript y que las herramientas (como Vite) compilan a JavaScript normal, a llamadas a `React.createElement`. No es magia: es una forma cómoda de describir la interfaz. Las llaves `{}` son la característica estrella: todo lo que pongas entre llaves se ejecuta como JavaScript (variables, operaciones, ternarios, llamadas a funciones, métodos). Recuerda que entre llaves solo van expresiones, no sentencias: la lógica compleja va arriba del `return` y en el JSX solo muestras resultados. Diferencias con HTML que debes tener grabadas: `className` en lugar de `class`, `htmlFor` en lugar de `for`, todas las etiquetas deben cerrarse (`<img />`, `<br />`), los estilos en línea son objetos con propiedades en camelCase (`style={{ color: "red" }}`), y los comentarios dentro de JSX se escriben `{/* así */}`.

Props: son los datos que recibe un componente, equivalentes a los parámetros de una función. Se pasan como atributos (`<Saludo nombre="Ana" />`) y el componente las recibe en su primer parámetro, normalmente desestructuradas: `function Saludo({ nombre })`. Las props pueden ser de cualquier tipo: texto con comillas, y números, booleanos, arreglos, objetos, variables y funciones con llaves. Los valores por defecto en la desestructuración (`{ nombre = "visitante" }`) hacen tus componentes robustos cuando una prop no llega. `children` es la prop especial que contiene lo que pongas entre la etiqueta de apertura y la de cierre, ideal para componentes contenedores como tarjetas, paneles y modales. Dos leyes de las props: fluyen en una sola dirección (del padre al hijo, nunca al revés) y son inmutables (el hijo las lee pero jamás las modifica; si necesita transformar un dato, crea una variable nueva).

Errores de principiante que ya sabes evitar: olvidar la mayúscula del componente, devolver varios elementos sin envolver, mutar las props, confundir comillas con llaves al pasar valores, usar `class` en vez de `className`, dejar etiquetas sin cerrar, meter sentencias `if` o `for` dentro de las llaves del JSX, y escribir mal el nombre de una prop (mayúsculas y minúsculas importan). Cada vez que algo no se muestre o salga `undefined`, revisa está lista antes de entrar en pánico.

Buenas prácticas profesionales: componentes pequeños con una sola responsabilidad; un componente por archivo; nombres descriptivos que digan que hace el componente; desestructurar las props en los parámetros; valores por defecto para props opcionales; convertir código JSX repetido en un componente con props; mantener la lógica compleja fuera del JSX; y usar `children` para los contenedores. Estas prácticas son las que separan el código de estudiante del código de empresa.

Que sigue: en la Unidad 3 entrarás al mundo del estado y los eventos: como los componentes recuerdan información y responden a lo que hace el usuario (clics, escritura en formularios). Todo lo que aprendiste aquí (componentes, JSX, props) es la base sobre la que se construye lo que viene. Práctica los ejercicios de esta unidad hasta que te salgan sin pensar, y la Unidad 3 se te hará mucho más fácil. Nos vemos ahí.

Una última idea para que te la lleves grabada: React premia a quien piensa en piezas. Cada vez que mires una página web de aquí en adelante, intenta verla con ojos de componentes: ese encabezado es un componente, esa tarjeta de producto es otro, ese botón se repite diez veces así que seguro es un componente con props. Esa forma de mirar las interfaces es la habilidad real que te deja esta unidad, más allá de la sintaxis. La sintaxis se consulta en dos minutos; la mentalidad de componentes es lo que te convierte en desarrollador React. Sigue practicando, rompe tu código a proposito, lee los errores con calma y construye cosas pequeñas todos los días. Así se aprende de verdad.

UNIDAD 3: Listas, condicionales y estado con useState

Hasta ahora has aprendido a crear componentes, a pasarles props y a escribir JSX. Eso ya es un gran avance, porque dominas los cimientos de React. Pero hay algo que debes notar: todo lo que has construido hasta este momento es estático. Un componente que recibe props y pinta un título siempre pinta lo mismo mientras las props no cambien. Es como una fotografía: bonita, útil, pero congelada en el tiempo.

Las aplicaciones reales no son fotografías. Son películas. Piensa en cualquier app que usas todos los días: Instagram muestra una lista de publicaciones que crece cada vez que alguien publica algo nuevo. WhatsApp muestra tus mensajes y los actualiza en tiempo real cuando llega uno nuevo. Un carrito de compras cambia el total cada vez que agregas o quitas un producto. Nada de eso es estático. Todo cambia, reacciona y se mueve.

Esta unidad te enseña exactamente eso: como darle vida a tus interfaces. Y la vida en React llega con tres herramientas que trabajan juntas como un trío inseparable. La primera es el renderizado de listas, que te permite tomar un arreglo de datos y convertirlo en elementos de la pantalla de forma automática. La segunda son los condicionales en JSX, que te permiten mostrar una cosa u otra dependiendo de una condición. Y la tercera es useState, el hook que guarda información dentro de un componente y hace que React vuelva a pintar la pantalla cada vez que esa información cambia.

Fíjate bien en por qué estas tres herramientas van juntas en la misma unidad. Casi cualquier cosa dinámica que quieras construir las combina. Una lista de tareas necesita un arreglo de datos (listas), necesita mostrar "no hay tareas" cuando la lista está vacía (condicionales) y necesita actualizarse cuando agregas una tarea nueva (estado con useState). Un menú de navegación necesita una lista de enlaces, necesita resaltar el enlace activo con un condicional y necesita guardar cuál sección está seleccionada en el estado. Un formulario necesita listas para mostrar errores, condicionales para mostrar mensajes de éxito y estado para guardar lo que el usuario escribe. Una vez que domines este trío, podrás construir el 80 por ciento de las interfaces interactivas que existen.

Hay algo importante que debes entender desde ahora: estas tres herramientas no son trucos raros ni sintaxis mágica. Son JavaScript puro aplicado a la interfaz. Renderizar listas es el método `map` de los arreglos, que ya conoces. Los condicionales son el operador ternario y el operador `&&`, que también ya conoces. Y useState, aunque es nuevo, es una función que devuelve un arreglo que desestructuras, algo que ya practicaste en unidades anteriores. React

no inventa un lenguaje nuevo para esto: toma el JavaScript que ya sabes y lo pone a trabajar pintando pantallas. Esa es la belleza del asunto.

En esta unidad vamos a ir paso a paso. Primero verás la explicación detallada de cada herramienta, con analogías, errores comunes y buenas prácticas. Después trabajaremos con ejemplos completos explicados línea por línea, que podrás copiar y probar en tu proyecto de Vite. Luego tendrás ejercicios de práctica con soluciones completas, para que te ensucies las manos de verdad. Y cerraremos con un resumen que amarre todo lo aprendido.

Un consejo antes de empezar: no leas esta unidad como si fuera una novela. Abre tu editor, crea un proyecto de React con Vite si todavía no lo tienes y escribe cada ejemplo con tus propias manos. Los errores que cometas tecleando son parte del aprendizaje. Si algo no te sale, relea la sección, compara tu código con el ejemplo y busca la diferencia. La programación se aprende programando, no leyendo. Así que manos a la obra: es hora de que tus interfaces dejen de ser fotografías y empiecen a ser películas.

Antes de entrar en materia, dejame pintarte el mapa completo de a donde vamos. Esta unidad tiene tres paradas, y cada una construye sobre la anterior. En la primera parada dominaras el renderizado de listas: tomaras arreglos de datos, del tipo que sea, y los convertirás en elementos de pantalla con `map`, entendiendo por fin para que sirve la famosa `prop key` y por qué React insiste tanto en ella. En la segunda parada dominaras los condicionales en JSX: el ternario, el operador `&&`, el `if` con variable y el `switch`, y sabras exactamente cuando usar cada uno sin dudar. En la tercera parada llegara `useState`, el hook que cambia las reglas del juego: tus componentes ganaran memoria y podran reaccionar a lo que hace el usuario.

Y aquí viene lo mejor: al final de la unidad no solo sabras cada herramienta por separado, sabras combinarlas. La lista de tareas del ejemplo 4 y la lista de invitados del ejercicio 5 son pruebas de que el trio funciona como un equipo: el estado guarda los datos, los condicionales deciden que mostrar y `map` convierte los datos en pantalla. Ese patrón de tres pasos aparece en carritos de compra, en muros de redes sociales, en bandejas de mensajes y en paneles de administración. Lo que aprendas aquí no es teoría de libro: es el pan de cada día del desarrollo con React.

Una última cosa antes de empezar: esta unidad es más larga que las anteriores a propósito. Hay mucho que desempacar porque estas tres herramientas tienen detalles que solo se entienden con calma: por qué React necesita keys, por qué el estado no se actualiza de inmediato, por qué no debes mutar arreglos. No te brinques las analogías ni los errores comunes pensando que son relleno: son exactamente los puntos donde la mayoría se atora, y leerlos ahora te ahorra horas de frustración después. Lee a tu ritmo, prueba cada ejemplo y cuando termines los ejercicios vas a sentir que React por fin "hace clic" en tu cabeza. Esa sensación es la meta de esta unidad.

Explicación detallada

Renderizar listas: de datos a pantalla con map

Imagina que tienes una lista de compras escrita en un papel: manzanas, pan, leche, huevos. Ahora imagina que tienes que escribir esa lista en un pizarrón, un renglón por producto. Lo lógico sería tomar cada producto del papel y copiarlo al pizarrón, uno por uno. Eso es exactamente lo que hace el método `map` en React: toma cada elemento de un arreglo y lo transforma en un elemento de la interfaz.

En JavaScript, `map` es un método de los arreglos que recorre cada elemento, le aplica una función y devuelve un arreglo nuevo con los resultados. Mira este ejemplo sin React:

```
const frutas = ["manzana", "pera", "uva"];
const mayusculas = frutas.map(function(fruta) {
  return fruta.toUpperCase();
});
// mayusculas es ["MANZANA", "PERA", "UVA"]
```

`map` no modifica el arreglo original: crea uno nuevo. Esa característica es importante en React, porque a React le gustan las cosas inmutables, como verás más adelante con `useState`.

Ahora, en React, en lugar de transformar texto a mayúsculas, transformas datos a elementos JSX. Mira:

```
const frutas = ["manzana", "pera", "uva"];

function ListaFrutas() {
  return (
    <ul>
      {frutas.map(function(fruta) {
        return <li>{fruta}</li>;
      })}
    </ul>
  );
}
```

Fíjate en lo que pasa aquí. Dentro del JSX usamos llaves { } para "escapar" a JavaScript, como aprendiste en la unidad anterior. Dentro de esas llaves llamamos a frutas.map, que recorre el arreglo y por cada fruta devuelve un elemento . El resultado es un arreglo de elementos , y React sabe como pintar un arreglo de elementos: los pone uno tras otro dentro del . El resultado en pantalla es una lista con tres elementos: manzana, pera, uva.

La versión corta con función flecha, que es la que verás en casi todo el código real, se ve así:

```
{frutas.map((fruta) => <li>{fruta}</li>)}
```

O, si prefieres ser explícito con el return:

```
{frutas.map((fruta) => {
  return <li>{fruta}</li>;
})}
```

Las tres formas hacen lo mismo. Usa la que te resulte más legible. En código profesional verás más la primera, la de flecha sin llaves, porque es compacta.

La prop key: el identificador que React necesita

El ejemplo anterior funciona, pero React te mostrara una advertencia en la consola: cada elemento de una lista necesita una prop especial llamada key. Vamos a entender por qué, porque no es un capricho: es una de las ideas más importantes de React.

Imagina que eres maestro y tienes una lista de asistencia con 30 alumnos. Un día llega un alumno nuevo y se sienta en medio del salón. Si tu lista no tiene números de folio y solo dice los nombres en orden, te cuesta trabajo saber exactamente donde insertarlo y si alguien faltó o solo cambió de lugar. Pero si cada alumno tiene un número de folio único, sabes al instante quien es quien, quien llegó nuevo y quien se fue, sin importar el orden en que estén sentados.

React enfrenta el mismo problema. Cuando tu componente se vuelve a pintar, React compara la lista anterior de elementos con la lista nueva para decidir que cambiar en la pantalla. Este proceso se llama reconciliación. Sin una key, React solo puede comparar por posición: "el primer elemento de antes contra el primer elemento de ahora". Si agregaste un elemento al principio de la lista, React pensará que todos los elementos cambiaron de contenido, y hará trabajo de más. Con una key única, React sabe exactamente cual elemento es cual, aunque cambie de posición, y solo actualiza lo que de verdad cambió.

La key se pasa como una prop especial:

```
{frutas.map((fruta) => <li key={fruta}>{fruta}</li>)}
```

Aquí usamos el nombre de la fruta como key porque en este ejemplo los nombres no se repiten. Pero ojo: la key debe ser única entre los elementos de la misma lista y, de preferencia, estable, es decir, que no cambie entre pintados. Los mejores candidatos son los identificadores que vienen de tus datos: un id de base de datos, un código de producto, un nombre de usuario.

Cuando trabajas con listas de objetos, que es lo más común en aplicaciones reales, cada objeto suele traer su propio id:

```
const productos = [
  { id: 1, nombre: "Laptop", precio: 15000 },
  { id: 2, nombre: "Mouse", precio: 300 },
  { id: 3, nombre: "Teclado", precio: 800 }
];

function ListaProductos() {
  return (
    <ul>
      {productos.map((producto) => (
        <li key={producto.id}>
          {producto.nombre} - ${producto.precio}
        </li>
      ))}
    </ul>
  );
}
```

Fíjate en los paréntesis alrededor del JSX multilinea: cuando el return de la función flecha abarca varias líneas, se envuelve entre paréntesis para que JavaScript no se confunda. Es un patrón que verás muchísimo.

El problema de usar el índice como key

Cuando no tienes un id a la mano, es tentador usar el índice del arreglo como key, porque map te lo regala como segundo parámetro:

```
{frutas.map((fruta, indice) => <li key={indice}>{fruta}</li>)}
```

Esto elimina la advertencia de la consola, pero puede causar errores sutiles y difíciles de rastrear. Te explicó por qué con un ejemplo concreto.

Supon que tienes una lista de tres tareas con checkboxes: "comprar leche", "pasear al perro", "estudiar React". Marcas la primera como completada. Ahora agregas una tarea nueva al principio de la lista: "pagar la luz". Con el índice como key, React ve lo siguiente: antes tenía keys 0, 1, 2 y ahora sigue teniendo keys 0, 1, 2. Como las keys no cambiaron, React reutiliza los mismos elementos de la pantalla y solo actualiza el texto. El problema es que el checkbox marcado estaba asociado al elemento con key 0, que antes era "comprar leche" y ahora es "pagar la luz". Resultado: la palomita se movio a la tarea equivocada. El usuario marco una tarea y la marca aparecio en otra.

Con keys estables basadas en id, esto no pasa: React sabe que el elemento con id 5 sigue siendo el mismo aunque ahora este en otra posición, y la palomita se queda con su tarea.

La regla práctica es esta: puedes usar el índice como key solo si la lista es estática, es decir, si nunca cambia de orden, nunca se le agregan elementos ni se le quitan. Una lista de días de la semana o de meses del año, por ejemplo. En cualquier otro caso, usa un identificador único y estable de tus datos. Si tus datos no traen id, considera generarlo cuando crees los elementos, por ejemplo con un contador o con `crypto.randomUUID`.

Condicionales en JSX: mostrar una cosa u otra

Las interfaces reales están llenas de decisiones: si el usuario está logueado muestra su nombre, si no muestra un botón de entrar. Si el carrito está vacío muestra un mensaje, si tiene productos muestra la lista. Si está cargando muestra un spinner, si ya terminó muestra los datos. Todo eso son condicionales, y en JSX tienes varias formas de escribirlos.

Recuerda la regla de oro: dentro del JSX, entre llaves `{ }`, solo puedes poner expresiones, no sentencias. Una expresión es algo que produce un valor: una variable, una llamada a función, un operador ternario. Una sentencia como `if` o `for` no produce un valor directamente, así que no puede ir dentro de las llaves. Por eso en JSX usamos operadores que si son expresiones.

El operador ternario

El ternario es el condicional más usado en JSX. Su forma es: `condición ? valorSiVerdadero : valorSiFalso`.

```
function Saludo({ nombre, logueado }) {  
  return (  
    <div>  
      {logueado ? <p>Hola, {nombre}</p> : <p>Hola, visitante</p>}  
    </div>  
  );  
}
```

Si `logueado` es `true`, se pinta "Hola, Ana". Si es `false`, se pinta "Hola, visitante". Simple y directo. El ternario es perfecto cuando tienes dos alternativas claras: una cosa u otra.

También puedes usar el ternario para decidir clases CSS, atributos o cualquier valor:


```
<button className={activo ? "btn-activo" : "btn-inactivo"}>
  {activo ? "Apagar" : "Encender"}
</button>
```

Un error comun es anidar ternarios dentro de ternarios para cubrir tres o más casos. Tecnicamente funciona, pero se vuelve ilegible rapidísimo:

```
// Evita esto: es difícil de leer
{rol === "admin" ? <PanelAdmin /> : rol === "editor" ? <PanelEditor /> :
<PanelLector />}
```

Para tres o más casos hay mejores herramientas, como verás en la sección del switch.

El operador && para mostrar u ocultar

Muchas veces no necesitas dos alternativas, sino mostrar algo solo si una condición se cumple, y no mostrar nada si no. Para eso sirve el operador &&:

```
{hayMensajes && <p>Tienes mensajes nuevos</p>}
```

Cómo funciona: JavaScript evalúa de izquierda a derecha. Si hayMensajes es false, la expresión completa es false y React no pinta nada cuando el resultado es false, null o undefined. Si hayMensajes es true, JavaScript sigue evaluando y el resultado es el elemento <p>, que React sí pinta.

Este patrón es ideal para avisos, insignias, mensajes de error y cualquier cosa que aparece o desaparece:

```
{error && <p className="error">{error}</p>}
{usuario.esAdmin && <button>Borrar todo</button>}
```

Pero cuidado con una trampa famosa: si el valor de la izquierda puede ser un número, como 0, el && te puede jugar una mala pasada. Mira:

```
{mensajes.length && <p>Tienes {mensajes.length} mensajes</p>}
```

Si mensajes.length es 0, la expresión completa vale 0, y React sí pinta el número 0 en la pantalla. Verías un feo "0" flotando en tu interfaz. La solución es asegurarte de que la condición sea un booleano de verdad:

```
{mensajes.length > 0 && <p>Tienes {mensajes.length} mensajes</p>}
```

Con `> 0` la condición es `true` o `false`, nunca un número. Acostúmbrate a escribir comparaciones explícitas cuando uses `&&` y te ahorrarás este dolor de cabeza.

If fuera del return con una variable

Cuando la lógica condicional se complica, la mejor práctica es sacarla del JSX y usar un `if` normal de JavaScript antes del `return`, guardando el resultado en una variable:

```
function Panel({ rol }) {  
  let contenido;  
  
  if (rol === "admin") {  
    contenido = <PanelAdmin />;  
  } else if (rol === "editor") {  
    contenido = <PanelEditor />;  
  } else {  
    contenido = <PanelLector />;  
  }  
  
  return <div>{contenido}</div>;  
}
```

Esto es mucho más legible que un ternario anidado. La regla general: si tu condicional cabe en una línea, usalo dentro del JSX; si necesita varias ramas o lógica extra, sacalo a una variable con `if/else` antes del `return`. Tu yo del futuro te lo agradecerá cuando tengas que modificar ese código.

También puedes hacer `returns` tempranos, es decir, regresar antes de tiempo según la condición:

```
function DatosUsuario({ usuario, cargando }) {
  if (cargando) {
    return <p>Cargando...</p>;
  }

  if (!usuario) {
    return <p>No se encontro el usuario.</p>;
  }

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

Los returns tempranos son elegantes para manejar estados de carga y error: primero descartas los casos especiales y al final pones el caso normal, sin anidamientos.

Switch para múltiples casos

Cuando tienes una variable que puede tomar varios valores conocidos, como el estado de un semáforo o el paso de un asistente, el switch es la herramienta más clara:

```
function Semaforo({ color }) {
  let mensaje;

  switch (color) {
    case "rojo":
      mensaje = "Alto total";
      break;
    case "amarillo":
      mensaje = "Precaucion";
      break;
    case "verde":
      mensaje = "Avance";
      break;
    default:
      mensaje = "Color desconocido";
  }

  return <p>{mensaje}</p>;
}
```

El switch deja clarísimo que estás eligiendo entre varias opciones de la misma variable. No olvides el break en cada caso ni el default para lo inesperado.

Cuando usar cada condicional

Te dejo una guía rápida para decidir:

- Ternario (? :): dos alternativas, una u otra. "Si esta logueado muestra esto, si no muestra aquello".
- && : mostrar algo o nada. "Si hay error muestra el mensaje".
- if/else con variable antes del return: lógica con varias ramas o que necesita calculos intermedios.
- Returns tempranos: descartar casos especiales como carga o error antes del contenido principal.
- switch: una variable con varios valores posibles conocidos.

Con la práctica, elegir se vuelve automatico. Lo importante es que el código se lea fácil: si tienes que releer tres veces tu propio ternario para entenderlo, es señal de que debiste usar un if.

useState: el hook que guarda memoria en tu componente

Hasta aquí, tus componentes son como personas sin memoria: reciben props, pintan algo y olvidan todo. Si el usuario hace clic en un botón, el componente no tiene donde guardar que ya se hizo clic. Necesitamos una forma de que el componente recuerde cosas entre un pintado y otro. Esa forma es el estado, y el hook para manejarlo es useState.

Primero, una distinción fundamental: props vs estado. Las props son información que un componente recibe de su padre, como los ingredientes que le pasan a un cocinero. El estado es información que el componente maneja por si mismo, como las notas que el cocinero apunta en su libreta. Las props vienen de fuera y el componente no las puede cambiar; el estado vive dentro y el componente si lo puede actualizar.

El experimento mental: por qué una variable normal no sirve

Mira este código y dime que crees que pasa:

```
function Contador() {
  let numero = 0;

  function incrementar() {
    numero = numero + 1;
    console.log(numero);
  }

  return <button onClick={incrementar}>Clics: {numero}</button>;
}
```

Haces clic y la consola muestra 1, 2, 3... la variable SI cambia. Pero el botón sigue diciendo "Clics: 0". ¿Por qué? Porque React no sabe que algo cambió. React vuelve a pintar un componente solo en dos situaciones: cuando cambian sus props o cuando cambia su estado. Cambiar una variable local no avisa a nadie, así que la pantalla se queda con el valor viejo. Es como cambiar el precio en tu libreta sin avisarle al cajero: tu libreta dice lo nuevo, pero la caja registradora sigue cobrando lo viejo.

Además hay otro problema: cada vez que el componente se vuelve a pintar por cualquier razón, la función se ejecuta de nuevo y `let número = 0` se reinicia a cero. La variable no sobrevive entre pintados. Necesitamos un lugar donde guardar el valor que sobreviva a los re-pintados y que además le avise a React cuando cambie. Ese lugar es el estado con `useState`.

Anatomía de `useState`

Primero se importa el hook:

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

Y se usa así dentro del componente:

```
function Contador() {
  const [numero, setNumero] = useState(0);

  function incrementar() {
    setNumero(numero + 1);
  }

  return <button onClick={incrementar}>Clics: {numero}</button>;
}
```

Vamos por partes. `useState(0)` hace dos cosas: crea una "casilla de memoria" para este componente con valor inicial 0, y devuelve un arreglo con dos elementos. El primero es el valor actual del estado (número). El segundo es una función para actualizarlo (`setNumero`). Por convención se llama `set` seguido del nombre del valor: si el estado se llama `nombre`, la función es `setNombre`.

La línea `const [numero, setNumero] = useState(0)` usa desestructuración de arreglos, que ya conoces: toma el arreglo que devuelve `useState` y asigna su primer elemento a `numero` y el segundo a `setNumero`. Es exactamente lo mismo que escribir:

```
const estado = useState(0);
const numero = estado[0];
const setNumero = estado[1];
```

Pero la versión desestructurada es más corta y es el estandar en todo el código React del mundo.

Cuando llamas a `setNumero` con un valor nuevo, pasan dos cosas: React guarda el valor nuevo en la casilla de memoria y React vuelve a pintar el componente, esta vez con el valor actualizado. Por eso el botón ahora si muestra "Clicks: 1", "Clicks: 2", etc. La función `set` es el "aviso al cajero" que le faltaba a la variable normal.

Nota algo importante: nunca modifiques el valor del estado directamente. No hagas `numero = numero + 1` esperando que la pantalla cambie, porque no va a cambiar: solo la función `set` provoca el re-pintado. El valor del estado se trata como si fuera de solo lectura; la única forma legítima de cambiarlo es con su función `set`.

El estado inicial

El valor que le pasas a `useState` es el valor inicial, el que tendrá el estado la primera vez que se pinte el componente. Puede ser de cualquier tipo: número, texto, booleano, arreglo, objeto, `null`.

```
const [nombre, setNombre] = useState("visitante");
const [logueado, setLogueado] = useState(false);
const [tareas, setTareas] = useState([]);
const [usuario, setUsuario] = useState(null);
```

Un detalle fino: si el valor inicial requiere un cálculo costoso, como procesar un arreglo enorme, puedes pasarle una función a `useState` y React solo la ejecutará una vez, en el primer pintado:

```
const [datos, setDatos] = useState(() => procesarDatosPesados());
```

Esto se llama inicialización diferida. No la necesitaras todos los días, pero es bueno saber que existe para cuando el calculo inicial sea pesado.

Las actualizaciones son asincronas: la trampa de `setNumero(número + 1)`

Aquí viene uno de los errores más comunes de los principiantes. Mira este código:

```
function incrementarTres() {  
  setNumero(numero + 1);  
  setNumero(numero + 1);  
  setNumero(numero + 1);  
}
```

Si número vale 0, cuanto crees que valdra después de llamar a `incrementarTres`? La intuición dice 3. La realidad dice 1. ¿Por qué? Porque las tres llamadas usan el mismo valor de número: el valor que tenía cuando se pinto el componente, que es 0. Las actualizaciones de estado no se aplican de inmediato; React las pone en una fila y las procesa juntas para ser eficiente. Es como pedir tres tacos al taquero de uno en uno pero el solo escucha el último pedido de cada ronda: las tres veces le pediste "uno más que cero", así que te da uno.

Además, si intentas leer el estado justo después de setearlo, verás el valor viejo:

```
function manejarClic() {  
  setNumero(numero + 1);  
  console.log(numero); // muestra el valor VIEJO, no el nuevo  
}
```

Esto no es un error de React, es su diseno: el valor de número dentro de esta ejecución de la función es una foto del momento del pintado, y no cambia hasta el siguiente pintado. Si necesitas hacer algo con el valor nuevo, hazlo en el siguiente pintado o calculalo tú mismo en una variable.

La forma funcional: `setNumero(n => n + 1)`

La solución para actualizaciones que dependen del valor anterior es la forma funcional: en lugar de pasarle el valor nuevo a `set`, le pasas una función que recibe el valor más reciente y devuelve el nuevo:

```
function incrementarTres() {  
  setNumero((n) => n + 1);  
  setNumero((n) => n + 1);  
  setNumero((n) => n + 1);  
}
```

Ahora si: cada llamada recibe el valor actualizado por la anterior, así que 0 se convierte en 3 correctamente. React pone las tres funciones en fila y las ejecuta en orden, cada una con el resultado de la anterior.

La regla práctica: si el nuevo valor se calcula a partir del valor anterior, usa la forma funcional. Si el nuevo valor no depende del anterior, como `setNombre("Ana")`, la forma directa está bien. Acostúmbrate a la forma funcional en contadores, toggles e incrementos, y evitaras una familia entera de errores.

Estado con objetos y arreglos: la inmutabilidad

En React nunca debes mutar el estado directamente. Mutar significa cambiar el objeto o arreglo original en su lugar: hacerle push a un arreglo del estado, o cambiarle una propiedad a un objeto del estado. En su lugar, debes crear una copia nueva con los cambios. Esto se llama inmutabilidad, y es importante porque React decide si debe re-pintar comparando referencias: si le pasas el mismo arreglo de siempre, aunque le hayas agregado algo por dentro, React puede no notar el cambio.

Para objetos se usa el operador spread (...), que copia todas las propiedades:

```
const [usuario, setUsuario] = useState({ nombre: "Ana", edad: 25 });  
  
// Correcto: crear un objeto nuevo con el cambio  
setUsuario({ ...usuario, edad: 26 });  
  
// MAL: mutar directamente, React puede no re-pintar  
// usuario.edad = 26;
```

El spread copia nombre: "Ana" y luego edad: 26 sobrescribe la edad vieja. El resultado es un objeto nuevo, con referencia nueva, y React detecta el cambio sin problema.

Para arreglos igual: para agregar un elemento, crea un arreglo nuevo con spread:


```
const [tareas, setTareas] = useState(["estudiar", "dormir"]);

// Correcto: arreglo nuevo con el elemento agregado
setTareas([...tareas, "programar"]);

// MAL: mutar con push
// tareas.push("programar");
```

Para quitar un elemento se usa filter, que devuelve un arreglo nuevo sin el elemento:

```
setTareas(tareas.filter((t) => t !== "dormir"));
```

Para modificar un elemento se usa map, que devuelve un arreglo nuevo:

```
const [productos, setProductos] = useState([
  { id: 1, nombre: "Laptop", enCarrito: false },
  { id: 2, nombre: "Mouse", enCarrito: false }
]);

setProductos(
  productos.map((p) =>
    p.id === 1 ? { ...p, enCarrito: true } : p
  )
);
```

Fíjate en el patrón: map recorre, y para el producto con id 1 crea un objeto nuevo con spread y la propiedad cambiada; los demás se quedan igual. Es el mismo map de renderizar listas, ahora usado para actualizar el estado. Todo se conecta.

Múltiples estados vs un solo objeto

Puedes llamar a useState todas las veces que quieras en un componente. Cada llamada crea una casilla de memoria independiente:

```
const [nombre, setNombre] = useState("");
const [edad, setEdad] = useState(0);
const [correo, setCorreo] = useState("");
```

O puedes agrupar todo en un objeto:

```
const [formulario, setFormulario] = useState({ nombre: "", edad: 0, correo: "" });
```

Cual es mejor? Depende. La guía práctica: si los valores cambian juntos y pertenecen a la misma "cosa" (los campos de un formulario, los datos de un usuario), un objeto tiene sentido. Si son independientes (un contador y un texto de búsqueda que no tienen nada que ver), estados separados son más simples, porque actualizar uno no te obliga a copiar los demás con spread. No hay una ley absoluta: empieza con estados separados, que es lo más simple, y agrupa en objetos cuando notes que siempre actualizas varios valores a la vez.

Listas anidadas, fragmentos y keys en componentes hijos

En aplicaciones reales las listas rara vez son planas. Piensa en un menu de restaurante: tiene categorías, y cada categoría tiene platillos. Eso es una lista dentro de otra lista, y se resuelve con un map dentro de otro map:

```
const menu = [
  { id: 1, categoria: "Desayunos", platillos: ["Hotcakes", "Chilaquiles"] },
  { id: 2, categoria: "Comidas", platillos: ["Tacos", "Enchiladas"] }
];

function MenuRestaurante() {
  return (
    <div>
      {menu.map((seccion) => (
        <div key={seccion.id}>
          <h3>{seccion.categoria}</h3>
          <ul>
            {seccion.platillos.map((platillo) => (
              <li key={platillo}>{platillo}</li>
            ))}
          </ul>
        </div>
      ))}
    </div>
  );
}
```

Cada nivel de map necesita sus propias keys. La key solo tiene que ser única entre sus hermanos, no en toda la aplicación: dos listas distintas pueden repetir keys sin problema, porque React las compara dentro de su propio nivel.

Otro caso comun: quieres devolver varios elementos por cada item del map sin envolverlos en un div extra que arruine tu diseno. Para eso sirven los fragmentos. El fragmento corto `<></>` no acepta key, así que cuando necesitas key debes usar la forma larga:

```
{productos.map((p) => (
  <Fragment key={p.id}>
    <h3>{p.nombre}</h3>
    <p>{p.precio}</p>
  </Fragment>
))}
```

Para usarlo necesitas importar Fragment de react: `import { Fragment } from "react"`. Es un detalle pequeño, pero cuando tu CSS depende de que no haya divs intermedios, te salva el día.

También es muy común extraer el contenido de cada item a su propio componente para mantener el código ordenado. En ese caso la key va en el componente dentro del map, no dentro del componente hijo:

```
{productos.map((p) => (
  <TarjetaProducto key={p.id} producto={p} />
))}
```

La key no se pasa como prop normal: React la usa internamente y el componente hijo no la recibe. Si el hijo necesita el id, pasalo también como prop explícita, por ejemplo `id={p.id}`.

Estado derivado: calcula en lugar de guardar

Un error de diseño muy común es guardar en el estado cosas que se pueden calcular a partir del estado que ya tienes. Mira este ejemplo:

```
const [productos, setProductos] = useState([...]);
const [total, setTotal] = useState(0); // estado derivado: mala idea
```

El total se puede calcular sumando los precios de productos en cualquier momento. Si lo guardas en un estado separado, tienes dos fuentes de verdad que se pueden desincronizar: cada vez que agregues o quites un producto tendrás que acordarte de actualizar el total también, y el día que se te olvide, la pantalla mostrara un total incorrecto. La solución es calcularlo durante el pintado:

```
const [productos, setProductos] = useState([...]);
const total = productos.reduce((suma, p) => suma + p.precio, 0);
```

total no es estado: es una variable normal calculada a partir del estado en cada pintado. Como productos si es estado, cada vez que cambie el componente se vuelve a pintar y el total se recalcula solo. Una sola fuente de verdad, cero riesgo de desincronización.

La regla general: si un valor se puede calcular a partir de props o de estado existente, no lo guardes en otro estado. Calculalo. Reserva useState para los datos que cambian por acción del usuario o del sistema y que no se pueden deducir de nada más: el texto de un input, los items de una lista, si algo esta encendido o apagado.

Las reglas de los hooks: el contrato con React

useState es un hook, y los hooks tienen reglas estrictas que debes respetar siempre. Son solo dos, pero romperlas causa errores extraños.

Regla uno: los hooks solo se llaman en el nivel superior del componente. Nunca dentro de un if, un for, un while o una función anidada. Esto es incorrecto:

```
// MAL: hook dentro de un condicional
if (condicion) {
  const [valor, setValor] = useState(0);
}
```

¿Por qué? Porque React identifica cada hook por el orden en que se llama. En el primer pintado registra: "primer hook, segundo hook, tercer hook". En el siguiente pintado espera encontrarlos en el mismo orden. Si un hook está dentro de un if, en un pintado existe y en otro no, el orden se rompe y React le entrega a cada hook el estado del hook equivocado. El resultado son valores mezclados y errores difíciles de rastrear. La solución es simple: llama todos tus useState al inicio del componente, siempre en el mismo orden, y usa los valores después con condicionales normales.

Regla dos: los hooks solo se llaman desde componentes de React o desde otros hooks personalizados. Nunca desde funciones normales de JavaScript. Esto garantiza que siempre haya un componente "dueno" del estado.

Si sigues estas dos reglas, los hooks se comportan de forma predecible. Vale la pena repetirlo porque es la causa número uno de confusiones raras en principiantes: cuando tu estado se comporta de forma imposible, revisa primero que no estes llamando un hook dentro de un if.

El ciclo completo: como trabajan juntas las tres herramientas

Ya viste cada herramienta por separado. Ahora mira el patrón completo que las une, porque es el que repetirás en casi todo lo que construyas con React. El ciclo tiene tres pasos y siempre va en este orden.

Paso uno: los datos viven en el estado. Preguntate que información cambia con el tiempo en tu interfaz: la lista de tareas, el texto del buscador, si el menu está abierto, los productos del carrito. Todo eso va en `useState`. Si un dato nunca cambia, como el catálogo de la tienda del ejemplo 5, no necesita estado: puede ser una constante fuera del componente.

Paso dos: los condicionales deciden que mostrar. Con los datos del estado en la mano, decides los casos: si la lista está vacía muestra un mensaje, si esta cargando muestra un spinner, si el usuario es admin muestra el botón de borrar. Aquí eliges la herramienta según el caso: ternario para dos caminos, `&&` para mostrar u ocultar, `if` con variable o `switch` para lógica ramificada.

Paso tres: `map` convierte las colecciones en pantalla. Los arreglos del estado se transforman en elementos con `map` y cada uno lleva su `key`. Si el arreglo está vacío, el paso dos ya se encargo de mostrar el mensaje alternativo, así que el `map` solo se ejecuta cuando hay algo que pintar.

Mira el ciclo en una sola función mental: estado -> decisión -> lista. La lista de invitados del ejercicio 5 lo muestra perfecto: `invitados` y `nombre` son estado, el ternario decide entre "Todavía no hay invitados" y la lista, y `map` pinta cada invitado con su `key` y su botón de eliminar. Cuando agregas un invitado, el estado cambia, React vuelve a pintar, el ternario ve que ya hay elementos y el `map` pinta la lista actualizada. Todo fluye solo.

Este ciclo también te ayuda a depurar. Si tu interfaz no muestra lo que esperas, revisa en orden: primero, el estado tiene los datos correctos? Puedes poner un `console.log` del estado antes del `return` para verlo. Segundo, el condicional esta evaluando lo que crees? Revisa la condición con los valores reales. Tercero, el `map` esta recorriendo el arreglo correcto y cada elemento tiene `key`? La mayoría de los errores de esta unidad caen en una de estas tres preguntas, y responderlas en orden te lleva directo al problema sin andar adivinando.

Acostumbrate a pensar así antes de escribir código: que datos cambian, que casos existen, que listas voy a pintar. Cinco minutos de este análisis te ahorran una hora de prueba y error. Con el tiempo se vuelve automatico y empezaras a ver el patrón estado -> decisión -> lista en cada pantalla que disenes.

Analogías para que no se te olvide

Piensa en el estado como la memoria de un personaje de videojuego: su vida, sus monedas, su nivel. Cada vez que algo cambia, la pantalla se actualiza para mostrarlo. La función set es como guardar la partida: sin guardar, el juego no registra tu avance.

O piensa en useState como un pizarrón con un borrador mágico: tu escribes el valor inicial con gises, y cada vez que llamas a set, el borrador mágico borra el valor viejo, escribe el nuevo y grita "¡pinten de nuevo!" para que React actualice la pantalla.

Y las keys de las listas son como los números de folio de los alumnos: sin folio, el maestro se pierde cuando alguien cambia de lugar; con folio, siempre sabe quien es quien.

Errores comunes y como evitarlos

Uno: mutar el estado directamente. Hacer `tareas.push("nueva")` o `usuario.edad = 30` no avisa a React y la pantalla no se actualiza, o se actualiza a medias. Solución: siempre crea copias nuevas con spread, map o filter, y pasalas con la función set.

Dos: olvidar la key en las listas. Funciona, pero la consola se llena de advertencias y la reconciliación se vuelve ineficiente o incorrecta. Solución: pon `key={id}` con un identificador único y estable.

Tres: usar el índice como key en listas que cambian. Causa errores visuales extraños, como checkboxes que se marcan en el elemento equivocado. Solución: usa ids de tus datos; el índice solo para listas totalmente estáticas.

Cuatro: leer el estado justo después de setearlo esperando el valor nuevo. El valor se actualiza en el siguiente pintado, no de inmediato. Solución: si necesitas el valor nuevo ya, guardalo en una variable local primero.

Cinco: llamar a `setNumero(número + 1)` varias veces seguidas esperando que se acumulen. Solo se aplica una vez porque todas usan el valor viejo. Solución: usa la forma funcional `setNumero((n) => n + 1)`.

Seis: poner sentencias if dentro del JSX entre llaves. Las llaves solo aceptan expresiones. Solución: usa ternario, `&&`, o saca la lógica a una variable antes del return.

Siete: el 0 fantasma con `&&`. Escribir `{lista.length && <Componente />}` pinta un "0" cuando la lista está vacía. Solución: usa una comparación booleana explícita, `{lista.length > 0 && <Componente />}`.

Ocho: Llamar hooks condicionalmente. Los hooks como `useState` siempre deben llamarse en el mismo orden, en el nivel superior del componente, nunca dentro de un `if`, un `for` o una función anidada. Si pones un `useState` dentro de un `if`, React se confunde porque en un pintado el hook existe y en otro no. Solución: llama todos tus `useState` al inicio del componente, siempre en el mismo orden.

Buenas prácticas para cerrar

Nombra tu estado de forma descriptiva: `[tareas, setTareas]` se entiende solo; `[x, setX]` no dice nada. Inicializa el estado con el tipo correcto desde el inicio: si será un arreglo, inicialízalo como `[]` y no como `null`, para poder usar `.map` sin miedo. Mantén el estado lo más simple posible: no guardes en el estado cosas que puedes calcular a partir de él, como el total de un carrito que puedes calcular con los productos. Sube el estado al componente padre solo cuando dos componentes hermanos necesiten compartirlo; mientras un solo componente lo use, el estado vive ahí. Y cuando combines las tres herramientas de esta unidad, sigue este orden mental: primero define qué datos cambian (estado), después decide cómo se muestran según condiciones (condicionales) y al final pinta las colecciones (listas con `map` y `key`). Con ese orden, casi cualquier interfaz interactiva se vuelve un problema manejable.

Ejemplos paso a paso

En esta sección vamos a construir cuatro ejemplos completos, explicados línea por línea. Todos están listos para probar en tu proyecto de Vite: crea un archivo nuevo en `src` para cada uno, importalo en `App.jsx` y corre `npm run dev`. Escribe el código tú mismo en lugar de copiarlo y pegarlo: tus dedos también necesitan aprender.

Ejemplo 1: lista de frutas con keys

Empezamos con lo más básico del renderizado de listas: un arreglo de objetos, cada uno con su `id`, convertido en una lista HTML.

```

const frutas = [
  { id: 1, nombre: "Manzana", color: "roja" },
  { id: 2, nombre: "Pera", color: "verde" },
  { id: 3, nombre: "Uva", color: "morada" },
  { id: 4, nombre: "Platano", color: "amarillo" }
];

function ListaFrutas() {
  return (
    <div>
      <h2>Mis frutas favoritas</h2>
      <ul>
        {frutas.map((fruta) => (
          <li key={fruta.id}>
            {fruta.nombre} - es de color {fruta.color}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default ListaFrutas;

```

Línea por línea: primero definimos frutas, un arreglo de objetos. Cada objeto tiene un id único, un nombre y un color. El id será nuestra key. Definir los datos fuera del componente está bien porque son estáticos: no cambian nunca.

Dentro del componente, el return pinta un div con un título h2 y una lista ul. Dentro del ul abrimos llaves para entrar a JavaScript y llamamos a frutas.map. Por cada fruta, la función flecha devuelve un li. Fíjate en los paréntesis que envuelven el JSX del li: como ocupa varias líneas, los paréntesis evitan problemas con el return implícito de la flecha.

Cada li lleva key={fruta.id}. Es la prop especial que React usa para la reconciliación: así sabe que el li con id 2 siempre es la pera, aunque el orden cambie. Dentro del li mostramos el nombre y el color con llaves, interpolando valores del objeto.

El resultado en pantalla es una lista de cuatro elementos: "Manzana - es de color roja", y así con cada una. Si abres la consola del navegador, no verás ninguna advertencia, porque cada elemento tiene su key. Prueba este experimento: quita la prop key, guarda y mira la consola. Verás la famosa advertencia de React. Vuelve a ponerla y desaparece. Así de directo es el feedback.

Ejemplo 2: semaforo condicional con ternario y &&

Ahora combinamos condicionales: un semaforo que muestra un mensaje distinto según el color, más un aviso extra que solo aparece en rojo.

```
function Semaforo({ color }) {  
  return (  
    <div>  
      <h2>Semaforo</h2>  
      <p>  
        {color === "rojo"  
          ? "Alto total: no pases"  
          : color === "amarillo"  
            ? "Precaucion: baja la velocidad"  
            : color === "verde"  
              ? "Avance: puedes pasar"  
              : "Color no reconocido"}  
        </p>  
        {color === "rojo" && <p>Recuerda: el rojo dura 30 segundos.</p>}  
      </div>  
    );  
  }  
}  
  
export default Semaforo;
```

Analicemos: el componente recibe la prop color. El primer condicional es un ternario anidado que elige entre cuatro mensajes. Te lo muestro aquí a proposito para que veas cómo se ve en código real, pero recuerda la lección de la teoría: con más de dos o tres ramas, un switch o un if con variable antes del return se lee mejor. Si este semaforo creciera, lo refactorizarías.

La indentación escalonada ayuda a leer el anidado: cada ? y : de un nivel más profundo va más a la derecha. Aun así, nota como cuesta seguirlo con la vista. Es la prueba viviente de por qué la teoría recomienda no abusar del anidado.

Después del parrafo viene el operador &&: {color === "rojo" && <p>...</p>}. Solo cuando el color es rojo se pinta el aviso de los 30 segundos. En cualquier otro color, la expresión vale false y React no pinta nada. Fíjate que la condición es una comparación con ===, así que siempre es un booleano: no hay riesgo del 0 fantasma.

Para probarlo, usalo así en App.jsx:

```
<Semaforo color="rojo" />  
<Semaforo color="verde" />
```

Cambia el valor de la prop y observa como el mensaje cambia. Eso es un condicional en JSX en acción: la misma estructura, distinto contenido según los datos.

Ejemplo 3: contador con useState

El ejemplo clásico para entender el estado: un contador con botones para sumar, restar y reiniciar. Aquí verás la forma funcional de actualizar el estado.

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

function Contador() {
  const [numero, setNumero] = useState(0);

  function sumar() {
    setNumero((n) => n + 1);
  }

  function restar() {
    setNumero((n) => n - 1);
  }

  function reiniciar() {
    setNumero(0);
  }

  return (
    <div>
      <h2>Contador: {numero}</h2>
      <button onClick={sumar}>+1</button>
      <button onClick={restar}>-1</button>
      <button onClick={reiniciar}>Reiniciar</button>
    </div>
  );
}

export default Contador;
```

Línea por línea: importamos useState de react, porque sin el import el hook no existe en este archivo. Dentro del componente, const [número, setNumero] = useState(0) crea la casilla de memoria con valor inicial 0 y nos da el valor actual y la función para cambiarlo.

Las funciones sumar y restar usan la forma funcional: setNumero((n) => n + 1). La función recibe el valor más reciente (n) y devuelve el nuevo. Así, si el usuario hace clic rapidísimo

varias veces, cada actualización parte del valor correcto y no se pierde ningún clic. reiniciar usa la forma directa `setNumero(0)` porque el nuevo valor no depende del anterior: siempre es 0.

En el return, el `h2` muestra el valor actual con `{numero}`. Cada vez que el estado cambia, React vuelve a pintar el componente y el `h2` muestra el número nuevo. Los tres botones llaman a sus funciones con `onClick`. Fíjate que escribimos `onClick={sumar}` sin paréntesis: le pasamos la función para que React la llame cuando ocurra el clic. Si escribieras `onClick={sumar()}` con paréntesis, la función se ejecutaría de inmediato al pintarse, y eso no es lo que quieres.

Prueba este experimento para entender la asincronía: cambia `sumar` por una versión que llame tres veces a `setNumero(numero + 1)` con la forma directa. Haz clic una vez y verás que el contador solo sube 1, no 3. Ahora vuelve a la forma funcional con tres llamadas a `setNumero((n) => n + 1)` y verás que sube 3. Esa diferencia es la lección de las actualizaciones en fila, comprobada con tus propios ojos.

Ejemplo 4: lista de tareas agregando items con spread

El ejemplo que une todo: estado con un arreglo, un input controlado por estado, agregar elementos con `spread`, renderizar la lista con `map` y `key`, y un condicional para la lista vacía.

```

import { useState } from "react";

function ListaTareas() {
  const [tareas, setTareas] = useState([]);
  const [texto, setTexto] = useState("");

  function agregarTarea() {
    if (texto.trim() === "") {
      return;
    }
    const nuevaTarea = {
      id: Date.now(),
      texto: texto
    };
    setTareas([...tareas, nuevaTarea]);
    setTexto("");
  }

  return (
    <div>
      <h2>Mis tareas</h2>
      <input
        type="text"
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        placeholder="Escribe una tarea"
      />
      <button onClick={agregarTarea}>Agregar</button>

      {tareas.length === 0 ? (
        <p>No hay tareas. Agrega la primera.</p>
      ) : (
        <ul>
          {tareas.map((tarea) => (
            <li key={tarea.id}>{tarea.texto}</li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default ListaTareas;

```

Vamos con calma porque aquí pasa mucho. Tenemos dos estados: tareas, un arreglo que empieza vacío, y texto, que guarda lo que el usuario escribe en el input. El input es un input controlado: su value siempre es igual al estado texto, y cada vez que el usuario escribe, onChange actualiza el estado con e.target.value. Así el estado y lo que se ve en pantalla siempre están sincronizados.

La función agregarTarea primero valida: si el texto está vacío o solo tiene espacios, hace un return temprano y no agrega nada. Es el patrón de returns tempranos aplicado a lógica de negocio. Si hay texto, crea un objeto nuevaTarea con un id generado por Date.now(), que da un número único basado en la hora actual, perfecto como key porque nunca se repite.

Luego viene la línea clave: setTareas([...tareas, nuevaTarea]). El spread copia todas las tareas existentes en un arreglo nuevo y agrega la nueva al final. No mutamos el arreglo original: creamos uno nuevo, como manda la inmutabilidad. Después limpiamos el input con setTexto(""), que también limpia lo que se ve porque el input esta controlado por ese estado.

En el return, el condicional decide que pintar: si tareas.length === 0, muestra el mensaje "No hay tareas"; si no, pinta la lista con map, cada li con su key={tarea.id}. Fíjate como las tres herramientas de la unidad trabajan juntas en un solo componente: estado para los datos, condicional para la lista vacía, map con keys para la lista.

Para probarlo, escribe varias tareas y agrégalas. Luego abre las herramientas de desarrollo y observa que no hay advertencias de keys. Este ejemplo es la mini versión de cualquier app real de lista: un carrito, una lista de comentarios o una bandeja de mensajes funcionan con la misma estructura.

Ejemplo 5: tienda con stock y carrito

Cerremos los ejemplos con uno que mezcla las tres herramientas en un escenario realista: una lista de productos donde cada uno muestra su disponibilidad con condicionales y un botón que lo agrega al carrito actualizando el estado.

```

import { useState } from "react";

const catalogo = [
  { id: 1, nombre: "Laptop", precio: 15000, stock: 3 },
  { id: 2, nombre: "Mouse", precio: 300, stock: 0 },
  { id: 3, nombre: "Teclado", precio: 800, stock: 5 }
];

function Tienda() {
  const [carrito, setCarrito] = useState([]);

  function agregarAlCarrito(producto) {
    setCarrito([...carrito, producto]);
  }

  const total = carrito.reduce((suma, p) => suma + p.precio, 0);

  return (
    <div>
      <h2>Tienda</h2>
      <p>Productos en el carrito: {carrito.length}</p>

      {catalogo.map((producto) => (
        <div key={producto.id}>
          <h3>{producto.nombre} - ${producto.precio}</h3>
          {producto.stock > 0 ? (
            <p>Disponible: {producto.stock} piezas</p>
          ) : (
            <p>Agotado</p>
          )}
          {producto.stock > 0 && (
            <button onClick={() => agregarAlCarrito(producto)}>
              Agregar al carrito
            </button>
          )}
        </div>
      ))}

      <h3>Total del carrito: ${total}</h3>
    </div>
  );
}

export default Tienda;

```

Analicemos por capas. El catálogo es estático, así que vive fuera del componente. El carrito si cambia, así que es estado que inicia como arreglo vacío. La función `agregarAlCarrito` crea un arreglo nuevo con `spread` más el producto, respetando la inmutabilidad.

El total es estado derivado: se calcula con `reduce` a partir del carrito en cada `pintado`, no se guarda en otro estado. Así nunca se desincroniza.

En el JSX, el `map` recorre el catálogo con `key={producto.id}`. Dentro de cada tarjeta hay un `ternario` que muestra "Disponible" o "Agotado" según el stock, y un `&&` que solo pinta el botón de agregar cuando hay stock. El botón usa una flecha para pasarle el producto como argumento a `agregarAlCarrito`. Fíjate como conviven los patrones que aprendiste: `ternario` para dos textos alternativos, `&&` para mostrar u ocultar el botón, todo dentro del mismo `map`.

Pruebalo agregando productos al carrito y observa como el contador y el total se actualizan solos. Este ejemplo es, en miniatura, el corazon de cualquier tienda en línea.

Para llevar los ejemplos más lejos

Los cinco ejemplos funcionan tal cómo están, pero aquí tienes variaciones para exprimirlos y convertir cada uno en un mini laboratorio. Haz una variación por día y tu dominio del tema se va a disparar.

Con la lista de frutas, prueba ordenar el arreglo alfabeticamente antes del `map` usando `[...frutas].sort()` y observa que las keys con `id` mantienen cada elemento en su lugar. Luego cambia las keys a índices, agrega un `checkbox` a cada fruta, marca algunas y reordena: verás con tus propios ojos el problema de los índices como key que la teoría describe.

Con el semaforo, refactoriza el `ternario` anidado a un `switch` con variable antes del `return` y compara la legibilidad. Después agrega una `prop` tiempo que muestre los segundos restantes y un `&&` que pinte "Casi cambia" cuando queden menos de 5 segundos.

Con el contador, agrega un limite: que no pueda pasar de 20 ni bajar de 0, usando condicionales en las funciones de sumar y restar. Luego agrega un historial: un estado que guarde en un arreglo cada valor por el que pasa el contador y lo pinte con `map` debajo. Ahí estaras combinando estado numerico con estado de arreglo en el mismo componente.

Con la lista de tareas, agrega la función de eliminar del ejercicio de invitados y un `checkbox` para marcar tareas como completadas, guardando `completada: true` o `false` en cada objeto y usando `map` para alternarlo. El tachado del texto completado lo logras con un `ternario` en el `className`.

Con la tienda, agrega un botón para vaciar el carrito con `setCarrito([])` y un botón por producto en el carrito para quitarlo con `filter`. Cuando logres eso sin mirar los ejemplos, tendrás el patrón completo dominado: crear, leer, actualizar y borrar elementos del estado, que en la industria se conoce como operaciones CRUD y es la base de casi toda aplicación que maneja datos.

Práctica

Es hora de ensuciarte las manos. Aquí tienes cinco ejercicios ordenados de menor a mayor dificultad. Intenta resolver cada uno por tu cuenta antes de mirar la solución: tapa la solución con la mano, escribe tu versión, pruebala en el navegador y solo después compara. Equivocarte aquí es gratis y es donde más se aprende.

Ejercicio 1: galeria de productos desde un arreglo

Enunciado: crea un componente `GaleriaProductos` que reciba por props un arreglo de productos, donde cada producto tiene `id`, `nombre` y `precio`. Debe pintar una tarjeta por producto con su nombre y su precio, usando `map`. Cada tarjeta debe tener su `key` correcta.

Pista: una tarjeta puede ser un `div` con un `h3` para el nombre y un `p` para el precio.

Solución:

```
function GaleriaProductos({ productos }) {
  return (
    <div>
      <h2>Productos</h2>
      <div>
        {productos.map((producto) => (
          <div key={producto.id}>
            <h3>{producto.nombre}</h3>
            <p>Precio: ${producto.precio}</p>
          </div>
        ))}
      </div>
    </div>
  );
}

export default GaleriaProductos;
```

Explicación paso a paso: el componente recibe `productos` por props, así que lo desestructuramos en los parámetros. Dentro del JSX abrimos llaves y aplicamos `map` al

arreglo. Por cada producto devolvemos un div que funciona como tarjeta, con `key={producto.id}` para que React identifique cada tarjeta. Dentro de la tarjeta interpolamos `producto.nombre` en un `h3` y `producto.precio` en un párrafo. Para probarlo, pasale este arreglo desde `App.jsx`:

```
const productos = [
  { id: 1, nombre: "Laptop", precio: 15000 },
  { id: 2, nombre: "Mouse", precio: 300 },
  { id: 3, nombre: "Teclado", precio: 800 }
];
<GaleriaProductos productos={productos} />
```

Lo que practicas aquí: transformar datos en interfaz con `map` y asignar keys estables desde los ids de los datos. Si olvidas la key, la consola te lo recordara.

Ejercicio 2: interruptor encendido/apagado

Enunciado: crea un componente `Interruptor` con un estado booleano `encendido` que inicie en `false`. Debe mostrar el texto "La luz esta encendida" o "La luz esta apagada" según el estado, usando un `ternario`. Un botón debe alternar el estado entre `true` y `false`.

Pista: para alternar un booleano usa la forma funcional: `setEncendido((e) => !e)`.

Solución:

```

import { useState } from "react";

function Interruptor() {
  const [encendido, setEncendido] = useState(false);

  function alternar() {
    setEncendido((e) => !e);
  }

  return (
    <div>
      <h2>{encendido ? "La luz esta encendida" : "La luz esta apagada"}</h2>
      <button onClick={alternar}>
        {encendido ? "Apagar" : "Encender"}
      </button>
    </div>
  );
}

export default Interruptor;

```

Explicación paso a paso: el estado encendido guarda si la luz esta prendida o no, iniciando en false. La función alternar usa la forma funcional con la negación !e: si estaba en true pasa a false y viceversa. Usamos la forma funcional porque el nuevo valor depende del anterior, y así evitamos problemas si el usuario hace clic muy rápido. En el JSX hay dos ternarios: uno decide el texto del título y otro decide el texto del botón. Fíjate en el detalle de experiencia de usuario: el botón dice "Encender" cuando la luz esta apagada y "Apagar" cuando esta encendida, es decir, describe la acción que va a ocurrir, no el estado actual. Es un detalle pequeño que hace las interfaces más claras.

Lo que practicas aquí: estado booleano, actualización funcional con negación y ternarios para texto dinámico.

Ejercicio 3: contador con botones +1, -1 y reset

Enunciado: crea un componente ContadorBotones con un estado número que inicie en 0. Tres botones: uno suma 1, otro resta 1 y otro reinicia a 0. Además, si el número es mayor que 10, muestra un aviso que diga "Vas muy alto"; si es menor que -10, muestra "Vas muy bajo". Usa el operador && para los avisos.

Solución:

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

function ContadorBotones() {
  const [numero, setNumero] = useState(0);

  return (
    <div>
      <h2>Valor: {numero}</h2>
      <button onClick={() => setNumero((n) => n + 1)}>+1</button>
      <button onClick={() => setNumero((n) => n - 1)}>-1</button>
      <button onClick={() => setNumero(0)}>Reiniciar</button>
      {numero > 10 && <p>Vas muy alto</p>}
      {numero < -10 && <p>Vas muy bajo</p>}
    </div>
  );
}

export default ContadorBotones;
```

Explicación paso a paso: el estado número inicia en 0. En lugar de definir funciones separadas, aquí pasamos funciones flecha directamente al onClick: `onClick={() => setNumero((n) => n + 1)}`. La flecha es necesaria porque queremos llamar a `setNumero` con un argumento; si escribiéramos `onClick={setNumero((n) => n + 1)}` se ejecutaría al pintarse. Los botones de +1 y -1 usan la forma funcional porque dependen del valor anterior; el de reiniciar usa la forma directa con 0 porque no depende de nada. Los avisos usan `&&` con comparaciones booleanas explícitas (`número > 10`), así no hay riesgo del 0 fantasma. Cuando el número está entre -10 y 10, ambas condiciones son false y no se pinta nada.

Lo que practicas aquí: handlers en línea con flechas, forma funcional vs directa y condicionales `&&` para avisos.

Ejercicio 4: input que muestra texto en vivo

Enunciado: crea un componente `TextoEnVivo` con un estado texto que inicie como cadena vacía. Un input controlado por ese estado y, debajo, un párrafo que muestre "Escribiste: " seguido de lo que el usuario escribe, en tiempo real. Si el texto está vacío, en lugar del párrafo muestra "Empieza a escribir algo" usando un ternario.

Solución:

```

import { useState } from "react";

function TextoEnVivo() {
  const [texto, setTexto] = useState("");

  return (
    <div>
      <h2>Texto en vivo</h2>
      <input
        type="text"
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        placeholder="Escribe algo"
      />
      {texto === "" ? (
        <p>Empieza a escribir algo</p>
      ) : (
        <p>Escribiste: {texto}</p>
      )}
    </div>
  );
}

export default TextoEnVivo;

```

Explicación paso a paso: el estado `texto` es la única fuente de verdad. El `input` es controlado: `value={texto}` hace que siempre muestre el estado, y `onChange={(e) => setTexto(e.target.value)}` actualiza el estado con cada tecla que el usuario presiona. Cada vez que el estado cambia, React re-pinta y el párrafo muestra el texto nuevo al instante: por eso se ve "en vivo". El ternario decide entre el mensaje inicial y el texto escrito, comparando `texto === ""`. Fíjate que la comparación es estricta contra cadena vacía, así que es un booleano limpio. Este patrón de `input` controlado es la base de todos los formularios en React: lo verás en logins, buscadores y registros.

Lo que practicas aquí: `inputs` controlados, el evento `onChange` con `e.target.value` y ternario para estado vacío vs con contenido.

Ejercicio 5: lista de invitados agregable

Enunciado: crea un componente ListaInvitados que combine todo lo aprendido. Debe tener: un estado invitados como arreglo vacío, un estado nombre para el input, un botón para agregar al invitado con un id único, la lista renderizada con map y keys, un mensaje cuando no hay invitados y, como reto extra, un botón en cada invitado para eliminarlo de la lista.

Pista: para eliminar usa filter, que devuelve un arreglo nuevo sin el elemento.

Solución:

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

function ListaInvitados() {
  const [invitados, setInvitados] = useState([]);
  const [nombre, setNombre] = useState("");

  function agregarInvitado() {
    if (nombre.trim() === "") {
      return;
    }
    const nuevo = { id: Date.now(), nombre: nombre };
    setInvitados([...invitados, nuevo]);
    setNombre("");
  }

  function eliminarInvitado(id) {
    setInvitados(invitados.filter((inv) => inv.id !== id));
  }

  return (
    <div>
      <h2>Lista de invitados</h2>
      <input
        type="text"
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
        placeholder="Nombre del invitado"
      />
      <button onClick={agregarInvitado}>Agregar invitado</button>

      {invitados.length === 0 ? (
        <p>Todavía no hay invitados.</p>
      ) : (
        <ul>
          {invitados.map((inv) => (
            <li key={inv.id}>
```

```

        {inv.nombre}
        <button onClick={() => eliminarInvitado(inv.id)}>
          Eliminar
        </button>
      </li>
    )}}
  </ul>
)}
</div>
);
}

export default ListaInvitados;

```

Explicación paso a paso: dos estados, como en el ejemplo de tareas: invitados guarda el arreglo y nombre guarda el texto del input controlado. agregarInvitado valida con return temprano si el nombre está vacío, crea el objeto con id único de Date.now() y lo agrega con spread en un arreglo nuevo, sin mutar el original. Después limpia el input.

La novedad es eliminarInvitado: recibe el id del invitado a borrar y usa filter para crear un arreglo nuevo que conserva a todos excepto al que tiene ese id. Fíjate en la condición `inv.id !== id`: filter se queda con los que cumplen la condición, así que conserva a todos menos al eliminado. Es inmutabilidad en acción: nunca tocamos el arreglo original.

En el JSX, el botón de eliminar está dentro de cada li y su onClick es una flecha que llama a eliminarInvitado con el id de ese invitado: `onClick={() => eliminarInvitado(inv.id)}`. La flecha es obligatoria aquí, porque necesitamos pasarle el id como argumento; sin la flecha, la función se ejecutaria durante el pintado y borraría todo al cargar. El ternario muestra el mensaje de lista vacía o la lista con map y keys, cerrando el círculo de las tres herramientas de la unidad.

Lo que practicas aquí: todo junto, estado con arreglos de objetos, agregar con spread, eliminar con filter, keys estables y handlers con argumentos. Si dominas este ejercicio sin mirar la solución, la unidad esta aprendida.

Ejercicio 6: buscador en vivo con filtro

Enunciado: crea un componente Buscador con un arreglo fijo de nombres de frutas, un estado búsqueda para el texto del input y una lista que muestre solo las frutas cuyo nombre incluya el texto buscado, sin importar mayúsculas o minúsculas. Si ninguna fruta coincide, muestra el mensaje "Sin resultados". Usa filter para filtrar y map para pintar.

Pista: para comparar sin importar mayúsculas, convierte ambos textos con `toLowerCase()` antes de usar `includes()`.

Solución:

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

const frutas = ["Manzana", "Pera", "Uva", "Platano", "Mango", "Sandia"];

function Buscador() {
  const [busqueda, setBusqueda] = useState("");

  const filtradas = frutas.filter((fruta) =>
    fruta.toLowerCase().includes(busqueda.toLowerCase())
  );

  return (
    <div>
      <h2>Buscador de frutas</h2>
      <input
        type="text"
        value={busqueda}
        onChange={(e) => setBusqueda(e.target.value)}
        placeholder="Buscar fruta"
      />
      {filtradas.length === 0 ? (
        <p>Sin resultados</p>
      ) : (
        <ul>
          {filtradas.map((fruta) => (
            <li key={fruta}>{fruta}</li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default Buscador;
```

Explicación paso a paso: el arreglo de frutas es fijo, así que va fuera del componente. El estado búsqueda guarda el texto del input controlado. La variable filtradas es estado derivado: se calcula con filter en cada pintado a partir del estado, así que siempre esta sincronizada sin necesidad de otro useState. El filter conserva las frutas cuyo nombre en minúsculas incluye la búsqueda también en minúsculas; con la búsqueda vacía, includes("") es true para todas, así que al inicio se muestran todas las frutas.

El ternario decide entre el mensaje "Sin resultados" y la lista. Como los nombres no se repiten, usamos el nombre como key. Escribe "an" en el buscador y verás como la lista se filtra en vivo: Manzana, Platano... borra el texto y vuelven todas. Este patrón de filtrar con filter y pintar con map es la base de todos los buscadores, filtros de tienda y autocompletados que existen.

Lo que practicas aquí: filter para búsquedas, estado derivado calculado en el pintado, comparaciones sin importar mayúsculas y el ciclo completo estado -> calculo -> lista.

Resumen de la unidad

Llegamos al final de la unidad y es momento de amarrar todo. Empezamos con una idea: tus componentes eran fotografías estaticas y esta unidad los convirtio en películas. Las tres herramientas que lo hicieron posible fueron el renderizado de listas, los condicionales en JSX y el hook useState. Cada una resuelve una necesidad distinta, pero juntas resuelven casi cualquier interfaz interactiva.

El renderizado de listas toma un arreglo de datos y lo convierte en elementos de pantalla con el método map. Por cada elemento del arreglo devuelves un elemento JSX, y React los pinta uno tras otro. La lección clave aquí es la prop key: cada elemento de la lista necesita un identificador único y estable para que React pueda reconciliar la lista anterior con la nueva sin confundirse. Los ids de tus datos son las mejores keys; el índice del arreglo solo sirve para listas totalmente estaticas, porque en listas que cambian provoca errores visuales como marcas que se mueven al elemento equivocado.

Los condicionales en JSX te permiten mostrar una cosa u otra según una condición. Como dentro de las llaves solo caben expresiones, usas el ternario para elegir entre dos alternativas, el operador && para mostrar algo o nada, y sacas la lógica a un if con variable antes del return cuando hay varias ramas. El switch es ideal cuando una variable puede tomar varios valores conocidos, y los returns tempranos sirven para descartar casos de carga o error antes del contenido principal. La trampa a recordar: con &&, asegurate de que la condición sea booleana, porque un 0 a la izquierda se pinta como texto en la pantalla.

El hook `useState` le da memoria a tus componentes. Con `const [valor, setValor] = useState(inicial)` creas una casilla de memoria que sobrevive entre pintados, y cada vez que llamas a la función `set`, React guarda el valor nuevo y vuelve a pintar el componente. Aprendiste que una variable normal no sirve porque no avisa a React ni sobrevive a los re-pintados. Aprendiste que las actualizaciones se procesan en fila, así que para valores que dependen del anterior debes usar la forma funcional `setValor((v) => v + 1)`. Y aprendiste la inmutabilidad: nunca mutes el estado directamente; crea copias nuevas con `spread`, `map` o `filter` para que React detecte el cambio.

Los errores más comunes quedaron identificados: mutar el estado, olvidar la `key`, usar índices como `key` en listas dinámicas, leer el estado justo después de setearlo, acumular `set` directos en lugar de la forma funcional, meter sentencias `if` dentro del `JSX` y el 0 fantasma del `&&`. Si interiorizas está lista, te ahorrarás horas de depuración.

En los ejemplos construiste una lista de frutas con `keys`, un semáforo con condicionales, un contador con estado y una lista de tareas que combina las tres herramientas. En la práctica resolviste una galería de productos, un interruptor, un contador con avisos, un `input` en vivo y una lista de invitados con agregar y eliminar. Cada ejercicio reforzó un patrón que verás en aplicaciones reales: carritos, comentarios, formularios y paneles usan exactamente estas estructuras.

El mensaje final es este: ya no dependes de interfaces congeladas. Sabes tomar datos y pintarlos como listas, sabes decidir que mostrar según condiciones y sabes guardar información que cambia con el tiempo. Con `props` de la unidad anterior y el `trio` de esta unidad, tienes el vocabulario completo para construir componentes que responden al usuario. En la siguiente unidad seguiras sumando herramientas, pero la base interactiva ya la tienes. Sigue practicando, rompe el código a propósito para ver qué pasa y confía en el proceso: así se aprende React de verdad.

UNIDAD 4: Eventos e interactividad

Hay un momento en el aprendizaje de todo programador en el que la pantalla deja de ser un papel y se convierte en una conversación. Hasta ahora, en las unidades anteriores de este curso, tus componentes de React eran como carteles pegados en una pared: bonitos, bien organizados, con datos que cambiaban si tu modificabas el código, pero mudos. Les dabas clic y no pasaba nada. Escribías dentro de un campo de texto y nadie escuchaba. El usuario miraba, pero no podía tocar. En esta unidad eso cambia para siempre, porque vas a aprender a darle oídos y manos a tus componentes: los eventos.

Piensa en como nació la web. En los años noventa, una página web era básicamente un documento: texto, algunas imágenes y enlaces azules que te llevaban a otro documento. Era como una enciclopedia gigante repartida en computadoras de todo el mundo. Nadie esperaba que una página "hiciera" algo; su trabajo era mostrar información y punto. Si querías interactuar con algo, llenabas un formulario, le dabas al botón de enviar, la página se recargaba entera y el servidor te devolvía otro documento. Todo el trabajo pesado ocurría lejos de ti, en una computadora que nunca veías.

Después la web creció y se volvió ambiciosa. La gente quiso hacer en el navegador lo que antes solo se podía hacer en programas instalados: escribir documentos, editar fotos, chatear en tiempo real, jugar, comprar con un par de clics, ver videos, llevar su contabilidad. Para lograrlo, las páginas tuvieron que dejar de comportarse como documentos y empezar a comportarse como aplicaciones. Y hay una sola cosa que distingue a una aplicación de un documento: la interactividad. Una aplicación escucha lo que haces y responde al instante, sin recargar nada, sin pedirle permiso al servidor para cada movimiento.

Esa respuesta inmediata es exactamente lo que vas a construir en esta unidad. Cuando termines, tus componentes ya no serán carteles mudos: serán máquinas que reaccionan. Un botón que suma, un campo de texto que se refleja en pantalla mientras escribes, un formulario que saluda por tu nombre, un interruptor que cambia toda la página de tema claro a tema oscuro. Todo eso nace de una idea muy simple que vas a dominar a fondo: el usuario hace algo, ese algo dispara un evento, el evento cambia el estado, y el cambio de estado vuelve a dibujar la pantalla.

Fíjate bien en esa cadena porque es el corazón de todo React: evento -> cambio de estado -> nuevo renderizado -> interfaz actualizada. En la unidad 3 aprendiste la segunda mitad de la cadena: como el estado (`useState`) hace que la pantalla se vuelva a dibujar sola. Ahora aprenderás la primera mitad: como capturar lo que el usuario hace (clics, teclas, movimientos, envíos de formulario) y convertirlo en cambios de estado. Cuando unas las dos mitades,

tendrás el ciclo completo de interactividad, y con ese ciclo puedes construir prácticamente cualquier cosa que veas en la web moderna.

Los tres temas de esta unidad están encadenados como eslabones. Primero verás los eventos y sus manejadores: como decirle a React "cuando el usuario haga clic aquí, ejecuta esta función", por qué en React se escribe `onClick` con mayúscula intermedia y no `onclick` como en el HTML de antes, y que es ese objeto misterioso llamado "e" que aparece en los manejadores. Después verás los inputs controlados, que es la técnica para que React tome el mandó total de lo que el usuario escribe: el estado se vuelve la única fuente de verdad y el campo de texto simplemente obedece. Y todo el tiempo estaremos regresando al ciclo completo, para que no solo memorices sintaxis sino que entiendas el mecanismo que mueve cada aplicación de React que existe.

Al terminar esta unidad dominaras: el sistema de eventos de React y su escritura camelCase; la diferencia entre pasar una función y llamarla (uno de los errores más comunes de todo principiante); como pasar argumentos a los manejadores; cómo leer el objeto del evento y para que sirve `preventDefault`; los inputs controlados con `value` y `onChange`, incluyendo `textarea`, `select` y `checkbox`; la diferencia entre un input controlado y uno no controlado; el ciclo completo evento -> estado -> render; los errores típicos que te haran tropezar (y como levantarte rápido); y las buenas prácticas para escribir manejadores limpios y profesionales.

Una advertencia honesta antes de empezar: esta es la unidad donde más se equivoca la gente, pero también donde más se divierte. Te vas a equivocar escribiendo `onClick={miFuncion()}` en lugar de `onClick={miFuncion}`, se te va a olvidar el `preventDefault` y tu formulario va a recargar la página, vas a intentar leer el estado un instante después de cambiarlo y no te va a dar el valor nuevo. Todo eso es normal. Cada error de esta unidad es una lección disfrazada, y aquí los vamos a provocar a proposito para que los entiendas de una vez por todas. Respira, abre tu editor, y vamos a darle vida a tus componentes.

Hay algo que conviene dejar claro antes de entrar en materia: la interactividad no es un adorno, es una conversación. Cada vez que un usuario hace clic, escribe una letra o mueve el puntero sobre tu página, te está haciendo una pregunta silenciosa: "¿qué pasa si toco aquí?". Cada respuesta inmediata y correcta construye confianza; cada botón muerto, cada recarga inesperada y cada campo que no responde la destruye un poco. Las aplicaciones que la gente ama usar (el chat donde los mensajes aparecen al instante, el mapa que se desliza bajo tu dedo, el buscador que adivina lo que quieres escribir) no son mágicas: son conversaciones rápidas y bien diseñadas entre el usuario y el estado. Y tu trabajo como programador de React es exactamente ese: diseñar conversaciones. Decidir qué escucha cada componente, cómo reacciona a lo que escucha, y qué le muestra al usuario como respuesta. Un buen componente interactivo se siente vivo porque responde a todo; uno malo se siente roto aunque su código "funcione".

En esta unidad aprenderás el vocabulario completo de esa conversación, en tres bloques. Primero, los eventos y sus manejadores: cómo escuchar clics, teclas y envíos de formulario, por qué se escriben en camelCase, y cómo evitar la trampa de los paréntesis que ha hecho tropezar a generaciones enteras de programadores. Segundo, los inputs controlados: la técnica para que React tome el mando absoluto de lo que el usuario escribe, con `value` y `onChange` como pareja inseparable, aplicada a campos de texto, áreas de texto, listas desplegables y casillas de verificación. Y tercero, el ciclo completo que une ambas mitades: evento, cambio de estado, nuevo dibujo, interfaz actualizada. Al final te esperan ejemplos explicados línea por línea y ejercicios con solución, para que la teoría se vuelva memoria muscular.

Explicación detallada

El sistema de eventos de React

Empecemos por lo más visible: la forma en que se escriben los eventos. Si alguna vez programaste JavaScript puro o viste HTML antiguo, probablemente recuerdes algo como esto: un botón con `onclick="saludar()"` todo en minúsculas. En React, en cambio, escribes `onClick` con la C mayúscula. No es un capricho ni una moda: hay una razón técnica detrás, y entenderla te va a ahorrar muchas confusiones.

JSX no es HTML, aunque se le parezca muchísimo. Cuando escribes `<button onClick={saludar}>`, eso que parece un atributo HTML en realidad es una propiedad de un objeto de JavaScript. React toma tu JSX y lo convierte en llamadas a funciones que crean objetos, y esos objetos tienen propiedades con nombres. En JavaScript, por convención, las

propiedades de los objetos se escriben en camelCase: primera palabra en minúsculas, las siguientes con mayúscula inicial. Por eso `onClick`, `onChange`, `onSubmit`, `onKeyDown`, `onMouseOver`. Si escribieras `onclick` en minúsculas dentro de JSX, React lo trataría como un atributo HTML común y corriente, no como un evento de su sistema, y tu función jamás se ejecutaría. Es un error silencioso y traicionero: no sale ningún mensaje de error, simplemente el botón no hace nada y tu te quedas mirando la pantalla preguntandote que hiciste mal. Ahora ya lo sabes: en React, los eventos siempre van en camelCase.

La segunda cosa importante es que React no te entrega el evento del navegador tal como viene de fábrica. Lo envuelve en su propio objeto, al que la documentación llama "evento sintético" (`SyntheticEvent`). Piensa en él como un traductor simultáneo: los navegadores no siempre describen los eventos de la misma forma (Chrome dice una cosa, Firefox otra, Safari otra), así que React pone un intérprete en medio que te entrega siempre el mismo formato, sin importar donde se abra tu aplicación. Para ti, en la práctica, esto significa que puedes confiar en que `e.target`, `e.type` o `e.preventDefault` van a existir y a comportarse igual en todos lados. Es una de esas ayudas invisibles que React te da sin pedir nada a cambio.

La lista de eventos que vas a usar el noventa por ciento del tiempo cabe en una mano: `onClick` para clics en botones y elementos; `onChange` para cuando cambia el valor de un input, un select o un checkbox; `onSubmit` para cuando se envía un formulario; `onKeyDown` y `onKeyUp` para teclas presionadas y liberadas; `onMouseOver` y `onMouseOut` para cuando el puntero entra y sale de un elemento; `onFocus` y `onBlur` para cuando un campo gana o pierde el foco. Todos siguen la misma lógica: "on" + el nombre del evento en camelCase. Si ya conoces los eventos del JavaScript tradicional, solo tienes que cambiarles la capitalización y pasarles funciones en lugar de texto.

Hay un detalle elegante en cómo React maneja los eventos por dentro: no le pone un escuchador a cada botón de tu página. En su lugar, pone unos cuantos escuchadores en la raíz de tu aplicación y desde ahí reparte los eventos a quien corresponda, como una recepcionista de un edificio que recibe todos los paquetes y los distribuye a cada oficina. Esto se llama delegación de eventos y hace que tu aplicación sea más rápida y use menos memoria, sobre todo cuando tienes listas con cientos de elementos. Tu no tienes que hacer nada para aprovecharlo: ocurre automáticamente. Solo es bonito saber que React está cuidando el rendimiento mientras tu te concentras en escribir manejadores.

Manejadores de eventos: el arte de decir "cuando pase esto, haz aquello"

Un manejador de eventos (event handler) no es más que una función común y corriente de JavaScript a la que le das un trabajo: "quédate quieta aquí, y cuando el usuario haga clic, corre". La gracia está en cómo se la entregas a React. Mira estas dos formas de escribir lo mismo:

```
function App() {  
  function saludar() {  
    alert("Hola, bienvenido a Lucas EGA Academy");  
  }  
  
  return (  
    <button onClick={saludar}>  
      Saludame  
    </button>  
  );  
}
```

Aquí saludar es una función nombrada: tiene nombre, vive dentro del componente, y se la pasamos a onClick entre llaves, sin paréntesis. React la guarda y la ejecuta cada vez que alguien hace clic en el botón. Esta es la forma recomendada cuando el manejador tiene más de una línea o cuando lo vas a reutilizar: el código queda limpio, legible y fácil de probar.

La otra forma es la función flecha escrita directamente en el JSX, que verás muchísimo en ejemplos cortos:

```
function App() {  
  return (  
    <button onClick={() => alert("Hola, bienvenido")}>  
      Saludame  
    </button>  
  );  
}
```

Funciona igual, pero tiene un costo: cada vez que el componente se vuelve a dibujar, se crea una función nueva. Para botones y ejemplos pequeños no importa nada, pero si la lógica crece, ese manejador inline se vuelve difícil de leer. La regla práctica es sencilla: si cabe en una línea y solo se usa ahí, la flecha inline está bien; si hace más de una cosa, dale un nombre y sacala del JSX. Tu yo del futuro, depurando código a las dos de la mañana, te lo va a agradecer.

Ahora, el error más famoso de todos los principiantes de React, el que todo el mundo comete al menos una vez: escribir onClick={saludar()} con paréntesis en lugar de onClick={saludar} sin

paréntesis. Parecen casi idénticos, pero hacen cosas radicalmente distintas. Cuando escribes `onClick={saludar}`, le estás pasando a React la función misma, como quien le entrega a alguien una receta escrita en un papel: "guardala y usala cuando la necesites". Cuando escribes `onClick={saludar()}`, con los paréntesis, estás LLAMANDO a la función en ese mismo instante, durante el dibujado del componente, y le pasas a `onClick` lo que la función devuelva (que normalmente es `undefined`, o sea, nada). El resultado es tragicómico: la alerta aparece sola en cuanto carga la página, sin que nadie haya hecho clic, y después el botón queda muerto porque no tiene ningún manejador asignado. Grábatelo a fuego: en los eventos se pasa la función, no se llama a la función. Los paréntesis son la diferencia entre entregar la receta y cocinar el platillo en ese momento.

Hay una excepción aparente que confunde a muchos: cuando necesitas pasarle argumentos a tu manejador. Supon que tienes una lista de frutas y quieres que cada botón salude con su fruta:

```
function App() {
  function saludar(fruta) {
    alert("Te gusta la " + fruta);
  }

  return (
    <div>
      <button onClick={() => saludar("manzana")}>
        Manzana
      </button>
      <button onClick={() => saludar("mango")}>
        Mango
      </button>
    </div>
  );
}
```

Aquí la flecha inline no es un capricho: es necesaria. Si escribieras `onClick={saludar("manzana")}`, estarías llamando a la función durante el dibujado (el error de los paréntesis otra vez). La flecha crea una función nueva que no hace nada hasta que ocurre el clic, y cuando el clic ocurre, esa función llama a `saludar` con el argumento correcto. Piensa en la flecha como un sobre: dentro del sobre va la instrucción "llama a saludar con manzana", y el sobre solo se abre cuando el usuario hace clic. Este patrón (flecha que envuelve una llamada con argumentos) lo vas a usar miles de veces en tu vida como programador de React.

Hablemos ahora del objeto del evento, esa "e" que aparece en tantos ejemplos. Cuando ocurre un evento, React llama a tu manejador y le pasa como argumento un objeto con toda la información de lo que paso: en que elemento ocurrió, que tipo de evento fue, que tecla se presiono, y más. Por convención se le llama e, event o evt. Mira cómo se usa para leer lo que el usuario escribió:

```
function App() {  
  function manejarCambio(e) {  
    console.log("Escribiste: " + e.target.value);  
    console.log("Tipo de evento: " + e.type);  
  }  
  
  return (  
    <input onChange={manejarCambio} placeholder="Escribe algo" />  
  );  
}
```

La propiedad más útil de todas es e.target: es el elemento del DOM donde ocurrió el evento, o sea, el input mismo. Y e.target.value es el texto que hay dentro del input en ese momento. Con esas dos piezas ya puedes construir espejos de texto, validaciones en vivo, buscadores con sugerencias y todo lo que verás en los ejemplos de esta unidad. Otras propiedades útiles: e.type te dice el tipo de evento ("click", "change", "submit"); e.key en eventos de teclado te dice que tecla se presiono; e.preventDefault() es un método que cancela el comportamiento por defecto del navegador.

Detengamonos en preventDefault porque te va a salvar de un dolor de cabeza clásico. Los navegadores tienen comportamientos automáticos: cuando envías un formulario, recargan la página; cuando haces clic en un enlace, navegan a otra dirección. En una aplicación de React, casi nunca quieres eso: si el formulario recarga la página, se pierde todo el estado y tu aplicación "se reinicia". La solución es llamar a e.preventDefault() al inicio de tu manejador de submit:


```
function App() {
  function manejarEnvio(e) {
    e.preventDefault();
    alert("Formulario enviado sin recargar la pagina");
  }

  return (
    <form onSubmit={manejarEnvio}>
      <input placeholder="Tu nombre" />
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Sin esa línea, el navegador haría su trabajo por defecto (recargar) y tu función de React quedaría interrumpida a la mitad. Con esa línea, tu tomas el control: la página no se recarga, el estado se conserva y tú decides que hacer con los datos. Cada vez que uses `onSubmit` en un formulario, `preventDefault` es prácticamente obligatorio. Anotalo como reflejo: veo un form, escribo `preventDefault`.

Un primo cercano de `preventDefault` es `e.stopPropagation()`, que detiene la propagación del evento hacia los elementos padres. Imagina un botón dentro de una tarjeta, y tanto el botón como la tarjeta tienen `onClick`: al hacer clic en el botón, se dispararían los dos manejadores (primero el del botón, luego el de la tarjeta, porque el evento "burbujea" hacia arriba). Si solo quieres que se ejecute el del botón, llamas a `e.stopPropagation()` dentro de su manejador. No lo necesitaras todos los días, pero cuando tengas elementos anidados con sus propios clics, te va a sacar de un apuro.

Más allá del clic: teclado, ratón y foco

El clic es el rey de los eventos, pero no gobierna solo. Los usuarios también escriben con el teclado, mueven el puntero y navegan entre campos con la tecla `Tab`, y cada una de esas acciones es un evento que puedes escuchar. Dominar estos eventos secundarios es lo que separa una aplicación "que funciona" de una que "se siente bien".

Los eventos de teclado (`onKeyDown`, `onKeyUp`) te dicen que tecla se presiono a traves de `e.key`. El uso más comun es reaccionar a la tecla `Enter` dentro de un campo, como en los chats donde `Enter` envia el mensaje. Mira este mini ejemplo de lista de tareas:

```

function App() {
  const [texto, setTexto] = useState("");
  const [tarefas, setTareas] = useState([]);

  function agregarConEnter(e) {
    if (e.key === "Enter" && texto !== "") {
      setTareas([...tarefas, texto]);
      setTexto("");
    }
  }

  return (
    <div>
      <input
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        onKeyDown={agregarConEnter}
        placeholder="Escribe una tarea y pulsa Enter"
      />
      <ul>
        {tarefas.map((t, i) => (
          <li key={i}>{t}</li>
        ))}
      </ul>
    </div>
  );
}

```

El manejador `agregarConEnter` se pasa sin paréntesis a `onKeyDown`, igual que con los clics. Dentro, `e.key === "Enter"` detecta la tecla específica; la condición `texto !== ""` evita agregar tareas vacías. La línea `setTareas([...tarefas, texto])` merece atención: el operador `spread` crea un arreglo NUEVO con las tareas viejas más la nueva, en lugar de modificar el arreglo original. Esto es importante porque en React no debes mutar el estado directamente: si hicieras `tareas.push(texto)`, estarías cambiando el arreglo original y React podría no detectar el cambio. Nuevo estado, nuevo arreglo, siempre. Por último, `setTexto("")` limpia el campo después de agregar, porque el input es controlado y obedece al estado.

Los eventos del puntero (`onMouseOver`, `onMouseOut`, `onMouseEnter`, `onMouseLeave`) te sirven para efectos visuales que dependen del estado, como resaltar una tarjeta cuando el puntero pasa sobre ella. Puedes lograr hover con CSS puro, pero cuando el efecto debe combinarlo con lógica (por ejemplo, mostrar un botón de "eliminar" solo sobre la tarjeta activa), hacerlo con estado es más limpio: un booleano activo que `onMouseOver` pone en true y `onMouseOut` en false, y el JSX decide que mostrar.

Los eventos de foco (`onFocus`, `onBlur`) ocurren cuando un campo gana o pierde el foco, o sea, cuando el usuario entra o sale de el (normalmente con Tab o con clic). Son perfectos para dos cosas: resaltar el campo activo y validar cuando el usuario termina de escribir. Un patrón muy comun es validar el campo en `onBlur` en lugar de en cada tecla, para no reganar al usuario mientras todavia esta escribiendo:

```
function App() {
  const [correo, setCorreo] = useState("");
  const [tocado, setTocado] = useState(false);

  const mostrarError = tocado && !correo.includes("@");

  return (
    <div>
      <input
        value={correo}
        onChange={(e) => setCorreo(e.target.value)}
        onBlur={() => setTocado(true)}
        placeholder="tu@correo.com"
      />
      {mostrarError && <p>Escribe un correo valido.</p>}
    </div>
  );
}
```

Aquí el estado `tocado` registra si el usuario ya salio del campo al menos una vez. El mensaje de error solo aparece si el campo fue tocado Y el correo no tiene arroba. Mientras el usuario escribe por primera vez, no lo molestas; cuando sale del campo con algo mal escrito, ahí si le avisas. Es un detalle pequeno de experiencia de usuario que hace que tus formularios se sientan profesionales.

Existe también `onDoubleClick` para el doble clic, aunque se usa poco porque muchos usuarios no descubren esa interacción por si solos. Mencion honorifica y seguimos.

El evento y los argumentos, juntos

Ya sabes pasar solo la función (`onClick={saludar}`), solo argumentos envueltos en flecha (`onClick={() => saludar("Ana")}`) y solo el evento (`onChange={manejarCambio}`). Pero a veces necesitas las tres cosas a la vez: el evento Y tus propios argumentos. El patrón es combinarlos: `onClick={(e) => borrar(e, id)}`. La flecha recibe el evento y se lo pasa a tu función junto con lo que necesites.

El caso típico es una lista donde cada elemento tiene su botón de eliminar, y además el elemento completo tiene su propio clic:

```
function App() {
  const [frutas, setFrutas] = useState(["manzana", "mango", "platano"]);

  function borrar(e, fruta) {
    e.stopPropagation();
    setFrutas(frutas.filter((f) => f !== fruta));
  }

  return (
    <ul>
      {frutas.map((fruta) => (
        <li key={fruta} onClick={() => alert("Elegiste " + fruta)}>
          {fruta}
          <button onClick={(e) => borrar(e, fruta)}>X</button>
        </li>
      ))}
    </ul>
  );
}
```

Desmenucemos lo que pasa al hacer clic en el botón X de "mango". El botón ejecuta la flecha, que llama a borrar pasándole el evento y la fruta "mango". Dentro de borrar, lo primero es `e.stopPropagation()`: sin esto, el clic "burbujearía" hacia el li padre y dispararía también su `onClick`, mostrando la alerta "Elegiste mango" justo cuando querías borrarlo. Después, `setFrutas(frutas.filter((f) => f !== fruta))` crea un arreglo nuevo sin la fruta borrada: `filter` devuelve un arreglo nuevo con solo los elementos que cumplen la condición, así que no mutas el estado original. React vuelve a dibujar y la fruta desaparece de la lista. Tres conceptos de la unidad trabajando juntos en seis líneas: argumentos con flecha, `stopPropagation` y actualización inmutable del estado. Cuando este ejemplo te parezca obvio, ya piensas en React.

Inputs controlados: cuando React toma el mandó del teclado

Hasta aquí, el usuario hacía clic y tu reaccionabas. Pero hay una forma de interacción mucho más íntima y continua: escribir. Cada vez que alguien teclea en un campo de texto, está alimentando a tu aplicación letra por letra, y tú tienes que decidir quién manda sobre ese campo: el navegador o React. La respuesta profesional es React, siempre React, y la técnica se llama "input controlado".

Para entender por qué, primero mira el problema. Un input de HTML tiene su propio estado interno: el navegador guarda lo que el usuario escribe dentro del elemento del DOM, sin pedirle permiso a nadie. Si además guardas ese texto en un estado de React, tienes DOS lugares con la misma información: el DOM y tu estado. Dos fuentes de verdad son una receta para el desastre: pueden desincronizarse, y cuando eso pasa, la pantalla muestra una cosa y tu lógica cree otra. Es como tener dos relojes que dan horas distintas: nunca sabes cuál es la correcta.

El input controlado resuelve esto con un trato muy simple entre React y el campo de texto, que se resume en dos props inseparables: `value` y `onChange`.

```
function App() {
  const [texto, setTexto] = useState("");

  return (
    <input
      value={texto}
      onChange={(e) => setTexto(e.target.value)}
      placeholder="Escribe tu nombre"
    />
  );
}
```

Fíjate en lo que está pasando, porque este es el patrón más importante de toda la unidad. La prop `value={texto}` le dice al input: "tu contenido visible será EXACTAMENTE lo que diga el estado `texto`, ni una letra más ni una menos". El input ya no decide por sí mismo que mostrar: obedece. Y la prop `onChange={(e) => setTexto(e.target.value)}` le dice: "cada vez que el usuario intente escribir algo, avísame y yo actualizaré el estado con lo que escribió". El ciclo completo de una tecla presionada es así: el usuario pulsa la letra "a", el navegador intenta ponerla en el campo y dispara `onChange`, tu manejador lee `e.target.value` (el texto con la "a" incluida) y llama a `setTexto`, React actualiza el estado y vuelve a dibujar el componente, y el input recibe el nuevo `value` y muestra la "a". Da una vuelta completa por React en cuestión de milisegundos, tan rápido que parece instantáneo.

Parece un rodeo innecesario (por qué no dejar que el input muestre la letra directamente, como hace el HTML normal), pero ese rodeo es poder puro. Como todo pasa por tu estado, puedes intervenir en el camino: convertir el texto a mayúsculas mientras se escribe, impedir que escriban números en un campo de nombre, contar los caracteres restantes de un tuit, validar el correo en vivo y pintar el borde de rojo si está mal, o deshabilitar el botón de enviar hasta que el formulario este completo. Nada de eso es posible si el navegador es el dueño del texto. Con el input controlado, React es la única fuente de verdad: hay un solo reloj, y es el tuyo.

Ahora comparemos con el input NO controlado, para que veas la diferencia con claridad. Un input no controlado es el de toda la vida: no le pasas value, el navegador guarda el texto por su cuenta, y tu lo lees solo cuando lo necesitas (por ejemplo, al enviar el formulario) usando una referencia o el evento del submit. Funciona, y para formularios sencillos de "llenar y enviar" es perfectamente válido. Pero pierdes todo lo de arriba: no puedes transformar el texto mientras se escribe, no puedes validarlo en vivo, no puedes mostrar un espejo de lo escrito en otra parte de la pantalla. La regla para decidir es esta: si necesitas REACCIONAR a lo que se escribe mientras se escribe, usa un input controlado; si solo necesitas LEERLO al final, un no controlado basta. En las aplicaciones reales, la mayoría de los inputs terminan siendo controlados.

Hay un error clásico que te va a pasar con los inputs controlados: poner value sin onChange. Si escribes `<input value={texto} />` sin manejador de cambio, el input queda congelado: muestra el valor del estado, el usuario intenta escribir, pero como nada actualiza el estado, React vuelve a dibujar el mismo valor una y otra vez y las letras "rebotan". React incluso te muestra una advertencia en la consola diciendote que un campo con value necesita un onChange. Es su forma de decirte: "oye, me pediste que yo mande, pero no me diste forma de enterarme de los cambios". value y onChange son pareja inseparable: uno sin el otro no funciona.

Otro tropiezo común es intentar "arreglar" el input mutando `e.target.value` directamente dentro del manejador, algo como `e.target.value = e.target.value.toUpperCase()`. Parece que funciona, pero estas peleando contra React: modificas el DOM a sus espaldas y en el siguiente renderizado React puede sobrescribir tu cambio con el valor del estado, que no se enteró de la mutación. La forma correcta es transformar el dato ANTES de guardarlo en el estado: `setTexto(e.target.value.toUpperCase())`. El estado recibe el texto ya transformado, el renderizado lo muestra, y todo el mundo está de acuerdo. Regla de oro: nunca toques el DOM directamente cuando React está a cargo; habla siempre a través del estado.

Los inputs controlados no son solo para campos de texto. El textarea en React también se controla con value, con una diferencia respecto al HTML tradicional: en HTML el texto inicial va

entre las etiquetas, pero en React va en la prop value. Si pones texto entre las etiquetas de un textarea en React, te va a dar error. Es una de esas pequeñas diferencias que conviene memorizar.

El select controlado es igual de lógico: en lugar de marcar la opción elegida con selected en cada option (como en HTML), pones value en el select y React se encarga de mostrar la opción correspondiente:

```
function App() {
  const [fruta, setFruta] = useState("mango");

  return (
    <select value={fruta} onChange={(e) => setFruta(e.target.value)}>
      <option value="manzana">Manzana</option>
      <option value="mango">Mango</option>
      <option value="platano">Platano</option>
    </select>
  );
}
```

Aquí el estado fruta dice "mango", así que el select muestra "Mango" seleccionado. Cuando el usuario elige otra opción, onChange actualiza el estado y el select se vuelve a dibujar con la nueva selección. La única fuente de verdad, otra vez: si en algún momento tu lógica decide que la fruta debe ser "platano" (por ejemplo, porque se acabaron los mangos), basta con hacer setFruta("platano") y el select lo refleja solo. Intenta hacer eso manipulando options a mano en JavaScript puro y verás por qué React enamora.

Los checkbox (y los radio buttons) tienen su propia versión del trato: en lugar de value usan checked, porque lo que importa no es un texto sino un si o no. Y en el onChange se lee e.target.checked en lugar de e.target.value:

```
function App() {  
  const [acepto, setAcepto] = useState(false);  
  
  return (  
    <label>  
      <input  
        type="checkbox"  
        checked={acepto}  
        onChange={(e) => setAcepto(e.target.checked)}  
      />  
      Acepto los terminos  
    </label>  
  );  
}
```

El estado `acepto` es un booleano: `true` o `false`. `checked={acepto}` hace que la palomita del checkbox refleje exactamente ese booleano, y `onChange` lo actualiza con `e.target.checked` cada vez que el usuario lo marca o desmarca. Con esto puedes habilitar un botón solo cuando el checkbox este marcado, que es el típico "acepto los terminos y condiciones" de los formularios reales.

Cuando un formulario tiene varios campos, no necesitas un `useState` por cada uno (aunque para empezar está bien). El patrón profesional es un solo objeto en el estado y un manejador generico que usa el atributo `name` de cada campo para saber cual actualizar:


```

function App() {
  const [datos, setDatos] = useState({ nombre: "", correo: "" });

  function manejarCambio(e) {
    setDatos({
      ...datos,
      [e.target.name]: e.target.value
    });
  }

  return (
    <form>
      <input
        name="nombre"
        value={datos.nombre}
        onChange={manejarCambio}
        placeholder="Nombre"
      />
      <input
        name="correo"
        value={datos.correo}
        onChange={manejarCambio}
        placeholder="Correo"
      />
    </form>
  );
}

```

Hay dos trucos aquí que vale la pena desmenuzar. Primero, `...datos` (el operador spread) copia todas las propiedades actuales del objeto, para no perder los campos que no cambiaron; si lo omitieras, cada tecla borraría los demás campos. Segundo, `[e.target.name]` con corchetes es una "propiedad computada": le dice a JavaScript "usa como nombre de propiedad el valor que traiga `e.target.name`". Si el campo que cambió tiene `name="correo"`, la expresión se convierte en un objeto con la propiedad `correo` actualizada. Un solo manejador para todos los campos del formulario: eso es código de profesional. El único requisito es no olvidar el atributo `name` en cada `input`, porque sin él, `e.target.name` sería `undefined` y tu estado se llenaría de basura.

El ciclo completo de interactividad: la máquina que ya tienes en las manos

Ya tienes todas las piezas sobre la mesa: eventos que escuchan, manejadores que reaccionan, estados que guardan, y renderizados que dibujan. Ahora vamos a armar la máquina completa y verla funcionar vuelta por vuelta. Este ciclo es lo que ocurre CADA VEZ

que un usuario interactúa con CUALQUIER aplicación de React en el mundo, desde un simple contador hasta el panel de control de un banco. Si entiendes este ciclo, entiendes React.

Tomemos el ejemplo canónico, el contador:

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

  return (
    <div>
      <p>Contador: {num}</p>
      <button onClick={() => setNum(num + 1)}>
        Sumar uno
      </button>
    </div>
  );
}
```

Sigue la vuelta completa conmigo, paso por paso. Paso uno: el usuario hace clic en el botón. El navegador detecta el clic y React, con su sistema de eventos, lo envuelve en su evento sintético y busca que manejador le corresponde. Paso dos: React ejecuta la función flecha que le pasaste a `onClick`. Esa función llama a `setNum(num + 1)`. En ese momento `num` vale 0, así que le pides a React que el nuevo estado sea 1. Ojo: `setNum` no cambia la variable `num` en ese instante; lo que hace es programar la actualización, como quien deja una nota que dice "para la próxima vuelta, `num` vale 1". Paso tres: React marca el componente como "sucio" (algo cambió) y lo vuelve a dibujar: ejecuta la función `App` otra vez, pero ahora `useState` devuelve 1 en lugar de 0. Paso cuatro: React compara el nuevo dibujo con el anterior (esto es la famosa reconciliación con el DOM virtual) y detecta que solo cambió el texto del párrafo, de "Contador: 0" a "Contador: 1". Paso cinco: actualiza únicamente ese pedacito del DOM real, sin tocar nada más. El usuario ve el número cambiar. Fin de la vuelta. Y todo esto ocurre en milisegundos.

Fíjate en lo que TU no hiciste: no buscaste el párrafo en el DOM, no le cambiaste el texto a mano, no le sumaste nada a ningún elemento. Tu único trabajo fue describir dos cosas: cómo se ve la interfaz para cada valor del estado, y que cambió de estado provoca cada evento. React se encarga del resto. Esta es la diferencia fundamental entre programación imperativa ("busca este elemento y cambiale el texto") y programación declarativa ("la pantalla siempre refleja el estado"). React es declarativo: tu declaras la relación entre estado e interfaz, y él se encarga de mantenerla. Cuando algo se ve mal en pantalla, la pregunta correcta casi nunca es "¿qué le paso al DOM?", sino "¿qué valor tiene mi estado?". El estado manda; la pantalla obedece.

Hay una analogía que me gusta mucho para este ciclo: un restaurante. El usuario es el cliente que hace un pedido (el evento: "quiero un taco más"). El manejador es el mesero que toma el pedido y lo lleva a la cocina (setNum). La cocina es React, que prepara el nuevo platillo (el nuevo renderizado con el estado actualizado). Y el plato servido en la mesa es la interfaz actualizada que ve el cliente. El cliente nunca entra a la cocina a cocinar su propio taco (eso sería manipular el DOM directamente); hace su pedido y confía en el sistema. Y el sistema funciona porque hay un solo libro de pedidos: el estado.

Otra analogía útil, más eléctrica: el interruptor de la luz. El estado es si el foco está prendido o apagado (un booleano). El evento es tu dedo moviendo el interruptor (onClick). El manejador invierte el valor (setPrendido(!prendido)). El renderizado es el cuarto que se ilumina o se oscurece. Nadie piensa en "ir a cambiar los fotones del cuarto": cambias el interruptor y la consecuencia ocurre sola. Así se programa en React: cambias el estado y la consecuencia visual ocurre sola.

Y una tercera, para los inputs controlados: el espejo. Un espejo no tiene cara propia; refleja exactamente lo que tiene enfrente. Un input controlado es el espejo de tu estado: muestra exactamente lo que el estado dice. Si el estado dice "hola", el input muestra "hola". Si tu lógica decide que el estado debe decir "HOLA", el input lo refleja al instante. Y cuando el usuario intenta escribir, en realidad le está pidiendo al estado que cambie, a través del onChange. El espejo nunca miente porque no tiene opinión propia.

Errores comunes: tropezones con moraleja

Vamos a provocar a propósito los errores más comunes de esta unidad, para que cuando te pasen (porque te van a pasar), los reconozcas en segundos en lugar de perder una tarde.

Error 1: llamar la función en vez de pasarla. Ya lo vimos, pero vale repetirlo porque es el número uno mundial: `onClick={saludar()}` ejecuta la función durante el dibujo y deja el botón muerto; `onClick={saludar}` la pasa correctamente. Sintoma: algo pasa solo, sin que hagas clic, y después el botón no responde. Cura: quita los paréntesis. Si necesitas argumentos, envuelve en flecha: `onClick={() => saludar("Ana")}`.

Error 2: olvidar `e.preventDefault()` en el `onSubmit`. Sintoma: llenas el formulario, le das a enviar, la página parpadea y se recarga, y todo tu estado se pierde. Cura: primera línea del manejador de submit, siempre: `e.preventDefault()`. Hazlo un reflejo muscular.

Error 3: value sin onChange (el input congelado). Sintoma: el campo muestra el valor inicial pero no te deja escribir nada, y la consola muestra una advertencia. Cura: agrega el onChange que actualiza el estado. Recuerda: value y onChange son pareja.

Error 4: mutar `e.target.value` directamente. Sintoma: el "arreglo" funciona una vez y luego React lo sobrescribe, o el estado y la pantalla se desincronizan misteriosamente. Cura: transforma el dato antes de guardarlo: `setTexto(e.target.value.toUpperCase())`. Nunca pelees con React por el control del DOM; él siempre gana.

Error 5: leer el estado inmediatamente después de `setState` esperando el valor nuevo. Por ejemplo: `setNum(num + 1); console.log(num);` y el console muestra el valor VIEJO. Esto confunde muchísimo, pero no es un error de eventos: es que `setState` es asíncrono, programa la actualización para la siguiente vuelta del ciclo, no la aplica al instante. La variable `num` dentro de ese manejador seguirá valiendo lo viejo hasta el próximo renderizado. Cura: si necesitas el valor nuevo, calculalo en una variable primero: `const nuevo = num + 1; setNum(nuevo); console.log(nuevo)`. (Este tema se profundiza en la unidad de estados, pero necesitabas saberlo desde ahora para no volverte loco.)

Error 6: olvidar el atributo `name` cuando usas un manejador genérico para varios campos. Sintoma: escribes en un campo y el estado se llena con una propiedad llamada "undefined", o los campos se borran entre sí. Cura: cada `input` controlado de un formulario con manejador genérico debe tener su `name` único, y ese `name` debe coincidir con la propiedad del objeto en el estado.

Error 7: lógica pesada o efectos secundarios dentro del manejador inline del JSX. Por ejemplo, un `onClick` con diez líneas que validan, transforman, guardan y navegan. Sintoma: el JSX se vuelve ilegible y el mismo código se repite en varios botones. Cura: extrae la lógica a una función nombrada fuera del JSX y pásala por referencia.

Buenas prácticas: escribe como profesional desde el día uno

Primera: nombra tus manejadores con el prefijo "handle" seguido de lo que manejan: `handleClick`, `handleChange`, `handleSubmit`, `handleKeyDown`. Es una convención universal en el mundo React: cualquier programador que lea `handleSubmit` sabe al instante que es el manejador del envío del formulario, sin tener que leer su contenido. Los nombres son documentación gratis.

Segunda: manten los manejadores enfocados en una sola responsabilidad. `handleSubmit` valida y envía; no debería también cambiar el tema de la página y reproducir un sonido. Si un manejador crece, divídelo en funciones pequeñas con nombres claros. El código se lee de arriba hacia abajo como una historia: "al enviar, válido; si es válido, guardó; luego limpio".

Tercera: no pongas lógica pesada directamente en el JSX. La flecha inline es para cosas de una línea: `onClick={() => setNum(num + 1)}`. En cuanto necesites un if, un calculo o más de una instrucción, crea una función nombrada. Tu JSX debe leerse como el índice de un libro: títulos claros, sin párrafos enteros.

Cuarta: modela el estado mínimo necesario. Si un valor se puede calcular a partir del estado (por ejemplo, "el formulario esta completo" se deduce de los campos), no lo guardes en otro estado: calculalo durante el renderizado. Cada estado duplicado es una futura desincronización esperando a ocurrir. Una sola fuente de verdad, siempre.

Quinta: usa el estado para controlar la interfaz, no solo los datos. Un botón deshabilitado (`disabled={!acepto}`), un mensaje de error que aparece y desaparece, un tema claro u oscuro: todo eso son estados booleanos o de texto que se reflejan en props. Cuando dudes de cómo hacer que algo "se prenda o se apague" en pantalla, la respuesta casi siempre es un `useState`.

Sexta: inicializa tus estados con valores del tipo correcto. Un contador empieza en 0 (número), un texto en "" (cadena vacía), un checkbox en false (booleano), una selección en el valor de la opción por defecto. Esto evita errores raros como inputs que empiezan como undefined y React se queja de que cambian de "no controlado" a "controlado".

Septima: escribe manejadores accesibles. Un `<div onClick={...}>` funciona con el ratón, pero las personas que navegan con el teclado no pueden activarlo con Enter o Espacio, y los lectores de pantalla no lo anuncian como un botón. Si algo se comporta como un botón, que SEA un botón: usa la etiqueta `<button>` y obtienes gratis el foco con Tab, la activación con Enter y Espacio, y la semantica correcta para tecnologías de apoyo. Reserva los `onClick` en divs para casos genuinos (como una tarjeta completa clicable) y aun ahí piensa en como la usara alguien sin ratón. El código profesional no solo funciona: funciona para todos.

Ejemplos paso a paso

La teoría ya está en tu cabeza; ahora vamos a ensuciarnos las manos. Cada ejemplo viene completo, listo para copiar y pegar, y explicado línea por línea para que no quede ni una duda. Te recomiendo escribirlos tú mismo en lugar de solo leerlos: la memoria muscular también programa.

Ejemplo 1: Contador con botones (sumar, restar y reiniciar)

Este es el "hola mundo" de la interactividad. Un número en pantalla y tres botones que lo modifican. Parece trivial, pero contiene el ciclo completo: evento -> estado -> render.

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

  function sumar() {
    setNum(num + 1);
  }

  function restar() {
    setNum(num - 1);
  }

  function reiniciar() {
    setNum(0);
  }

  return (
    <div>
      <h2>Contador: {num}</h2>
      <button onClick={sumar}>+ Sumar</button>
      <button onClick={restar}>- Restar</button>
      <button onClick={reiniciar}>Reiniciar</button>
    </div>
  );
}
```

Línea por línea: `const [num, setNum] = useState(0)` crea el estado `num` que empieza en 0, con su función actualizadora `setNum`. Las tres funciones (`sumar`, `restar`, `reiniciar`) son manejadores nombrados: cada uno pide un nuevo valor del estado. Fíjate que `sumar` usa el valor actual de `num` para calcular el siguiente (`num + 1`); esto funciona porque en cada renderizado `num` tiene el valor fresco. En el JSX, el `h2` muestra el estado entre llaves: `{num}`. Cada botón recibe su manejador SIN paréntesis: `onClick={sumar}`. Si pusieras `onClick={sumar()}`, el contador se volvería loco sumando solo durante el dibujado.

Qué pasa cuando haces clic en "+ Sumar": React ejecuta `sumar`, que llama a `setNum(0 + 1)`; React programa el nuevo estado (1) y vuelve a dibujar; el `h2` ahora muestra "Contador: 1". Cada clic es una vuelta completa del ciclo. Prueba mental: haz clic tres veces en `sumar` y una en `restar`. ¿Qué muestra la pantalla? Si dijiste 2, ya entendiste el ciclo.

Ejemplo 2: Botón que cambia de color

Aquí el estado no es un número sino un texto, y lo que cambia no es un párrafo sino el estilo del propio botón. Esto te enseña que el estado puede controlar CUALQUIER prop, incluyendo los estilos.

```
function App() {
  const [color, setColor] = useState("gris");

  function cambiarColor() {
    if (color === "gris") {
      setColor("verde");
    } else {
      setColor("gris");
    }
  }

  return (
    <button
      onClick={cambiarColor}
      style={{ backgroundColor: color, padding: "10px" }}
    >
      Color actual: {color} (clic para cambiar)
    </button>
  );
}
```

Línea por línea: el estado color empieza en "gris". La función cambiarColor revisa el valor actual: si es "gris", pide "verde"; si no, pide "gris". Es un interruptor de dos posiciones, como el de la luz. En el JSX, el botón recibe onClick={cambiarColor} y, lo interesante, style={{ backgroundColor: color }}. Esas dobles llaves confunden al principio: las de afuera son las llaves de JSX que dicen "aquí va JavaScript", y las de adentro son las de un objeto de JavaScript con la propiedad backgroundColor. El valor de esa propiedad es el estado color, así que cada vez que el estado cambia, el fondo del botón cambia solo. El texto del botón también muestra el estado: {color}.

Que aprendiste: el estado no solo guarda datos para mostrar; también controla apariencia y comportamiento. Este mismo patrón (un estado que alterna entre dos valores) es el que usaras para temas claro/oscuro, para mostrar u ocultar menus, y para pestañas de navegación. Domina el interruptor y dominas media web.

Ejemplo 3: Espejo de texto en vivo

El ejemplo estrella de los inputs controlados: escribes en un campo y el texto se refleja instantáneamente en otro lugar de la pantalla. Es la base de los buscadores con sugerencias, los chats y los editores.

```
function App() {
  const [texto, setTexto] = useState("");

  return (
    <div>
      <input
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        placeholder="Escribe algo aqui"
      />
      <p>Estas escribiendo: {texto}</p>
      <p>Tu texto tiene {texto.length} letras.</p>
    </div>
  );
}
```

Línea por línea: el estado texto empieza como cadena vacía. El input es controlado: `value={texto}` lo obliga a mostrar exactamente el estado, y `onChange={(e) => setTexto(e.target.value)}` actualiza el estado con cada tecla. Como el estado es la única fuente de verdad, los dos párrafos de abajo siempre están sincronizados con el campo: el primero repite lo escrito (el espejo) y el segundo calcula `texto.length` en cada renderizado (un valor derivado, sin necesidad de otro estado).

Sigue una tecla conmigo: el estado vale "hol" y pulsas "a". El navegador dispara `onChange` con `e.target.value` igual a "hola". El manejador llama a `setTexto("hola")`. React vuelve a dibujar: el input recibe `value="hola"`, el primer párrafo muestra "Estas escribiendo: hola" y el segundo "Tu texto tiene 4 letras". Una tecla, una vuelta completa del ciclo, tres lugares actualizados a la vez. Eso es el poder de la fuente única de verdad: un solo cambió de estado, y toda la interfaz que depende de el se pone al día sola.

Ejemplo 4: Selector de opciones con select controlado

Un caso real de formulario: el usuario elige su curso favorito de una lista desplegable y la página responde con un mensaje personalizado. Incluye un botón de confirmación para redondear el flujo.


```

function App() {
  const [curso, setCurso] = useState("react");
  const [confirmado, setConfirmado] = useState("");

  function confirmar() {
    setConfirmado(curso);
  }

  return (
    <div>
      <label>Elige tu curso favorito: </label>
      <select value={curso} onChange={(e) => setCurso(e.target.value)}>
        <option value="react">React</option>
        <option value="javascript">JavaScript</option>
        <option value="python">Python</option>
        <option value="html">HTML5</option>
      </select>
      <button onClick={confirmar}>Confirmar eleccion</button>
      {confirmado !== "" && (
        <p>Te inscribiste al curso de {confirmado}. Buena eleccion.</p>
      )}
    </div>
  );
}

```

Línea por línea: hay DOS estados con dos responsabilidades distintas: `curso` guarda la selección actual del select (empieza en "react"), y `confirmado` guarda la elección ya confirmada con el botón (empieza como cadena vacía, o sea, "nada confirmado todavía"). El select es controlado: `value={curso}` y `onChange` actualiza `curso`. Cada option tiene un `value` interno (el que se guarda en el estado) y un texto visible (el que ve el usuario); pueden ser distintos. El botón llama a `confirmar`, que copia el valor actual de `curso` en `confirmado`. Y la última parte es renderizado condicional: `{confirmado !== "" && (...)}` significa "si `confirmado` no está vacío, muestra este párrafo; si está vacío, no muestres nada". El operador `&&` aquí funciona como un interruptor de visibilidad.

El flujo completo: el usuario cambia el select a "python" (ciclo: `onChange` -> `setCurso` -> render; el select muestra Python). Luego hace clic en confirmar (ciclo: `onClick` -> `setConfirmado("python")` -> render; aparece el párrafo "Te inscribiste al curso de python..."). Si después cambia el select a "html" sin confirmar, el párrafo sigue diciendo python, porque `confirmado` no cambió. Dos estados, dos responsabilidades, cero confusión. Este patrón (un estado para el borrador y otro para lo confirmado) aparece en formularios reales todo el tiempo: lo que el usuario está editando vs. lo que ya se guardó.

Ejemplo 5: Validador de correo en vivo

Este ejemplo combina input controlado, valores derivados y validación visual en tiempo real: el borde del campo cambia de color según lo que escribes, aparecen mensajes de ayuda, y el botón solo se activa con un correo válido. Es el esqueleto de cualquier formulario de registro real.

```
function App() {
  const [correo, setCorreo] = useState("");

  const esValido = correo.includes("@") && correo.includes(".");

  function colorBorde() {
    if (correo === "") return "1px solid gray";
    return esValido ? "2px solid green" : "2px solid red";
  }

  return (
    <div>
      <input
        value={correo}
        onChange={(e) => setCorreo(e.target.value)}
        placeholder="tu@correo.com"
        style={{ border: colorBorde() }}
      />
      {correo !== "" && !esValido && (
        <p style={{ color: "red" }}>Ese correo no parece valido.</p>
      )}
      {esValido && (
        <p style={{ color: "green" }}>Correo valido. Ya puedes continuar.</p>
      )}
      <button disabled={!esValido}>Registrarse</button>
    </div>
  );
}
```

Línea por línea: el estado correo guarda lo escrito. La constante esValido se calcula en cada renderizado: es true solo si el texto contiene arroba y punto. Ojo, esto es una validación didáctica simplificada; la validación real de correos es más estricta, pero para aprender el patrón basta. La función colorBorde devuelve el estilo del borde según el caso: gris neutro si el campo está vacío (para no reganar al usuario antes de empezar), verde si es válido, rojo si no. El input es controlado y su style usa esa función. Debajo hay dos mensajes condicionales mutuamente excluyentes: el rojo aparece solo si hay texto Y no es válido; el verde solo si es

válido. Y el botón usa `disabled={!esValido}`: mientras el correo no sea válido, no se puede hacer clic.

Fíjate en la economía de estados: solo hay UNO (correo). Todo lo demás (`esValido`, el color del borde, los mensajes, el botón habilitado) se deriva de el durante el renderizado. Si hubieras creado estados separados para "es válido" y "color del borde", tendrías que mantenerlos sincronizados a mano con cada tecla, y tarde o temprano se desincronizarían. Un estado, muchas vistas: esa es la forma React de pensar.

Ejemplo 6: Mini lista de tareas (todo junto)

El ejemplo capstone de la unidad: una lista de tareas donde escribes en un campo controlado, agregas con Enter o con botón, y eliminas cada tarea con su botón X. Junta eventos de teclado, inputs controlados, argumentos en manejadores y actualización inmutable de arreglos.

```

function App() {
  const [texto, setTexto] = useState("");
  const [tarefas, setTareas] = useState([]);

  function agregar() {
    if (texto === "") return;
    setTareas([...tarefas, texto]);
    setTexto("");
  }

  function agregarConEnter(e) {
    if (e.key === "Enter") {
      agregar();
    }
  }

  function eliminar(indice) {
    setTareas(tareas.filter((t, i) => i !== indice));
  }

  return (
    <div>
      <h3>Mis tareas</h3>
      <input
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        onKeyDown={agregarConEnter}
        placeholder="Nueva tarea"
      />
      <button onClick={agregar}>Agregar</button>
      <ul>
        {tarefas.map((tarea, i) => (
          <li key={i}>
            {tarea}
            <button onClick={() => eliminar(i)}>X</button>
          </li>
        ))}
      </ul>
      <p>Total: {tarefas.length} tareas.</p>
    </div>
  );
}

```

Línea por línea: dos estados, cada uno con su responsabilidad: texto (el borrador del campo) y tareas (la lista confirmada). La función agregar valida que el texto no este vacío (si lo esta, return sale sin hacer nada), crea un arreglo nuevo con la tarea agregada y limpia el campo. agregarConEnter escucha la tecla Enter y reutiliza agregar: un manejador llamando a otro, sin duplicar lógica. eliminar recibe el índice y usa filter para crear un arreglo nuevo sin ese elemento. En el JSX, el input combina onChange (controlado) con onKeyDown (Enter). El botón Agregar llama a agregar sin paréntesis. La lista se dibuja con map: cada tarea es un li con su botón X, que usa la flecha con argumento onClick={() => eliminar(i)} para saber cual borrar. El key={i} le dice a React como identificar cada elemento de la lista (en listas reales se usa un id único; el índice basta para este ejemplo). Y el parrafo final muestra tareas.length, otro valor derivado sin estado extra.

Prueba mental completa: escribes "estudiar React", pulsas Enter. onChange fue actualizando texto con cada letra; onKeyDown detecta Enter y llama a agregar; agregar crea ["estudiar React"] y limpia el campo; el renderizado muestra la tarea en la lista y "Total: 1 tareas.". Haces clic en X: eliminar(0) filtra el arreglo a [] y la tarea desaparece. Cada interacción, una vuelta del ciclo. Si puedes seguir este ejemplo sin perderte, estas listo para la sección de práctica.

Fíjate en lo que tienen en comun los seis ejemplos, porque ahí esta el patrón universal de esta unidad: en todos hay un estado que guarda la verdad, un evento que lo modifica a traves de un manejador, y una interfaz que lo refleja sin que toques el DOM. Contador, colores, espejo de texto, selector, validador y lista de tareas parecen aplicaciones distintas, pero por dentro son la misma máquina con distinto disfraz. Cuando empieces tus propios proyectos y no sepas por donde empezar, vuelve a esta idea: define primero que verdad necesita guardar tu componente (el estado), después que puede hacer el usuario (los eventos), y al final cómo se ve cada estado posible (el JSX). En ese orden, siempre. El resto lo hace React.

Práctica

Leer sobre interactividad es cómo leer sobre natación: hasta que no te metes al agua, no aprendes. Aquí van cinco ejercicios ordenados de fácil a retador. Intenta cada uno por tu cuenta antes de mirar la solución; si te atorras más de quince minutos, mira la pista, y solo al final la solución completa. El que programa de verdad es el que se equivoca con sus propias manos.

Ejercicio 1: Contador con limites (0 a 10)

Enunciado: crea un contador que empiece en 0, con botones para sumar y restar, pero que no pueda bajar de 0 ni subir de 10. Cuando llegue a un limite, el botón correspondiente debe deshabilitarse (quedarse gris y sin responder).

Pista: la prop disabled de un botón recibe un booleano. Piensa que condición hace que "sumar" deba deshabilitarse, y cuál hace que "restar" deba deshabilitarse.

Solución:

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

  function sumar() {
    if (num < 10) {
      setNum(num + 1);
    }
  }

  function restar() {
    if (num > 0) {
      setNum(num - 1);
    }
  }

  return (
    <div>
      <h2>Contador: {num} / 10</h2>
      <button onClick={sumar} disabled={num >= 10}>
        + Sumar
      </button>
      <button onClick={restar} disabled={num <= 0}>
        - Restar
      </button>
    </div>
  );
}
```

Explicación: hay dos capas de protección, y eso es a proposito. La primera capa está en los manejadores: sumar solo actualiza si `num < 10`, y restar solo si `num > 0`. La segunda capa está en el JSX: `disabled={num >= 10}` desactiva el botón de sumar cuando se llega al tope, y `disabled={num <= 0}` desactiva el de restar cuando se llega a cero. Por qué las dos? Porque un usuario habilidoso podría activar el botón con el teclado o con las herramientas del

navegador; la validación en el manejador es la que realmente cuida los datos, mientras que disabled es la que le comunica el límite al usuario de forma visual. Moraleja profesional: la interfaz sugiere, pero la lógica protege. Además, fíjate en que el límite (10) aparece en tres lugares: en el texto, en la condición del manejador y en el disabled. En una aplicación real lo guardarías en una constante (const MÁXIMO = 10) para no repetirlo; te lo dejo como mejora opcional.

Ejercicio 2: Botón "Me gusta" con contador

Enunciado: crea un botón que diga "Me gusta" y muestre cuantas personas le han dado clic. Cada clic suma uno. Si el contador llega a 100, el botón debe mostrar el mensaje "Eres muy popular" en lugar del número.

Pista: el texto del botón puede ser una expresión condicional que dependa del estado, igual que cualquier otra parte del JSX.

Solución:

```
function App() {
  const [likes, setLikes] = useState(0);

  function darLike() {
    setLikes(likes + 1);
  }

  return (
    <div>
      <button onClick={darLike}>
        Me gusta
      </button>
      {likes >= 100 ? (
        <p>Eres muy popular.</p>
      ) : (
        <p>A {likes} personas les gusta esto.</p>
      )}
    </div>
  );
}
```

Explicación: el estado likes cuenta los clics. El manejador darLike es de una sola línea: cada clic suma uno, el ciclo hace el resto. Lo interesante está en el renderizado condicional con el operador ternario: likes >= 100 ? (muestra el mensaje especial) : (muestra el conteo normal).

El ternario es el "si / si no" dentro del JSX: condición ? valorSiTrue : valorSiFalse. Fíjate en que el botón mismo no cambia; lo que cambia es el párrafo de abajo según el estado. Este patrón (mostrar una cosa u otra según un umbral del estado) es exactamente cómo funcionan las insignias, los niveles y los mensajes de logro en aplicaciones reales. Mejora opcional: deshabilita el botón al llegar a 100 para que el contador no siga subiendo, o cambia su texto a "Ya eres popular".

Ejercicio 3: Convertidor de texto a mayúsculas en vivo

Enunciado: crea un campo de texto controlado. Mientras el usuario escribe, muestra debajo el mismo texto pero TODO EN MAYUSCULAS, y además muestra cuantas letras lleva. El campo debe conservar lo que el usuario escribió originalmente (con minúsculas y mayúsculas mezcladas), solo la vista de abajo va en mayúsculas.

Pista: no transformes el estado; transforma solo lo que muestras. El estado guarda la verdad original, y el renderizado la presenta como quieras.

Solución:

```
function App() {
  const [texto, setTexto] = useState("");

  return (
    <div>
      <input
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        placeholder="Escribe en minúsculas"
      />
      <p>En mayúsculas: {texto.toUpperCase()}</p>
      <p>Letras: {texto.length}</p>
    </div>
  );
}
```

Explicación: este ejercicio prueba si entendiste la diferencia entre el dato y su presentación. El estado texto guarda EXACTAMENTE lo que el usuario escribió, sin tocarlo: esa es la fuente de verdad. La transformación a mayúsculas ocurre solo en el momento de mostrar: {texto.toUpperCase()}. Como toUpperCase() no modifica el texto original (crea uno nuevo), el estado sigue intacto y el input sigue mostrando lo que el usuario tecleo. Si hubieras hecho setTexto(e.target.value.toUpperCase()) en el onChange, el campo mismo forzaría mayúsculas mientras escribes, que es un comportamiento distinto (y a veces deseado, pero no era lo

pedido). Moraleja: guarda la verdad cruda en el estado y transforma en el renderizado; así puedes mostrar el mismo dato de muchas formas (mayúsculas, conteo, vista previa) sin duplicar estados. El conteo con `texto.length` es otro valor derivado: se calcula solo, sin estado extra.

Ejercicio 4: Formulario de saludo personalizado

Enunciado: crea un formulario con dos campos controlados (nombre y ciudad) y un botón de enviar. Al enviar, SIN recargar la página, debe aparecer el mensaje: "Hola [nombre], saludos desde [ciudad]." Si algún campo está vacío, el botón debe estar deshabilitado.

Pista: necesitas `preventDefault` en el `onSubmit`, un estado objeto (o dos estados) para los campos, y un tercer estado para el mensaje final. El botón se deshabilita cuando nombre o ciudad estén vacíos.

Solución:

```

function App() {
  const [nombre, setNombre] = useState("");
  const [ciudad, setCiudad] = useState("");
  const [saludo, setSaludo] = useState("");

  function manejarEnvio(e) {
    e.preventDefault();
    setSaludo("Hola " + nombre + ", saludos desde " + ciudad + ".");
  }

  const formularioIncompleto = nombre === "" || ciudad === "";

  return (
    <div>
      <form onSubmit={manejarEnvio}>
        <input
          value={nombre}
          onChange={(e) => setNombre(e.target.value)}
          placeholder="Tu nombre"
        />
        <input
          value={ciudad}
          onChange={(e) => setCiudad(e.target.value)}
          placeholder="Tu ciudad"
        />
        <button type="submit" disabled={formularioIncompleto}>
          Enviar saludo
        </button>
      </form>
      {saludo !== "" && <p>{saludo}</p>}
    </div>
  );
}

```

Explicación: este ejercicio junta casi todo lo de la unidad en un solo componente, así que desmenucemoslo. Hay tres estados: nombre y ciudad (los borradores que el usuario edita) y saludo (el mensaje ya confirmado). Esta separación borrador vs. confirmado es el patrón profesional que viste en el ejemplo 4: mientras escribes, el párrafo no cambia; solo cambia cuando envías. El manejador `manejarEnvio` hace dos cosas: primero `e.preventDefault()` para que el formulario no recargue la página (sin esto, todo el ejercicio se rompe), y luego construye el mensaje con los valores actuales y lo guarda en `saludo`. La constante `formularioIncompleto` se calcula en cada renderizado: es `true` si algún campo está vacío. Al pasarla a `disabled`, el botón se prende y se apaga solo según lo que el usuario escriba, sin que tengas que tocar el

botón a mano. Por último, el renderizado condicional {saludo !== "" && <p>{saludo}</p>} muestra el mensaje solo después del primer envío. Nota que el botón es type="submit" dentro de un form con onSubmit: así funciona tanto con clic como con la tecla Enter, que es el comportamiento que los usuarios esperan de un formulario.

Ejercicio 5: Interruptor de tema claro / oscuro

Enunciado: crea una página con un interruptor que cambie entre tema claro y tema oscuro. El tema debe afectar al contenedor principal: fondo blanco con texto negro en modo claro, fondo negro con texto blanco en modo oscuro. El botón debe decir "Cambiar a modo oscuro" o "Cambiar a modo claro" según corresponda.

Pista: el estado puede ser un booleano (modoOscuro: true/false). El estilo del contenedor se calcula a partir de ese booleano con un ternario, igual que el texto del botón.

Solución:

```
function App() {
  const [modoOscuro, setModoOscuro] = useState(false);

  function alternarTema() {
    setModoOscuro(!modoOscuro);
  }

  const estilos = {
    backgroundColor: modoOscuro ? "black" : "white",
    color: modoOscuro ? "white" : "black",
    padding: "40px",
    minHeight: "200px"
  };

  return (
    <div style={estilos}>
      <h2>Mi pagina con temas</h2>
      <p>El tema actual es {modoOscuro ? "oscuro" : "claro"}.</p>
      <button onClick={alternarTema}>
        {modoOscuro ? "Cambiar a modo claro" : "Cambiar a modo oscuro"}
      </button>
    </div>
  );
}
```

Explicación: el corazón es el booleano `modoOscuro` y la línea `setModoOscuro(!modoOscuro)`: el signo `!` invierte el valor actual, así que cada clic alterna entre `true` y `false`. Es el interruptor de la luz de la analogía, en una sola línea. Todo lo demás se DERIVA de ese booleano: el objeto `estilos` usa ternarios para elegir colores, el párrafo muestra "oscuro" o "claro", y el texto del botón ofrece siempre la acción contraria a la actual (si estás en oscuro, te ofrece cambiar a claro). Fíjate en la elegancia: un solo estado booleano controla el fondo, el color del texto, dos textos distintos y el comportamiento del botón. Cambias UNA cosa y CUATRO partes de la interfaz responden. Eso es programación declarativa en su máxima expresión, y es así cómo funcionan los modos oscuros de las aplicaciones reales (solo que ellas además guardan tu preferencia para la próxima visita, tema de otra unidad). Si este ejercicio te salió sin mirar la solución, el ciclo evento -> estado -> render ya vive en tu cabeza.

Como seguir practicando por tu cuenta

Los cinco ejercicios cubren lo esencial, pero la interactividad se domina jugando. Aquí van ideas para seguir, ordenadas de más fácil a más retadora. Primera: combina dos ejercicios en uno solo, por ejemplo el formulario de saludo (ejercicio 4) dentro de una página con interruptor de tema (ejercicio 5); practicarás como conviven varios estados en un mismo componente sin estorbarse. Segunda: rompe el código a propósito. Quita los paréntesis donde van y ponlos donde no van, elimina el `preventDefault`, deja un `value` sin `onChange`, y observa con calma que síntoma produce cada sabotaje; reconocer esos síntomas en tus propios proyectos te ahorrará horas de depuración. Tercera: agrega persistencia mental al ejercicio 2, haciendo que el botón "Me gusta" alterne entre "Me gusta" y "Ya no me gusta" restando el like si se vuelve a pulsar (pista: necesitas un booleano más). Cuarta: construye un cronómetro con botones de iniciar, pausar y reiniciar; te obligará a pensar que eventos disparan que cambios de estado cuando hay tiempo de por medio. Y quinta, la más divertida: toma cualquier página estática que hayas hecho en las unidades anteriores y conviértela en interactiva, aunque sea con un solo botón que muestre u oculte una sección. Ver tus viejos componentes cobrar vida es la mejor prueba de que esta unidad quedó dominada.

Resumen de la unidad

Llegaste al final de la unidad donde tus componentes aprendieron a escuchar y a responder. Hagamos un repaso de todo lo que ahora vive en tu caja de herramientas, porque es mucho y todo está conectado.

Primero, el sistema de eventos de React: los eventos se escriben en camelCase (`onClick`, `onChange`, `onSubmit`) porque en JSX no son atributos HTML sino propiedades de objetos de

JavaScript. React envuelve los eventos del navegador en eventos sintéticos que se comportan igual en todos los navegadores, y los reparte con delegación desde la raíz para que tu aplicación sea rápida sin que tengas que hacer nada.

Segundo, los manejadores de eventos: son funciones que se PASAN al evento, no se llaman. `onClick={saludar}` es correcto; `onClick={saludar()}` es el error más famoso del principiante, porque ejecuta la función durante el dibujado y deja el botón muerto. Cuando necesites argumentos, envuelve la llamada en una flecha: `onClick={() => saludar("Ana")}`. El manejador recibe el objeto del evento (e), cuya propiedad más útil es `e.target` (el elemento donde ocurrió todo): `e.target.value` para leer inputs, `e.target.checked` para checkbox. Y en los formularios, `e.preventDefault()` es obligatorio para que el navegador no recargue la página y borre tu estado.

Tercero, los inputs controlados: el patrón `value + onChange` convierte a React en la única fuente de verdad. El input muestra exactamente lo que dice el estado y avisa de cada tecla para que el estado se actualice. `value` sin `onChange` congela el campo; mutar `e.target.value` a espaldas de React rompe la sincronía. El mismo trato aplica a `textarea` (con `value`, no con `text` entre etiquetas), a `select` (`value` en el `select`, no `selected` en las opciones) y a `checkbox` (`checked + e.target.checked`). Para formularios con varios campos, un solo objeto en el estado con un manejador genérico que usa `e.target.name` es el patrón profesional.

Cuarto, el ciclo completo de interactividad, la máquina que lo mueve todo: el usuario provoca un evento, el manejador llama a la función actualizadora del estado, React vuelve a dibujar el componente con el nuevo estado, compara con el dibujo anterior y actualiza solo lo que cambió en el DOM. Tu trabajo es declarativo: describes cómo se ve la interfaz para cada estado y qué evento produce cada cambio; React se encarga del resto. Cuando algo se ve mal, preguntas por el estado, no por el DOM.

Quinto, los errores que ya no te asustan: llamar en vez de pasar la función, olvidar `preventDefault`, congelar inputs con `value` sin `onChange`, mutar el DOM directamente, leer el estado justo después de `setState` esperando el valor nuevo, y olvidar el atributo `name` en manejadores genéricos. Cada uno tiene síntoma claro y cura rápida, y los documentaste aquí para reconocerlos en segundos.

Y sexto, las buenas prácticas que te hacen ver profesional desde el día uno: manejadores con nombres `handle...`, lógica fuera del JSX cuando crece, estado mínimo sin duplicados, valores derivados calculados en el renderizado, y estados inicializados con el tipo correcto.

Con esta unidad completas el circuito fundamental de React: estado (unidad 3) + eventos (unidad 4) = aplicaciones interactivas de verdad. Lo que sigue en el curso construye sobre este ciclo: listas dinámicas que reaccionan a tus clics, efectos que ocurren cuando el estado

cambia, y aplicaciones completas donde cada pantalla es una conversación con el usuario. Práctica los cinco ejercicios hasta que te salgan sin mirar, juega a romper el código a proposito para ver qué pasa, y cuando estes listo, pasa la página: tus componentes ya saben escuchar, y lo que viene es enseñarles a recordar y a actuar por su cuenta.

UNIDAD 5: Formularios y efectos secundarios

Hasta esta unidad todo lo que hiciste en React fue, por decirlo de alguna manera, de dientes para adentro: el usuario miraba tu interfaz, hacía clic en botones, veía contadores que subían y bajaban, y todo quedaba bonito dentro de la pantalla. Pero una aplicación real no vive aislada. Las aplicaciones de verdad tienen que hacer dos cosas que todavía no hemos cubierto: escuchar lo que el usuario escribe y comunicarse con el mundo exterior. La primera se logra con formularios, y la segunda con efectos secundarios. Esta unidad te enseña ambas cosas, y además te muestra cómo combinarlas, porque en la práctica casi siempre van juntas: un formulario pide datos, un efecto los envía o los trae de un servidor, y la pantalla se actualiza con el resultado.

Piensa en cualquier página que uses todos los días. El buscador de Google es un formulario. El inicio de sesión de tu banco es un formulario. El cuadro de comentarios de YouTube es un formulario. Y del otro lado, cuando abres una app del clima y ves la temperatura de tu ciudad, eso vino de un efecto secundario: tu app le preguntó a una API en internet "oye, cómo está el clima", esperó la respuesta y la dibujó en pantalla. Los formularios son el puente de entrada: el usuario hacia la aplicación. Los efectos son el puente de salida y de regreso: la aplicación hacia el mundo y el mundo hacia la aplicación.

La palabra "efecto secundario" suena a término de película de ciencia ficción, pero la idea es simple. En programación, un efecto secundario es cualquier cosa que tu código hace además de calcular un valor y devolverlo: cambiar el título de la pestaña del navegador, pedir datos a un servidor, arrancar un temporizador, guardar algo en el almacenamiento local, suscribirse a un evento del teclado. El componente de React, en teoría, debería ser una función pura: recibe props, devuelve JSX, y nada más. Pero las apps reales necesitan ensuciarse las manos, y React te da una herramienta oficial para hacerlo de forma ordenada: el hook `useEffect`. Él se encarga de decir "este código no es parte del dibujo, es un efecto; corre después de que la pantalla ya se pintó".

En esta unidad vas a aprender primero los formularios controlados: cómo conectar cada input a una variable de estado, cómo manejar el envío con `onSubmit`, y por qué tienes que llamar a `e.preventDefault()` si no quieres que la página se recargue como en la web de 1998. Vas a ver el truco de manejar diez inputs con un solo objeto de estado usando `e.target.name`, cómo limpiar el formulario después de enviarlo, y cómo hacer validaciones básicas sin librerías externas.

Después entraremos a `useEffect` a fondo: su sintaxis, el momento exacto en que se ejecuta (después del render, siempre), y su pieza más importante y más malentendida: el arreglo de

dependencias. Verás los tres casos posibles: sin arreglo (el efecto corre en cada render), con arreglo vacío `[]` (corre una sola vez al montar el componente) y con dependencias `[valor]` (corre cuando ese valor cambia). También aprenderás la función de limpieza, el `cleanup`, que sirve para apagar temporizadores y cancelar suscripciones antes de que el componente desaparezca o el efecto se repita. Y hablaremos del modo estricto de React, `StrictMode`, que en desarrollo ejecuta los efectos dos veces a propósito para ayudarte a detectar errores.

La tercera parte une todo con `fetch`: cómo pedir datos a una API REST real usando `fetch` dentro de `useEffect`, cómo manejar promesas con `.then` o con `async/await`, y el patrón de tres estados que vas a usar en todos tus proyectos de ahora en adelante: `loading` (cargando), `error` y `data` (datos). Lo practicaremos con una API pública gratuita, `jsonplaceholder`, y también con la API de Pokémon, que es perfecta para ejercicios.

Al final de la unidad tendrás cuatro ejemplos completos explicados línea por línea y cinco ejercicios con solución, para que no solo entiendas la teoría sino que la tengas en los dedos. Cuando termines, sabrás construir el ciclo completo de una app moderna: el usuario escribe, tú validas, pides datos, muestras un indicador de carga, presentas el resultado o un mensaje de error amable, y limpias todo al desmontar. Eso, créeme, es el 80 por ciento de lo que hacen las aplicaciones web profesionales todos los días.

Explicación detallada

Formularios en React: el puente con el usuario

Empecemos por lo básico: qué es un formulario en HTML puro. Un formulario es la etiqueta `<form>` con `inputs` adentro y un botón de tipo `submit`. Cuando el usuario presiona `Enter` dentro de un `input` o hace clic en el botón de enviar, el navegador dispara el evento `submit`. Y aquí viene un detalle histórico importantísimo: por defecto, el navegador toma todos los datos del formulario y hace una petición al servidor recargando la página completa. Así funcionaba la web en los años noventa: llenabas un formulario, se enviaba, la página se ponía en blanco, y el servidor te devolvía una página nueva con el resultado. Era lento y feo, pero era lo único que había.

React es una librería para construir interfaces que NO recargan la página. Las famosas SPA (Single Page Applications) viven de eso: todo pasa en la misma página, sin recargas. Entonces, cuando usas un formulario en React, lo primero que tienes que hacer es decirle al navegador: "gracias, pero yo me encargo; no recargues nada". Eso se hace con `e.preventDefault()`. La "e" es el objeto del evento, y `preventDefault` significa literalmente "prevén el comportamiento por defecto". Sin esa línea, tu app de React se comportaría como

una página de 1998: parpadeo, recarga, y se pierde todo el estado. Es el error número uno de los principiantes, así que grábatelo: en React, todo manejador de onSubmit empieza con e.preventDefault().

Mira el patrón básico:

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

function FormularioNombre() {
  const [nombre, setNombre] = useState('');

  function enviar(e) {
    e.preventDefault();
    alert('Hola, ' + nombre);
  }

  return (
    <form onSubmit={enviar}>
      <input
        type="text"
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
        placeholder="Escribe tu nombre"
      />
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Fíjate en tres cosas. Primera: el formulario usa onSubmit={enviar}, no onClick en el botón. ¿Por qué? Porque onSubmit captura también la tecla Enter cuando el usuario está dentro de un input, que es el comportamiento que los usuarios esperan. Si pones el manejador en el onClick del botón, el Enter no funcionará y tu formulario se sentirá roto. Segunda: el input es un "input controlado": su atributo value siempre es igual a la variable de estado nombre, y cada tecla que el usuario presiona dispara onChange, que actualiza el estado con e.target.value. React es la única fuente de la verdad: el input nunca tiene un valor por su cuenta, siempre refleja el estado. Tercera: el botón tiene type="submit", que es lo que hace que el clic dispare el evento submit del formulario.

¿Por qué inputs controlados y no dejar que el input guarde su propio valor? Porque si el valor vive en el estado de React, puedes hacer cosas poderosas: validar en tiempo real (mostrar "te faltan 3 caracteres" mientras escribes), transformar el texto (convertir todo a mayúsculas),

deshabilitar el botón de enviar hasta que todo sea válido, y limpiar el formulario con un simple `setNombre('')` después de enviar. Con inputs no controlados tendrías que ir a leer el DOM a mano, y eso es pelearse con React en lugar de trabajar con él.

Múltiples inputs con un solo objeto de estado

Un formulario real rara vez tiene un solo campo. Un registro típico pide nombre, correo y contraseña. Podrías crear tres `useState` separados, y para tres campos está bien, pero cuando el formulario crece a ocho o diez campos se vuelve un desorden. El truco profesional es guardar todos los campos en un solo objeto de estado y usar el atributo `name` de cada input para saber cuál actualizar:

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

function Registro() {
  const [datos, setDatos] = useState({
    nombre: '',
    correo: '',
    password: ''
  });

  function manejarCambio(e) {
    setDatos({
      ...datos,
      [e.target.name]: e.target.value
    });
  }

  function enviar(e) {
    e.preventDefault();
    console.log('Datos enviados:', datos);
    setDatos({ nombre: '', correo: '', password: '' });
  }

  return (
    <form onSubmit={enviar}>
      <input
        type="text"
        name="nombre"
        value={datos.nombre}
        onChange={manejarCambio}
        placeholder="Nombre"
      />
      <input
```

```

        type="email"
        name="correo"
        value={datos.correo}
        onChange={manejarCambio}
        placeholder="Correo"
      />
      <input
        type="password"
        name="password"
        value={datos.password}
        onChange={manejarCambio}
        placeholder="Contraseña"
      />
      <button type="submit">Registrarse</button>
    </form>
  );
}

```

La línea mágica es `[e.target.name]: e.target.value`. Los corchetes en un objeto literal de JavaScript significan "usa el valor de esta expresión como nombre de la propiedad". Si el usuario escribe en el input que tiene `name="correo"`, entonces `e.target.name` vale "correo" y la expresión actualiza solo la propiedad correo del objeto, dejando las demás intactas. Y el `...datos` (spread) es crucial: crea una copia del objeto anterior con todas sus propiedades, y luego sobrescribe solo la que cambió. Si olvidaras el spread, cada tecla borraría los demás campos, porque estarías reemplazando todo el objeto con un objeto de una sola propiedad. Es un error clásico: el usuario llena el nombre, pasa al correo, escribe una letra y el nombre desaparece. Spread siempre.

Observa también la última línea de enviar: después de procesar los datos, limpiamos el formulario poniendo todos los campos en cadenas vacías. Como los inputs son controlados, en cuanto el estado se vacía la pantalla se vacía sola. No hay que tocar el DOM, no hay que hacer `form.reset()`: React se encarga.

Validación básica de formularios

Antes de enviar datos a un servidor, conviene verificar que tengan sentido: que el correo parezca correo, que la contraseña tenga mínimo de caracteres, que ningún campo obligatorio esté vacío. Puedes hacerlo a mano con JavaScript normal dentro de la función enviar, y guardar los mensajes de error en otro estado:

```
function enviar(e) {
  e.preventDefault();
  const errores = {};
  if (datos.nombre.trim() === '') {
    errores.nombre = 'El nombre es obligatorio';
  }
  if (!datos.correo.includes('@')) {
    errores.correo = 'Escribe un correo válido';
  }
  if (datos.password.length < 6) {
    errores.password = 'Mínimo 6 caracteres';
  }
  setErrores(errores);
  if (Object.keys(errores).length === 0) {
    console.log('Todo válido, enviando:', datos);
  }
}
```

El truco de `datos.nombre.trim() === ''` detecta tanto el campo vacío como el que solo tiene espacios: `trim()` quita los espacios de los extremos. Y `Object.keys(errores).length === 0` te dice si el objeto de errores quedó vacío, es decir, si todo pasó la validación. En el JSX muestras cada error debajo de su input solo si existe:

```
{errores.nombre && <p className="error">{errores.nombre}</p>}
```

Este patrón (`&&` para render condicional) ya lo conoces de unidades anteriores. Para proyectos grandes existen librerías como Formik o React Hook Form que hacen todo esto por ti, pero es importante que primero entiendas cómo funciona a mano: las librerías son atajos, no magia.

Otro detalle útil: el atributo `required` de HTML5 (`<input required />`) le pide al navegador que bloquee el envío si el campo está vacío, y `type="email"` valida el formato básico del correo sin que escribas código. Puedes usarlos como primera línea de defensa, pero no confíes solo en ellos: la validación de verdad siempre se hace también en tu código, porque el usuario puede tener un navegador viejo o manipular el HTML.

Más controles: checkbox, radio, select y textarea

Hasta aquí usamos inputs de texto, pero los formularios reales tienen más controles y cada uno se conecta al estado de forma ligeramente distinta. Vamos a verlos porque te los vas a encontrar en todos lados.

El checkbox no usa value sino checked, porque su estado es un booleano (marcado o no):

```
const [acepto, setAcepto] = useState(false);

<input
  type="checkbox"
  checked={acepto}
  onChange={(e) => setAcepto(e.target.checked)}
/>
<label>Acepto los términos y condiciones</label>
```

Fíjate que leemos e.target.checked en lugar de e.target.value: para un checkbox, lo que importa es si está marcado. Este patrón sirve para "acepto términos", "recibir notificaciones" o listas de intereses donde el usuario marca varias opciones.

Los radio buttons trabajan en grupo: varios inputs comparten el mismo name y cada uno tiene un value distinto. El estado guarda el value del seleccionado:

```
const [genero, setGenero] = useState('');

<input
  type="radio"
  name="genero"
  value="masculino"
  checked={genero === 'masculino'}
  onChange={(e) => setGenero(e.target.value)}
/> Masculino
<input
  type="radio"
  name="genero"
  value="femenino"
  checked={genero === 'femenino'}
  onChange={(e) => setGenero(e.target.value)}
/> Femenino
```

El truco está en checked={genero === 'masculino'}: cada radio se marca solo si el estado coincide con su value. Como comparten el name, el navegador ya sabe que son excluyentes, pero es React quien decide cuál se ve marcado a través del estado.

El select (lista desplegable) se controla con value en la etiqueta select, no en las opciones:

```
const [ciudad, setCiudad] = useState('');

<select value={ciudad} onChange={(e) => setCiudad(e.target.value)}>
  <option value="">Elige tu ciudad</option>
  <option value="cdmx">Ciudad de México</option>
  <option value="gdl">Guadalajara</option>
  <option value="mty">Monterrey</option>
</select>
```

La primera opción con `value=""` funciona como placeholder: si el usuario no elige nada, el estado queda en cadena vacía y tu validación lo detecta igual que un campo de texto vacío.

El textarea, que ya viste en los ejercicios, se controla igual que un input de texto: `value` y `onChange`. La única diferencia con el HTML clásico es que en React no lleva el texto entre la etiqueta de apertura y cierre (`<textarea>texto</textarea>`), sino que usa el atributo `value` como todos los demás.

Un consejo práctico: cuando un formulario mezcla varios tipos de control, el truco del objeto único de estado sigue funcionando, pero el manejador genérico necesita un ajuste para los checkboxes:

```
function manejarCambio(e) {
  const { name, type, value, checked } = e.target;
  setDatos({
    ...datos,
    [name]: type === 'checkbox' ? checked : value
  });
}
```

Con ese ternario, el mismo manejador sirve para inputs de texto, selects, textareas y checkboxes. Es el manejador universal de formularios en React, y lo verás en muchísimos proyectos profesionales.

useEffect: qué es un efecto secundario

Ahora cambiemos de tema. Hasta aquí tus componentes han sido funciones más o menos puras: reciben props y estado, devuelven JSX. Pero las apps reales necesitan hacer cosas que no son "dibujar": pedir datos a un servidor, cambiar el título de la pestaña del navegador, arrancar un reloj, guardar en `localStorage`, suscribirse a eventos del teclado. Todo eso son efectos secundarios: operaciones que ocurren "al lado" del renderizado, que interactúan con el mundo fuera de React.

React te da el hook `useEffect` para manejarlas. Su idea central es una separación limpia: el cuerpo del componente se dedica a calcular qué se debe ver en pantalla, y `useEffect` se dedica a todo lo demás. La sintaxis es:

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

useEffect(() => {
  // código del efecto
}, [dependencias]);
```

Recibe dos argumentos: una función con el código del efecto y un arreglo de dependencias (opcional). Y la regla de oro del momento de ejecución: el efecto corre DESPUÉS de que React pinta la pantalla. Primero el componente se renderiza y el usuario ve algo; después, React ejecuta tu efecto. Esto es a propósito: así el efecto nunca bloquea ni retrasa lo que el usuario ve. Si el efecto tarda (por ejemplo, una petición a internet), la pantalla ya está visible con un mensaje de "cargando".

Mira el ejemplo más simple posible, el de la clase 14:

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

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

  useEffect(() => {
    document.title = 'Clicks: ' + num;
  }, [num]);

  return (
    <div>
      <p>Llevas {num} clicks</p>
      <button onClick={() => setNum(num + 1)}>Sumar</button>
    </div>
  );
}
```

Cada vez que `num` cambia, el componente se vuelve a renderizar con el nuevo número, y después del render React corre el efecto, que actualiza el título de la pestaña del navegador. Cambiar `document.title` es el ejemplo perfecto de efecto secundario: no es parte del JSX, es una manipulación del mundo exterior (el DOM del navegador fuera de tu app). Sin `useEffect` podrías poner `document.title = ...` directamente en el cuerpo del componente, y funcionaría... la mayoría de las veces. Pero sería incorrecto: el cuerpo del componente puede ejecutarse

varias veces por render (React a veces lo invoca de más en desarrollo), y mezclar efectos con el cálculo del JSX produce errores difíciles de rastrear. `useEffect` existe para que digas explícitamente "esto es un efecto, trátalo como tal".

El arreglo de dependencias: los tres casos

El segundo argumento de `useEffect`, el arreglo de dependencias, es lo que más confunde a los principiantes y lo que más errores causa. Es una lista de valores (estado o props) que el efecto "observa". React lo usa para decidir cuándo volver a ejecutar el efecto. Hay exactamente tres casos, y tienes que dominarlos:

Caso 1: sin arreglo de dependencias. Si omites el segundo argumento, el efecto se ejecuta después de CADA render del componente. Cada vez que cualquier estado cambie y la pantalla se repinte, el efecto corre de nuevo.

```
useEffect(() => {  
  console.log('Me ejecuto en cada render');  
});
```

Esto rara vez es lo que quieres, porque si dentro del efecto actualizas un estado, provocas otro render, que dispara el efecto otra vez, que actualiza el estado... bucle infinito. Úsalo solo para efectos que de verdad deben correr siempre, como registrar algo en la consola para depurar.

Caso 2: arreglo vacío `[]`. El efecto se ejecuta UNA SOLA VEZ, justo después del primer render (cuando el componente "se monta"). Es el caso del fetch de datos al cargar: quieres pedir la lista de usuarios una vez cuando la página abre, no cada vez que el usuario escribe una letra.

```
useEffect(() => {  
  console.log('Solo me ejecuto al montar');  
}, []);
```

Caso 3: arreglo con dependencias `[valor]`. El efecto se ejecuta al montar Y cada vez que alguno de los valores del arreglo cambie. Es el caso del título del contador: `[num]` significa "corre al inicio y cada vez que num cambie".

```
useEffect(() => {  
  document.title = 'Clicks: ' + num;  
}, [num]);
```

Puedes poner varias dependencias: `[nombre, edad]` significa "corre cuando nombre O edad cambien". Y la regla de oro que te evitará el 90 por ciento de los dolores de cabeza: todo valor

del componente (estado o prop) que uses DENTRO del efecto debe aparecer en el arreglo de dependencias. Si usas num dentro pero pones [], el efecto verá siempre el valor viejo de num (un "closure" desactualizado) y te volverás loco preguntándote por qué no se actualiza. React incluso te avisa con una advertencia en la consola si instalas el plugin de ESLint para hooks. Hazle caso a esa advertencia: casi siempre tiene razón.

Analogía para que no se te olvide: el arreglo de dependencias es como la lista de ingredientes de una receta pegada en el refri. Si la lista dice [huevo, leche], solo vuelves al súper cuando se acaba el huevo o la leche. Si la lista está vacía [], fuiste una vez y jamás regresas. Si no hay lista, vas al súper cada vez que cocinas, aunque no necesites nada.

La función de limpieza (cleanup)

Algunos efectos dejan cosas "encendidas" que hay que apagar: un temporizador con setInterval, una suscripción a un evento, una conexión abierta. Si el componente desaparece de la pantalla (se desmonta) o el efecto se va a repetir por un cambio de dependencias, primero hay que limpiar lo anterior. Para eso, la función del efecto puede DEVOLVER otra función: la función de limpieza o cleanup. React la ejecuta automáticamente antes de repetir el efecto y cuando el componente se desmonta.

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

function Reloj() {
  const [hora, setHora] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    const id = setInterval(() => {
      setHora(new Date().toLocaleTimeString());
    }, 1000);

    return () => {
      clearInterval(id);
    };
  }, []);

  return <p>Hora actual: {hora}</p>;
}
```

Sigue la secuencia: al montar, el efecto arranca un intervalo que actualiza la hora cada segundo. La función que devuelve el efecto (el cleanup) apaga ese intervalo con clearInterval. ¿Cuándo corre el cleanup? En este caso, con dependencias [], solo cuando el componente se

desmonte. Pero si las dependencias fueran [algo], el cleanup correría también justo antes de cada repetición del efecto: primero apaga el intervalo viejo, luego arranca el nuevo. Sin cleanup, cada repetición crearía un intervalo nuevo sin matar el anterior, y tendrías 2, 3, 10 relojes corriendo a la vez, cada vez más rápido. Es el clásico error del "reloj acelerado".

La regla es simple: si tu efecto ENCIENDE algo (intervalo, suscripción, listener), el cleanup lo APAGA. Piensa en el efecto como alguien que renta un salón para una fiesta: el cleanup es dejar el salón limpio al salir. Si no lo haces, la próxima fiesta empieza con la basura de la anterior.

Otro ejemplo típico: suscribirse a eventos del teclado.

```
useEffect(() => {
  function manejarTecla(e) {
    console.log('Tecla presionada:', e.key);
  }
  window.addEventListener('keydown', manejarTecla);

  return () => {
    window.removeEventListener('keydown', manejarTecla);
  };
}, []);
```

Sin el `removeEventListener` del cleanup, cada vez que el componente se montara se agregaría otro listener, y una sola tecla dispararía el manejador cinco veces. Los "listeners fantasma" son una fuente clásica de comportamientos raros.

StrictMode y el doble efecto en desarrollo

Si creaste tu proyecto con las herramientas modernas de React, tu app probablemente está envuelta en `<StrictMode>`. En modo desarrollo, `StrictMode` monta cada componente, lo desmonta y lo vuelve a montar, lo que hace que los efectos con `[]` se ejecuten DOS veces seguidas. Esto asusta a muchos principiantes: "mi fetch se llama dos veces, ¿está roto?".

No está roto: es a propósito. React lo hace para ayudarte a encontrar efectos mal escritos. Si tu efecto es correcto (con su cleanup bien hecho), ejecutarlo dos veces no causa daño: el primero se limpia antes de que corra el segundo. Si en cambio tu efecto duplica suscripciones o deja basura, el doble montaje lo revela de inmediato en desarrollo, en lugar de que te explote en producción un mes después. En el build de producción (la versión que subes a internet) `StrictMode` no hace el doble efecto; solo ocurre en desarrollo.

Moraleja: si ves doble ejecución en desarrollo, no lo "arregles" con trucos raros como banderas globales. Escribe el cleanup correctamente y el doble efecto será inofensivo. Y si tu fetch se dispara dos veces en desarrollo pero una en producción, todo está bien.

Moraleja: si ves doble ejecución en desarrollo, no lo "arregles" con trucos raros como banderas globales. Escribe el cleanup correctamente y el doble efecto será inofensivo. Y si tu fetch se dispara dos veces en desarrollo pero una en producción, todo está bien.

useEffect en la práctica avanzada: múltiples efectos y objetos en dependencias

Dos temas que tarde o temprano te van a morder si no los conoces: cómo organizar varios efectos en un componente y por qué meter objetos o arrays en el arreglo de dependencias es peligroso.

Primero, los múltiples efectos. Un componente puede (y debe) tener tantos useEffect como responsabilidades externas tenga. Cada uno con su propio arreglo de dependencias:

```
function PerfilUsuario({ id }) {
  const [usuario, setUsuario] = useState(null);
  const [visitas, setVisitas] = useState(0);

  // Efecto 1: pide el usuario cuando cambia el id.
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users/' + id)
      .then((r) => r.json())
      .then(setUsuario);
  }, [id]);

  // Efecto 2: actualiza el título cuando cambia el usuario.
  useEffect(() => {
    if (usuario) {
      document.title = usuario.name;
    }
  }, [usuario]);

  // Efecto 3: cuenta una visita solo al montar.
  useEffect(() => {
    setVisitas((v) => v + 1);
  }, []);

  // ...JSX
}
```

Tres efectos, tres trabajos, tres arrays distintos. Si los mezclaras en uno solo, las dependencias se enredarían: el fetch se repetiría cuando no debe o el título no se actualizaría cuando debe. La regla es: un efecto por sincronización con el exterior. Es más código en líneas, pero infinitamente más fácil de razonar.

Segundo, el peligro de los objetos en dependencias. React compara las dependencias con `Object.is`, que para objetos y arrays significa "¿es el mismo objeto en memoria?". Mira este error sutil:

```
const filtros = { categoria: 'libros' }; // se crea NUEVO en cada render

useEffect(() => {
  buscarConFiltros(filtros);
}, [filtros]); // MAL: filtros es un objeto nuevo cada vez -> bucle infinito
```

Como `filtros` se crea de nuevo en cada render, React cree que "cambió" siempre, y el efecto se repite sin parar. Las soluciones: pon las propiedades primitivas en el arreglo (`[filtros.categoría]`), o crea el objeto dentro del efecto, o memoízalo con `useMemo` (lo verás en otra unidad). La versión simple y correcta:

```
useEffect(() => {
  buscarConFiltros({ categoria: 'libros' });
}, []); // el objeto vive dentro del efecto, no en dependencias
```

O si la categoría viene de una prop o estado:

```
useEffect(() => {
  buscarConFiltros({ categoria });
}, [categoria]); // primitivo: string, se compara por valor, sin problema
```

Los primitivos (string, number, boolean) se comparan por valor y son seguros en dependencias. Los objetos, arrays y funciones se comparan por referencia y casi siempre causan repeticiones inesperadas. Cuando tu efecto se repita "sin razón", lo primero que debes revisar es si metiste un objeto al arreglo.

Un último detalle avanzado: a veces necesitas el valor anterior de una dependencia dentro del efecto. Para eso existe el patrón de guardar el valor previo en una ref (`useRef`), que también verás más adelante. Por ahora quédate con la idea de que el efecto siempre ve los valores del render en que se ejecutó, ni más nuevos ni más viejos.

Fetch de datos: conectar tu app con el mundo

Llegamos a la parte más emocionante: pedir datos a internet. Casi ninguna app moderna trae sus datos quemados en el código; los pide a un servidor a través de una API. Una API REST es, en términos simples, una dirección URL que en lugar de devolverte una página web te devuelve datos puros, normalmente en formato JSON. Por ejemplo, la API pública jsonplaceholder (<https://jsonplaceholder.typicode.com>) es un servicio gratuito de práctica que simula un blog: si pides <https://jsonplaceholder.typicode.com/users> te devuelve una lista de 10 usuarios falsos en JSON. Es perfecta para aprender porque no necesita registro ni claves.

Para hacer la petición usamos fetch, una función que ya viene en el navegador. fetch devuelve una promesa: un objeto que representa "un valor que todavía no tengo, pero tendré en el futuro". Las promesas se encadenan con .then: "cuando llegue la respuesta, haz esto; cuando ese resultado esté listo, haz lo otro".

```
useEffect(() => {  
  fetch('https://jsonplaceholder.typicode.com/users')  
    .then((respuesta) => respuesta.json())  
    .then((datos) => setUsuarios(datos));  
}, []);
```

Lee la cadena en voz alta: "pide la URL; cuando llegue la respuesta, conviértela de JSON a objeto JavaScript; cuando tengas los datos, guárdalos en el estado". El primer .then es necesario porque fetch te entrega la respuesta cruda (cabeceras y cuerpo), y respuesta.json() es otro paso asíncrono que parsea el cuerpo. El arreglo vacío [] garantiza que la petición se haga una sola vez al cargar el componente, que es exactamente lo que quieres: no quieres pedir la lista de usuarios cada vez que el usuario mueve el mouse.

La versión moderna del mismo código usa async/await, que se lee como si fuera código normal de arriba hacia abajo. Pero ojo con el error clásico: NO puedes declarar la función del efecto como async directamente, porque useEffect espera que la función devuelva o nada o una función de limpieza, y una función async siempre devuelve una promesa. La forma correcta es definir una función async ADENTRO del efecto y llamarla:

```
useEffect(() => {
  async function cargarUsuarios() {
    const respuesta = await fetch('https://jsonplaceholder.typicode.com/users');
    const datos = await respuesta.json();
    setUsuarios(datos);
  }
  cargarUsuarios();
}, []);
```

Esto es idéntico al .then de arriba, solo que más legible. await significa "espera aquí hasta que la promesa se resuelva y dame el valor". Elige el estilo que prefieras; en la industria verás ambos.

El patrón loading / error / data

Pedir datos a internet toma tiempo: medio segundo, dos segundos, a veces más. Y a veces falla: se cae la red, la API está caída, la URL está mal. Un componente profesional maneja tres estados posibles, y para eso usa tres variables de estado:

```

import { useState, useEffect } from 'react';

function ListaUsuarios() {
  const [usuarios, setUsuarios] = useState([]);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function cargar() {
      try {
        const respuesta = await
fetch('https://jsonplaceholder.typicode.com/users');
        if (!respuesta.ok) {
          throw new Error('La petición falló: ' + respuesta.status);
        }
        const datos = await respuesta.json();
        setUsuarios(datos);
      } catch (e) {
        setError(e.message);
      } finally {
        setCargando(false);
      }
    }
    cargar();
  }, []);

  if (cargando) {
    return <p>Cargando usuarios...</p>;
  }

  if (error) {
    return <p>Ocurrió un error: {error}</p>;
  }

  return (
    <ul>
      {usuarios.map((u) => (
        <li key={u.id}>{u.name} - {u.email}</li>
      ))}
    </ul>
  );
}

```

Este es EL patrón que vas a repetir en cada proyecto que consuma APIs, así que memorízalo: tres estados (datos, cargando, error), try/catch alrededor del fetch, y render condicional que muestra "cargando..." mientras esperas, el mensaje de error si algo falló, y la lista solo cuando los datos ya llegaron. El bloque finally se ejecuta siempre, haya error o no, y ahí apagamos el indicador de carga. Y la verificación `if (!respuesta.ok)` es importante porque fetch NO lanza error cuando el servidor responde 404 o 500: solo falla si ni siquiera pudo conectar. Un 404 ("no encontrado") llega como una respuesta "exitosa" para fetch, así que tienes que revisar `respuesta.ok` a mano y lanzar tu propio error.

Fíjate también en el orden de los `if` del render: primero preguntas si está cargando, luego si hubo error, y al final muestras los datos. Es como un semáforo: no puedes mostrar la lista si todavía no sabes si la petición funcionó.

Enviar datos: fetch con POST

Hasta aquí solo PEDIMOS datos (GET), pero los formularios sirven para ENVIAR datos: registrar un usuario, publicar un comentario, crear una tarea. Para eso fetch acepta un segundo argumento con el método POST y el cuerpo de la petición:


```

async function enviar(e) {
  e.preventDefault();
  setEnviando(true);
  try {
    const respuesta = await fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        title: datos.titulo,
        body: datos.mensaje,
        userId: 1
      })
    });
    if (!respuesta.ok) {
      throw new Error('El servidor rechazó el envío');
    }
    const creado = await respuesta.json();
    console.log('Creado con id:', creado.id);
    setDatos({ titulo: '', mensaje: '' });
  } catch (e) {
    setError(e.message);
  } finally {
    setEnviando(false);
  }
}

```

Tres cosas nuevas: `method: 'POST'` dice que estamos enviando, no pidiendo; `headers` con `Content-Type: application/json` avisa al servidor que el cuerpo va en formato JSON; y `body` lleva los datos convertidos con `JSON.stringify`, porque `fetch` no acepta un objeto directamente: hay que serializarlo a texto. Del otro lado, la respuesta también se parsea con `.json()` como siempre.

Fíjate en el estado `enviando`: mientras la petición viaja, el botón se deshabilita y muestra "Enviando...". Esto evita el clásico doble envío del usuario impaciente que le pica dos veces al botón y crea dos registros. Es un detalle pequeño que en producción vale oro.

La API `jsonplaceholder` acepta el `POST` y te devuelve el objeto creado con un `id` falso, perfecta para practicar. En una API real necesitarías además autenticación (un token en los `headers`), pero el mecanismo es el mismo.

Y un recordatorio de seguridad que debes grabarte desde ya: todo lo que validas en el formulario también debe validarlo el servidor. La validación en React es para darle buena experiencia al usuario; la validación en el servidor es la que protege los datos de verdad, porque cualquiera puede abrir las herramientas del desarrollador y saltarse tu formulario. Nunca confíes solo en el frontend.

Cómo depurar efectos: tu checklist mental

Cuando un efecto se porta mal, no adivines: sigue está lista en orden y encontrarás el problema en minutos.

Paso 1: ¿cuántas veces corre el efecto? Pon un `console.log` como primera línea del efecto y mira la consola. Si se repite sin parar, tienes un bucle: casi seguro una dependencia que cambia en cada render (un objeto creado en el cuerpo, o el mismo estado que el efecto actualiza). Si no corre nunca después del montaje, revisa que la dependencia que esperas esté en el arreglo.

Paso 2: ¿qué valores ve el efecto? Agrega otro `console.log` con las dependencias dentro del efecto. Si ves un valor viejo (por ejemplo, el texto del buscador de hace dos teclas), te falta una dependencia en el arreglo o estás leyendo un closure desactualizado. La advertencia del linter de React sobre dependencias faltantes existe justo para esto: no la ignores.

Paso 3: ¿el cleanup hace su trabajo? Si los síntomas son "cada vez va más rápido" (reloj acelerado), "el manejador se dispara varias veces" (listeners fantasma) o "la app se alenta con el uso", el cleanup falta o está mal. Verifica que la función que devuelve el efecto realmente apague lo que el efecto encendió, con el identificador correcto.

Paso 4: ¿es StrictMode? Si el efecto corre exactamente dos veces al montar en desarrollo pero una sola en producción, no hay bug: es el doble montaje intencional. No lo "arregles" con banderas; asegúrate de que el cleanup deje todo limpio y sigue adelante.

Paso 5: ¿la petición es la que crees? Abre la pestaña Network de las herramientas del desarrollador y mira la URL exacta que se pidió, el método y la respuesta del servidor. La mitad de los "bugs de fetch" son URLs mal armadas (un parámetro undefined colado en la cadena, mayúsculas donde la API quiere minúsculas) o respuestas 404 que fetch no reporta como error.

Un hábito que te va a ahorrar horas: mientras desarrollas, deja los `console.log` de depuración dentro del efecto y quítalos al final, o mejor, usa un estado de "debug" que muestre en pantalla los valores clave. Ver el estado real en pantalla es más rápido que adivinar qué está pasando.

Errores comunes (y cómo evitarlos)

Vamos a repasar los errores que comete todo el mundo al aprender esta unidad, para que los reconozcas cuando te pasen:

1. Olvidar `e.preventDefault()`. El formulario se envía, la página se recarga, el estado se pierde y parece que "React no funciona". Solución: primera línea de todo `onSubmit`.
2. Bucle infinito por dependencias mal puestas. El caso típico: pones el estado que actualizas dentro del propio arreglo de dependencias.

```
const [datos, setDatos] = useState([]);
useEffect(() => {
  fetch(url).then(r => r.json()).then(setDatos);
}, [datos]); // MAL: datos cambia -> el efecto se repite -> cambia datos -> ...
```

El efecto corre, actualiza datos, como datos cambió el efecto se repite, actualiza datos... infinito. La petición a la API se dispara cientos de veces por segundo. Solución: si el fetch solo debe correr al montar, usa `[]`.

3. El caso contrario: dependencias faltantes. Usas una variable dentro del efecto pero no la pones en el arreglo, y el efecto trabaja con un valor viejo (stale closure). Ejemplo: un buscador que hace fetch con el texto, pero el arreglo está vacío: siempre busca el texto inicial. Solución: incluye la variable en el arreglo, o mejor, sigue la advertencia del linter.
4. `async` directo en la función de `useEffect`. `useEffect(async () => {...})` devuelve una promesa donde React espera una función de limpieza, y React se queja. Solución: función `async` interna que se invoca.
5. No manejar el estado de carga. El componente intenta dibujar `usuarios.map` cuando `usuarios` todavía es `undefined` o la petición no ha terminado, y la app truena con "cannot read property map of undefined". Solución: el patrón `loading/error/data`, siempre.
6. Olvidar el spread al actualizar un objeto de estado en formularios. Cada tecla borra los demás campos. Solución: `...datos` siempre antes de la propiedad actualizada.
7. Olvidar el `cleanup` en intervalos y suscripciones. Relojes acelerados y listeners fantasma que se acumulan. Solución: si el efecto enciende algo, devuelve una función que lo apague.
8. Poner el manejador en `onClick` del botón en lugar de `onSubmit` del formulario. El Enter deja de funcionar y los usuarios piensan que el formulario está roto. Solución: `onSubmit` en el `<form>`.

Buenas prácticas para cerrar

Primera: un efecto, una responsabilidad. Si un componente necesita pedir usuarios al montar y también actualizar el título cuando cambia un contador, usa dos `useEffect` separados, no uno gigante que hace todo. Es más fácil de leer y cada uno tiene sus propias dependencias.

Segunda: no uses `useEffect` para calcular valores que puedes derivar del estado durante el render. Si tienes nombre y apellido en el estado, el nombre completo es `const completo = nombre + ' ' + apellido` en el cuerpo del componente, no un efecto que actualiza otro estado. Los efectos son para sincronizar con sistemas externos, no para hacer cuentas.

Tercera: nombra tus manejadores de formulario con claridad (`enviar`, `manejarCambio`, `manejarEnvio`) y mantén la lógica de validación en funciones separadas para poder probarlas.

Cuarta: cuando el `fetch` depende de lo que el usuario escribe (un buscador), considera esperar un poco antes de pedir (`debounce`) para no bombardear la API con cada tecla. Es un tema avanzado, pero que sepas que existe: con un `setTimeout` y su `cleanup` dentro del efecto lo puedes implementar cuando lo necesites.

Quinta: muestra siempre retroalimentación. Un botón que al enviarse deshabilita y dice "Enviando..." vale oro en experiencia de usuario: el usuario sabe que su clic sí se registró y no le pica diez veces generando diez peticiones.

Sexta: cuando un formulario crezca mucho (más de seis u ocho campos, validaciones condicionales, campos que aparecen y desaparecen), considera seriamente una librería como `React Hook Form`. No porque no puedas hacerlo a mano, sino porque el costo de mantener validación manual crece con cada campo y las librerías ya resolvieron esos problemas. La regla práctica es: formularios chicos a mano para aprender y controlar todo; formularios grandes con librería para no reinventar la rueda.

Séptima y última: documenta con nombres claros qué hace cada efecto con un comentario de una línea encima (`// pide usuarios al montar`, `// sincroniza título con contador`). Dentro de seis meses, cuando vuelvas a ese código, ese comentario valdrá más que cualquier otra documentación.

Con esto tienes la teoría completa. Ahora vamos a verla en acción con ejemplos comentados línea por línea.

Mapa mental: ¿estado, efecto o manejador de evento?

Para cerrar la explicación, aquí tienes el mapa de decisiones que distingue a quien entiende la unidad de quien solo copió los ejemplos. Ante cualquier funcionalidad nueva, pregúntate en este orden:

Pregunta 1: ¿el dato lo escribe o lo elige el usuario? Entonces va en un estado (useState) conectado a un input controlado. El formulario es el puente de entrada y el estado es donde vive lo que el usuario dijo.

Pregunta 2: ¿algo debe pasar automáticamente cuando el componente aparece o cuando un dato cambia, sin que el usuario haga clic? Entonces va en un useEffect. Ejemplos: pedir la lista inicial al montar ([[]]), actualizar el título cuando cambia el contador ([num]), guardar en localStorage cuando cambia la lista ([tareas]). La palabra clave es "automáticamente": si nadie lo dispara, es un efecto.

Pregunta 3: ¿algo debe pasar como respuesta directa a una acción del usuario (clic, envío de formulario)? Entonces va en un manejador de evento (onClick, onSubmit), NO en un efecto. El buscador de Pokémon del ejercicio 5 es el ejemplo perfecto: la petición se dispara al enviar el formulario, así que el fetch vive en la función buscar, no en un useEffect. Meter un fetch disparado por clic dentro de un useEffect es uno de los errores de diseño más comunes: funciona, pero enreda las dependencias y hace el código difícil de seguir.

Pregunta 4: ¿el valor se puede calcular a partir de datos que ya tengo? Entonces es un valor derivado y se calcula durante el render, sin estado extra y sin efecto. Los caracteres restantes del ejercicio 4, el nombre completo a partir de nombre y apellido, el total de un carrito a partir de sus items: todo eso son const calculadas en el cuerpo del componente. Cada estado o efecto innecesario es una fuente potencial de bugs, porque es algo más que puede desincronizarse.

Memoriza este orden: estado para lo que el usuario dice, efecto para lo que pasa solo, evento para lo que el usuario dispara, cálculo directo para lo que se deriva. Cuando dudes entre efecto y evento, pregunta "¿quién lo dispara?": si la respuesta es "nadie, debe pasar solo", es efecto; si es "el usuario con un clic o un envío", es evento. Con este mapa, el 90 por ciento de tus decisiones de diseño en React se vuelven obvias.

Ejemplos paso a paso

Ejemplo 1: Formulario de registro con tres campos

Este ejemplo junta todo lo de formularios: un solo objeto de estado, la técnica de `[e.target.name]`, `preventDefault` y limpieza tras enviar. Es el formulario típico de "crear cuenta".

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

function FormularioRegistro() {
  // 1. Un solo objeto guarda los tres campos del formulario.
  const [datos, setDatos] = useState({
    nombre: '',
    correo: '',
    password: ''
  });
  // 2. Estado para el mensaje de confirmación tras el envío.
  const [mensaje, setMensaje] = useState('');

  // 3. Un solo manejador para los tres inputs.
  function manejarCambio(e) {
    setDatos({
      ...datos, // 4. Copia todos los campos anteriores...
      [e.target.name]: e.target.value // 5. ...y actualiza solo el que cambió.
    });
  }

  // 6. Manejador del envío del formulario.
  function enviar(e) {
    e.preventDefault(); // 7. Evita la recarga de la página.
    setMensaje('Cuenta creada para ' + datos.nombre);
    // 8. Limpia el formulario: los inputs controlados se vacían solos.
    setDatos({ nombre: '', correo: '', password: '' });
  }

  return (
    <form onSubmit={enviar}>
      <h2>Crear cuenta</h2>
      <input
        type="text"
        name="nombre" // 9. El name coincide con la propiedad del estado.
        value={datos.nombre} // 10. Input controlado: value siempre viene del
estado.
        onChange={manejarCambio} // 11. Cada tecla actualiza el estado.
        placeholder="Nombre completo"
      />
      <input
        type="email"
```

```

        name="correo"
        value={datos.correo}
        onChange={manejarCambio}
        placeholder="correo@ejemplo.com"
      />
      <input
        type="password"
        name="password"
        value={datos.password}
        onChange={manejarCambio}
        placeholder="Contraseña"
      />
      <button type="submit">Registrarse</button>
      {mensaje && <p>{mensaje}</p>}
    </form>
  );
}

```

Línea por línea: en 1 creamos el objeto de estado con los tres campos vacíos. En 2, un estado aparte para el mensaje de éxito, porque no es parte de los datos del formulario. La función manejarCambio (3) es la única que necesitan los tres inputs: en 4 el spread copia el objeto anterior y en 5 los corchetes convierten e.target.name en el nombre de la propiedad a actualizar. Si escribes en el input de correo, e.target.name es "correo" y solo esa propiedad cambia.

En 6 empieza el envío: 7 frena la recarga con preventDefault (sin esto, todo lo demás sería inútil porque la página se recargaría), luego guardamos el mensaje de confirmación usando el nombre antes de borrarlo, y en 8 reseteamos el objeto. Como los inputs son controlados (10), al vaciarse el estado se vacían en pantalla automáticamente.

Del 9 al 11 está el patrón que se repite en cada input: name igual a la propiedad del estado, value atado al estado, onChange al manejador común. Si mañana agregas un campo "teléfono", solo añades la propiedad al estado inicial y un input con name="teléfono": el manejador ya sabe qué hacer sin que lo toques. Esa es la belleza del truco.

Ejemplo 2: Contador que actualiza el título del documento

El ejemplo clásico de useEffect con dependencias: cada vez que el contador cambia, el título de la pestaña del navegador se actualiza. Es un efecto secundario puro (toca el DOM fuera de React) y muestra el arreglo [num] en acción.

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

function ContadorTitulo() {
  // 1. Estado del contador.
  const [num, setNum] = useState(0);

  // 2. Efecto que sincroniza el título con el contador.
  useEffect(() => {
    document.title = 'Clicks: ' + num; // 3. Efecto secundario: toca el DOM
    externo.
  }, [num]); // 4. Se repite solo cuando num cambia.

  return (
    <div>
      <p>Has hecho clic {num} veces</p>
      <button onClick={() => setNum(num + 1)}>Sumar</button>
      <button onClick={() => setNum(0)}>Reiniciar</button>
    </div>
  );
}
```

En 1 está el estado normal. En 2 declaramos el efecto: una función y el arreglo [num]. En 3, el código del efecto modifica document.title, algo que React no controla: es exactamente lo que significa "efecto secundario". En 4, el arreglo de dependencias dice "corre al montar y cada vez que num cambie". Si el usuario hace clic en "Reiniciar", num vuelve a 0, el componente se renderiza, y después del render el efecto actualiza el título a "Clicks: 0".

¿Qué pasaría con [] en lugar de [num]? El título se pondría en "Clicks: 0" al cargar y jamás se actualizaría, porque el efecto solo correría una vez con el valor inicial. ¿Y sin arreglo? Funcionaría, pero el efecto correría en cada render, incluso en renders que nada tienen que ver con num. [num] es la opción precisa: ni de más ni de menos.

Ejemplo 3: Lista de usuarios desde una API con estado de carga

Aquí aplicamos el patrón completo loading/error/data con fetch dentro de useEffect y arreglo vacío, porque la lista se pide una sola vez al cargar.

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

function ListaUsuarios() {
  // 1. Los tres estados del patrón: datos, cargando y error.
  const [usuarios, setUsuarios] = useState([]);
  const [cargando, setCargando] = useState(true);
```



```

const [error, setError] = useState(null);

// 2. El efecto corre una sola vez al montar el componente.
useEffect(() => {
  // 3. Función async interna: el efecto mismo no puede ser async.
  async function cargarUsuarios() {
    try {
      // 4. Petición a la API pública de práctica.
      const respuesta = await fetch('https://jsonplaceholder.typicode.com/users');
      // 5. fetch no falla con 404/500: hay que revisar ok a mano.
      if (!respuesta.ok) {
        throw new Error('Error del servidor: ' + respuesta.status);
      }
      // 6. Convierte el cuerpo JSON a objetos JavaScript.
      const datos = await respuesta.json();
      setUsuarios(datos);      // 7. Guarda los datos en el estado.
    } catch (e) {
      setError(e.message);    // 8. Si algo falló, guarda el mensaje.
    } finally {
      setCargando(false);    // 9. Siempre apaga el indicador de carga.
    }
  }
  cargarUsuarios();          // 10. Invoca la función async.
}, []);                      // 11. Array vacío: solo al montar.

// 12. Render condicional: primero el estado de carga...
if (cargando) {
  return <p>Cargando usuarios, espera un momento...</p>;
}
// 13. ...luego el estado de error...
if (error) {
  return <p>Algo salió mal: {error}</p>;
}
// 14. ...y al final los datos, que ya existen seguro.
return (
  <div>
    <h2>Usuarios registrados</h2>
    <ul>
      {usuarios.map((u) => (
        <li key={u.id}>
          {u.name} ({u.email})
        </li>
      ))}
    </ul>
  </div>

```

```
);  
}
```

En 1 preparamos los tres estados: usuarios empieza como arreglo vacío para que `.map` no trueene, cargando empieza en `true` porque la petición arranca de inmediato, error en `null` porque aún no hay error. En 2 el efecto con `[]` garantiza una sola petición. La función `cargarUsuarios` (3) es `async` porque usa `await`; recuerda que no puedes marcar como `async` la función que le pasas a `useEffect` directamente, por eso se define adentro y se invoca en 10.

En 4 se hace la petición; en 5 verificamos `respuesta.ok` porque un 404 no lanza excepción por sí solo; en 6 parseamos el JSON; en 7 guardamos. Si cualquier `await` falla (sin internet, servidor caído), el `catch` de 8 guarda el mensaje. El `finally` de 9 apaga "cargando" pase lo que pase: sin esto, un error dejaría a la app atorada en "Cargando..." para siempre.

Del 12 al 14, el render en orden de semáforo: si está cargando, muestra el mensaje; si no, pero hay error, muestra el error; si no, dibuja la lista con `.map` y `key={u.id}` como ya aprendiste. Este esqueleto te sirve para cualquier API: cambia la URL y los campos, y el resto es igual.

Ejemplo 4: Buscador que hace fetch al cambiar el texto

Este ejemplo combina formulario y efecto: un input controlado guarda el texto buscado, y un `useEffect` con `[texto]` dispara una petición cada vez que el texto cambia. Es el corazón de cualquier buscador.

```
import { useState, useEffect } from 'react';  
  
function BuscadorPosts() {  
  // 1. El texto del input, controlado por React.  
  const [texto, setTexto] = useState('');  
  // 2. Resultados de la búsqueda.  
  const [posts, setPosts] = useState([]);  
  const [cargando, setCargando] = useState(false);  
  
  // 3. Efecto que depende del texto: corre cada vez que cambia.  
  useEffect(() => {  
    // 4. Si el input está vacío, limpia resultados y no pidas nada.  
    if (texto.trim() === '') {  
      setPosts([]);  
      return;  
    }  
  
    async function buscar() {  
      setCargando(true);  
      const respuesta = await fetch(  

```

```

    'https://jsonplaceholder.typicode.com/posts?userId=' + texto
  );
  const datos = await respuesta.json();
  setPosts(datos);
  setCargando(false);
}
buscar();
}, [texto]); // 5. La dependencia es el texto del input.

return (
  <div>
    <h2>Buscar posts por usuario</h2>
    <input
      type="text"
      value={texto}
      onChange={(e) => setTexto(e.target.value)}
      placeholder="Escribe un número de usuario (1-10)"
    />
    {cargando && <p>Buscando...</p>}
    <ul>
      {posts.map((p) => (
        <li key={p.id}>{p.title}</li>
      ))}
    </ul>
  </div>
);
}

```

En 1 el input controlado de siempre. En 3 el efecto observa texto: cada tecla que el usuario escribe actualiza el estado, eso provoca un render, y después del render React ve que texto cambió y repite el efecto. En 4 hay un guardián: si el texto está vacío no tiene sentido pedirle nada a la API, así que limpiamos y salimos con return temprano.

La petición usa un query param (?userId=) para filtrar en el servidor: la API de jsonplaceholder acepta ese filtro y devuelve solo los posts de ese usuario. En 5 confirmamos la dependencia: sin [texto], el buscador jamás reaccionaría a lo que escribes (el error de "dependencias faltantes").

Nota honesta: en una app real no harías una petición por cada tecla sin más; usarías la técnica de debounce (esperar a que el usuario pause). Pero el mecanismo base es exactamente este: estado -> efecto con dependencia -> fetch -> render de resultados. Domina esto y el debounce será un adorno fácil de agregar después.

Práctica

Ejercicio 1: Formulario de contacto con validación de campos vacíos

Enunciado: crea un formulario de contacto con tres campos (nombre, correo y mensaje) que no permita enviar si algún campo está vacío o tiene solo espacios. Muestra un mensaje de error debajo de cada campo inválido y, si todo es válido, muestra un mensaje de éxito y limpia el formulario.

Solución:

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

function FormularioContacto() {
  const [datos, setDatos] = useState({ nombre: '', correo: '', mensaje: '' });
  const [errores, setErrores] = useState({});
  const [enviado, setEnviado] = useState(false);

  function manejarCambio(e) {
    setDatos({ ...datos, [e.target.name]: e.target.value });
  }

  function enviar(e) {
    e.preventDefault();
    // 1. Construye el objeto de errores campo por campo.
    const nuevosErrores = {};
    if (datos.nombre.trim() === '') {
      nuevosErrores.nombre = 'Escribe tu nombre';
    }
    if (datos.correo.trim() === '') {
      nuevosErrores.correo = 'Escribe tu correo';
    }
    if (datos.mensaje.trim() === '') {
      nuevosErrores.mensaje = 'Escribe tu mensaje';
    }
    setErrores(nuevosErrores);
    // 2. Si no hay errores, el envío es válido.
    if (Object.keys(nuevosErrores).length === 0) {
      setEnviado(true);
      setDatos({ nombre: '', correo: '', mensaje: '' });
    } else {
      setEnviado(false);
    }
  }
}
```

```

return (
  <form onSubmit={enviar}>
    <h2>Contacto</h2>
    <input
      type="text"
      name="nombre"
      value={datos.nombre}
      onChange={manejarCambio}
      placeholder="Tu nombre"
    />
    {errores.nombre && <p className="error">{errores.nombre}</p>}
    <input
      type="email"
      name="correo"
      value={datos.correo}
      onChange={manejarCambio}
      placeholder="Tu correo"
    />
    {errores.correo && <p className="error">{errores.correo}</p>}
    <textarea
      name="mensaje"
      value={datos.mensaje}
      onChange={manejarCambio}
      placeholder="Tu mensaje"
    />
    {errores.mensaje && <p className="error">{errores.mensaje}</p>}
    <button type="submit">Enviar mensaje</button>
    {enviado && <p>Mensaje enviado. Gracias por contactarnos.</p>}
  </form>
);
}

```

Explicación: usamos el patrón de objeto único de estado con `[e.target.name]`, que ya cubre también al textarea (funciona igual que un input). La validación usa `trim()` para atrapar los campos con solo espacios, que a simple vista parecen llenos pero están vacíos. Los errores se guardan en su propio estado y se muestran con render condicional (`{errores.nombre && ...}`). El truco de `Object.keys(nuevosErrores).length === 0` decide si el formulario pasó: un objeto sin propiedades significa cero errores. Solo entonces mostramos el mensaje de éxito y limpiamos. Nota que limpiamos también los errores indirectamente: en el siguiente envío se recalculan desde cero.

Ejercicio 2: Reloj digital con useEffect, intervalo y cleanup

Enunciado: crea un componente Reloj que muestre la hora actual actualizándose cada segundo. Usa useEffect con un intervalo y no olvides la función de limpieza para que no queden intervalos corriendo cuando el componente se desmonte.

Solución:

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

function Reloj() {
  const [hora, setHora] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    // 1. Al montar, arranca un intervalo que actualiza la hora cada segundo.
    const id = setInterval(() => {
      setHora(new Date().toLocaleTimeString());
    }, 1000);

    // 2. Cleanup: al desmontar, apaga el intervalo.
    return () => {
      clearInterval(id);
      console.log('Reloj desmontado, intervalo apagado');
    };
  }, []); // 3. Array vacío: el intervalo se crea una sola vez.

  return (
    <div>
      <h2>Hora actual</h2>
      <p>{hora}</p>
    </div>
  );
}
```

Explicación: el estado inicial se calcula una vez con `new Date().toLocaleTimeString()`. El efecto con `[]` corre solo al montar: crea el intervalo y guarda su identificador en `id`. Cada segundo, el callback del intervalo pide la hora nueva y actualiza el estado, lo que repinta el componente. La función que devuelve el efecto es el `cleanup`: `clearInterval(id)` mata el intervalo. ¿Por qué importa? Imagina que este reloj vive dentro de una pestaña que el usuario puede cerrar: sin `cleanup`, el intervalo seguiría ejecutándose en segundo plano, actualizando el estado de un componente que ya no existe. React te lanzará advertencias y, peor, cada vez que el usuario abra y cierre la pestaña se acumulará un intervalo más. Con `[]` el `cleanup` solo corre al desmontar; si las dependencias cambiaran, correría también antes de cada repetición. El

console.log del cleanup es opcional, pero útil mientras aprendes: te permite ver en la consola exactamente cuándo React limpia.

Ejercicio 3: Lista de posts desde jsonplaceholder

Enunciado: al cargar el componente, pide los posts de <https://jsonplaceholder.typicode.com/posts> y muéstralos en una lista con su título. Maneja los tres estados: muestra "Cargando..." mientras pide, un mensaje de error si falla, y la lista cuando los datos lleguen. Muestra solo los primeros 10 posts.

Solución:

```

import { useState, useEffect } from 'react';

function ListaPosts() {
  const [posts, setPosts] = useState([]);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function cargarPosts() {
      try {
        const respuesta = await
fetch('https://jsonplaceholder.typicode.com/posts');
        if (!respuesta.ok) {
          throw new Error('Servidor respondió ' + respuesta.status);
        }
        const datos = await respuesta.json();
        setPosts(datos.slice(0, 10)); // 1. Solo los primeros 10.
      } catch (e) {
        setError(e.message);
      } finally {
        setCargando(false);
      }
    }
    cargarPosts();
  }, []);

  if (cargando) {
    return <p>Cargando posts...</p>;
  }
  if (error) {
    return <p>Error al cargar: {error}</p>;
  }
  return (
    <div>
      <h2>Últimos posts</h2>
      <ul>
        {posts.map((p) => (
          <li key={p.id}>{p.title}</li>
        ))}
      </ul>
    </div>
  );
}

```


Explicación: es el patrón loading/error/data aplicado tal cual. La API devuelve 100 posts, así que en 1 usamos `datos.slice(0, 10)` para quedarnos con los primeros diez y no saturar la pantalla. El resto es mecánico: `try/catch/finally`, verificación de respuesta.ok (recuerda que `fetch` no truena con un 404), y el render en orden de semáforo. Si quieres practicar más, cambia la URL a `/posts/1` para pedir un solo post, o a `/comments` para ver otra colección. La estructura no cambia: solo la URL y los campos que dibujas.

Ejercicio 4: Contador de caracteres restantes en un textarea

Enunciado: crea un textarea con un límite de 200 caracteres que muestre en tiempo real cuántos caracteres quedan disponibles. Cuando queden 20 o menos, el contador debe verse en rojo. El botón de enviar debe deshabilitarse si el texto está vacío.

Solución:

```

import { useState } from 'react';

function CajaComentario() {
  const LIMITE = 200;
  const [texto, setTexto] = useState('');
  // 1. Valor derivado: no necesita su propio estado ni useEffect.
  const restantes = LIMITE - texto.length;
  const casiLleno = restantes <= 20;

  function enviar(e) {
    e.preventDefault();
    alert('Comentario enviado: ' + texto);
    setTexto('');
  }

  return (
    <form onSubmit={enviar}>
      <h2>Deja tu comentario</h2>
      <textarea
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        maxLength={LIMITE} // 2. El navegador impide escribir más del límite.
        rows={4}
        placeholder="Escribe aquí (máximo 200 caracteres)"
      />
      <p style={{ color: casiLleno ? 'red' : 'black' }}>
        Te quedan {restantes} caracteres
      </p>
      <button type="submit" disabled={texto.trim() === ''}>
        Publicar
      </button>
    </form>
  );
}

```

Explicación: este ejercicio enseña algo importante: NO todo necesita useEffect. En 1, restantes se calcula directamente durante el render a partir del estado: es un valor derivado, y la regla dice que los valores derivados se calculan en el cuerpo del componente, no en un efecto. Cada vez que el usuario escribe, el estado cambia, el componente se renderiza y restantes se recalcula solo. En 2, el atributo maxLength le pone el freno al navegador para que ni siquiera deje escribir de más. El color del contador cambia con un ternario en el style, y el botón se deshabilita con disabled={texto.trim() === ''} para que no se pueda publicar un comentario vacío o de puros espacios. Este es el tipo de detalle que separa un formulario amateur de uno

profesional: el usuario siempre sabe cuántos caracteres le quedan y nunca puede romper la regla.

Ejercicio 5: Fetch de un Pokémon por nombre con manejo de error

Enunciado: crea un buscador donde el usuario escriba el nombre de un Pokémon (por ejemplo "pikachu") y al enviar el formulario se pidan sus datos a <https://pokeapi.co/api/v2/pokemon/{nombre}>. Muestra su nombre, su imagen y su tipo. Si el nombre no existe, muestra un mensaje de error amable. Incluye estado de carga.

Solución:

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

function BuscadorPokemon() {
  const [nombre, setNombre] = useState('');
  const [pokemon, setPokemon] = useState(null);
  const [cargando, setCargando] = useState(false);
  const [error, setError] = useState(null);

  async function buscar(e) {
    e.preventDefault();
    // 1. Guarda: si no escribió nada, no pidas nada.
    if (nombre.trim() === '') {
      return;
    }
    setCargando(true);
    setError(null);
    setPokemon(null);
    try {
      // 2. La API distingue mayúsculas: normaliza a minúsculas.
      const url = 'https://pokeapi.co/api/v2/pokemon/' + nombre.trim().toLowerCase();
      const respuesta = await fetch(url);
      if (!respuesta.ok) {
        // 3. Un 404 aquí significa "ese Pokémon no existe".
        throw new Error('No encontramos a "' + nombre + '". Revisa el nombre.');
```

```

return (
  <div>
    <h2>Buscador Pokémon</h2>
    <form onSubmit={buscar}>
      <input
        type="text"
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
        placeholder="Ej: pikachu"
      />
      <button type="submit" disabled={cargando}>
        {cargando ? 'Buscando...' : 'Buscar'}
      </button>
    </form>

    {cargando && <p>Buscando en la Pokédex...</p>}
    {error && <p>{error}</p>}
    {pokemon && (
      <div>
        <h3>{pokemon.name}</h3>
        <img src={pokemon.sprites.front_default} alt={pokemon.name} />
        <p>Tipo: {pokemon.types.map((t) => t.type.name).join(', ')}</p>
        <p>Altura: {pokemon.height} | Peso: {pokemon.weight}</p>
      </div>
    )}
  </div>
);
}

```

Explicación: aquí el fetch no va dentro de useEffect sino dentro del manejador del submit, y eso es correcto: la petición debe ocurrir cuando el usuario la pide (al enviar), no al montar ni al cambiar un estado. useEffect es para sincronizar con el exterior de forma automática; un botón de "buscar" es una acción del usuario y va en un manejador de evento. Esa distinción es clave y muchos la confunden.

En 1 evitamos pedir con texto vacío. En 2 normalizamos a minúsculas porque la API es sensible a mayúsculas ("Pikachu" fallaría, "pikachu" funciona). En 3 convertimos el 404 en un mensaje amable: en lugar de "Error 404", el usuario lee "No encontramos a ...". Antes de cada búsqueda reseteamos error y pokemon para no mostrar resultados viejos junto al indicador de carga. El botón se deshabilita mientras carga y cambia su texto a "Buscando...", la retroalimentación que mencionamos en las buenas prácticas. Para dibujar los tipos usamos `pokemon.types.map((t) => t.type.name).join(', ')`: cada tipo viene anidado como `{ type: { name: 'electric' } }`, así que hay que entrar dos niveles. Prueba con "pikachu", "charizard" y luego con un nombre inventado como "juanito" para ver el manejo del error en acción.

Ejercicio 6: Lista de tareas con persistencia en localStorage

Enunciado: crea una lista de tareas (to-do) donde puedas agregar tareas con un formulario y marcarlas como completadas con un checkbox. Las tareas deben guardarse en localStorage para que no se pierdan al recargar la página: usa un useEffect para cargarlas al montar y otro para guardarlas cada vez que cambien.

Solución:

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

function ListaTareas() {
  const [tareas, setTareas] = useState([]);
  const [texto, setTexto] = useState('');

  // 1. Al montar, lee las tareas guardadas (si hay).
  useEffect(() => {
    const guardadas = localStorage.getItem('mis-tareas');
    if (guardadas) {
      setTareas(JSON.parse(guardadas));
    }
  }, []);

  // 2. Cada vez que tareas cambia, guárdalas.
  useEffect(() => {
    localStorage.setItem('mis-tareas', JSON.stringify(tareas));
  }, [tareas]);

  function agregar(e) {
    e.preventDefault();
    if (texto.trim() === '') {
      return;
    }
  }
}
```

```

// 3. Nueva tarea con id único basado en la fecha.
const nueva = {
  id: Date.now(),
  texto: texto.trim(),
  completada: false
};
setTareas([...tareas, nueva]);
setTexto('');
}

function alternar(id) {
  // 4. Crea un array nuevo marcando solo la tarea indicada.
  setTareas(
    tareas.map((t) =>
      t.id === id ? { ...t, completada: !t.completada } : t
    )
  );
}

function eliminar(id) {
  setTareas(tareas.filter((t) => t.id !== id));
}

return (
  <div>
    <h2>Mis tareas</h2>
    <form onSubmit={agregar}>
      <input
        type="text"
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        placeholder="Nueva tarea"
      />
      <button type="submit">Agregar</button>
    </form>
    <ul>
      {tareas.map((t) => (
        <li key={t.id}>
          <input
            type="checkbox"
            checked={t.completada}
            onChange={() => alternar(t.id)}
          />
          <span style={{ textDecoration: t.completada ? 'line-through' : 'none' }}>
            {t.texto}

```

```

        </span>
        <button onClick={() => eliminar(t.id)}>X</button>
      </li>
    )}
  </ul>
  {tareass.length === 0 && <p>No hay tareas. Agrega la primera.</p>}
</div>
);
}

```

Explicación: este ejercicio es la cereza del pastel porque combina TODO lo de la unidad: formulario con preventDefault y validación de vacío, inputs controlados, y dos useEffect con dependencias distintas trabajando en equipo. El efecto 1 (con []) lee localStorage una sola vez al montar: localStorage.getItem devuelve texto o null, y JSON.parse lo convierte de texto a arreglo. El efecto 2 (con [tareass]) escribe en localStorage cada vez que la lista cambia: JSON.stringify convierte el arreglo a texto porque localStorage solo guarda strings.

Nota el orden de ejecución al cargar por primera vez: el efecto 1 lee las tareas guardadas y actualiza el estado; eso dispara un render y luego el efecto 2, que guarda... lo mismo que acababa de leer. No pasa nada malo: simplemente sobrescribe con datos idénticos. Es un ejemplo perfecto de efectos que cooperan sin estorbarse.

En 3, Date.now() da un id único (milisegundos actuales) sin librerías. En 4, el .map crea un arreglo nuevo donde solo la tarea indicada invierte su completada, usando spread para no mutar el objeto original: en React nunca se muta el estado directamente, siempre se crean copias. El tachado con textDecoration: line-through es el toque visual que le dice al usuario "esta ya quedó". Recarga la página y comprueba que tus tareas siguen ahí: ese es el poder de combinar estado, efectos y almacenamiento local.

Resumen de la unidad

En esta unidad aprendiste las dos habilidades que convierten tus componentes en aplicaciones de verdad: escuchar al usuario con formularios y hablar con el mundo exterior con efectos secundarios.

De formularios te llevas el patrón completo: el evento onSubmit en la etiqueta <form> (no onClick en el botón, para que el Enter funcione), e.preventDefault() como primera línea para evitar la recarga de la página (ese es el comportamiento por defecto heredado de la web de los noventa), y los inputs controlados donde el estado de React es la única fuente de la verdad. Aprendiste a manejar muchos campos con un solo objeto de estado gracias al truco [e.target.name] combinado con el spread (...datos), a limpiar el formulario reseteando el estado

tras el envío, y a validar a mano con `trim()`, mensajes de error por campo y la comprobación de objeto vacío con `Object.keys().errores.length`.

De `useEffect` te llevas su idea central: es la herramienta oficial para el código que corre después del render y que interactúa con sistemas fuera de React (título del documento, temporizadores, suscripciones, peticiones de red). Dominas sus tres modos según el arreglo de dependencias: sin arreglo (cada render), `[]` (solo al montar) y `[dep]` (al montar y cuando la dependencia cambie), además de la regla de oro de incluir en el arreglo todo valor del componente que uses dentro del efecto. Conoces la función de limpieza para apagar lo que el efecto enciende (`clearInterval`, `removeEventListener`) y entiendes por qué `StrictMode` ejecuta los efectos dos veces en desarrollo: para revelar efectos mal escritos antes de que lleguen a producción.

De `fetch` te llevas cómo pedir datos a una API REST con `fetch` y promesas (`.then`) o `async/await` (con la función `async` definida dentro del efecto, nunca el efecto mismo como `async`), por qué hay que revisar `respuesta.ok` a mano (`fetch` no falla con 404/500), y el patrón `loading/error/data` con `try/catch/finally` y render condicional en orden de semáforo, que repetirás en cada proyecto que consuma APIs.

También viste la distinción fina pero crucial: el `fetch` va en `useEffect` cuando la petición debe ocurrir automáticamente (al montar o al cambiar un dato), y va en un manejador de evento cuando la dispara una acción del usuario (como el botón de buscar). Y aprendiste que no todo necesita un efecto: los valores que se pueden calcular del estado durante el render (como los caracteres restantes) se calculan ahí, sin `useEffect`.

Los errores que ya no te van a sorprender: olvidar `preventDefault`, el bucle infinito por dependencias mal puestas, el closure desactualizado por dependencias faltantes, el `async` directo en `useEffect`, el render que truena por no manejar la carga, el spread olvidado que borra campos, el `cleanup` ausente que acelera relojes y multiplica listeners, y el `onClick` en el botón que rompe el Enter.

Con formularios, efectos y `fetch` dominados, ya puedes construir el ciclo completo de una app moderna: el usuario escribe, validas, pides datos, muestras carga, presentas el resultado o un error amable, y limpias al desmontar. En la siguiente unidad seguiremos subiendo de nivel con más hooks y patrones que usan los profesionales todos los días. Sigue practicando: la teoría de esta unidad se vuelve instinto después de tres o cuatro proyectos reales.

Como guía rápida para llevar, estos son los cinco patrones que debes poder escribir de memoria: el formulario controlado con `onSubmit` y `preventDefault`; el objeto único de estado con `[e.target.name]` y `spread`; el `useEffect` con `[]` para cargar datos al montar; el `useEffect` con `[dep]` para reaccionar a cambios; y el trío `loading/error/data` con `try/catch/finally` para cada petición. Si dominas esos cinco, dominas la unidad.

UNIDAD 6: Composición de componentes

Imagina que quieres construir una casa. Tienes dos opciones. La primera: contratar a un solo albañil y pedirle que haga todo él solito, desde los cimientos hasta el techo, pasando por la plomería, la electricidad y la pintura. Un solo trabajador, un solo plano gigante, todo mezclado en su cabeza. La segunda: contratar un equipo donde cada quien es experto en una cosa. Unos hacen los cimientos, otros levantan los muros, otros instalan la tubería, otros la luz. Cada equipo trabaja en su parte, con su propio plano pequeño y claro, y al final todas las partes encajan para formar la casa completa.

¿Cuál casa crees que queda mejor construida? ¿En cuál sería más fácil arreglar una fuga de agua sin tumbar media pared? La respuesta es obvia. Y sin embargo, cuando empezamos a programar en React, casi todos caemos en la trampa de la primera opción: escribimos UN componente gigante que hace todo. Un App.jsx de 500 líneas donde está el encabezado, el menú, la lista de productos, el formulario, el pie de página, la lógica del carrito y hasta el manejo de errores, todo revuelto como ropa sucia en un cesto.

Funciona al principio. El problema llega después: quieres cambiar el color de un botón y no sabes en qué línea está. Quieres reutilizar la tarjeta de producto en otra página y descubres que está pegada con la lógica del carrito y no la puedes separar sin romper todo. Quieres que un compañero te ayude y se pierde entre tus 500 líneas. Esa es la diferencia entre un proyecto de juguete y una aplicación real: las aplicaciones reales están hechas de piezas pequeñas.

A esa habilidad de dividir una aplicación en piezas pequeñas, claras y reutilizables se le llama COMPOSICIÓN DE COMPONENTES. Y es, sin exagerar, la habilidad más importante que vas a aprender en React. Más importante que memorizar hooks, más importante que saber CSS bonito. Porque un programador que sabe componer puede construir aplicaciones de cualquier tamaño: desde una página personal hasta un clon de Netflix. Un programador que no sabe componer se atora en cuanto su App.jsx pasa de las 200 líneas.

En esta unidad vas a dominar las tres herramientas de la composición: el render condicional, los componentes anidados y el paso de props entre niveles.

El RENDER CONDICIONAL es el pegamento de la composición. Ya no basta con dibujar componentes: necesitas decidir CUÁNDO se dibuja cada uno. Si los datos todavía no llegan del servidor, muestra un mensaje de "cargando". Si llegaron pero la lista está vacía, muestra "no hay resultados". Si algo falló, muestra el error. Si todo salió bien, muestra los datos. Esa lógica de "muestra esto o muestra lo otro según la situación" es el render condicional, y lo usarás en literalmente cada pantalla de cada aplicación que construyas. Aquí lo verás a fondo, con sus trucos y sus trampas: el famoso operador &&, los ternarios, el early return y el patrón profesional de los cuatro estados (cargando, vacío, error y datos).

Los COMPONENTES ANIDADOS son la división del trabajo. Aprenderás a mirar una pantalla y verla como un árbol de piezas: la App contiene un Header, un Contenido y un Footer; el Contenido contiene una Lista; la Lista contiene Tarjetas. Cada pieza vive en su propio archivo, hace una sola cosa y se puede probar, arreglar y reutilizar por separado. Verás cómo organizar tus carpetas, cómo nombrar tus componentes y cómo decidir cuándo una pieza ya creció lo suficiente para merecer su propio archivo.

El PROPS DRILLING es el costo de conectar piezas lejanas. Cuando un dato vive en el componente abuelo pero lo necesita el componente nieto, tienes que pasarlo de mano en mano: del abuelo al padre, del padre al hijo, del hijo al nieto. Funciona, pero en aplicaciones grandes se vuelve un enredo. Verás exactamente cuándo está bien hacerlo así y cuándo empieza a doler, y conocerás las salidas profesionales: la composición con children y el Context (que dominarás en la Unidad 8).

Las tres clases originales de esta unidad son: Clase 16 (Render condicional avanzado), Clase 17 (Componentes anidados) y Clase 18 (Props drilling).

Al terminar esta unidad vas a poder hacer algo que distingue a los principiantes de los programadores que ya trabajan en esto: mirar cualquier pantalla de cualquier aplicación y descomponerla mentalmente en componentes. "Esa página tiene un encabezado, una barra lateral, una cuadrícula de tarjetas y un pie; cada tarjeta tiene imagen, título, precio y botón". Esa mirada de rayos X es la que te permitirá construir aplicaciones reales sin perderte. Además, escribirás componentes pequeños y legibles, manejarás los cuatro estados de una petición de datos como un profesional y sabrás cuándo pasar props por varios niveles y cuándo buscar una mejor alternativa.

Un aviso honesto antes de empezar: esta unidad tiene menos "magia" y más "oficio" que las anteriores. No hay un hook nuevo y brillante que aprender. Hay algo mejor: criterio. El criterio para dividir, para nombrar, para decidir qué se muestra y cuándo. Ese criterio es lo que separa al que "sabe React" del que "sabe construir con React". Vamos a construirlo juntos, pieza por pieza, como debe ser.

Explicación detallada

Render condicional: decidir qué se dibuja y cuándo

Hasta ahora, tus componentes han sido bastante obedientes: siempre dibujan lo mismo. Les pasas datos y los muestran. Pero las aplicaciones reales no funcionan así. Piensa en la app de tu banco: cuando la abres, primero ves una ruedita de "cargando". Luego, si todo sale bien, ves tu saldo. Si no tienes movimientos, ves "no hay movimientos recientes". Si se cayó el internet, ves un mensaje de error con un botón de "reintentar". Una sola pantalla, cuatro caras distintas según la situación. Eso es el render condicional: tu componente decide qué dibujar dependiendo del estado de los datos.

La idea es simple pero poderosa: en React, el return de tu componente puede cambiar según condiciones, igual que cualquier función de JavaScript puede devolver cosas distintas según un if. La diferencia es que aquí "devolver algo distinto" significa "dibujar algo distinto en pantalla". Y como React vuelve a ejecutar tu componente cada vez que cambia el estado, la pantalla se actualiza sola cuando la situación cambia: pasas de "cargando" a "datos" sin escribir ni una línea de código para manipular el DOM.

Hay varias técnicas para hacer render condicional, de la más corta a la más explícita. Ninguna es "la correcta" en todos los casos: cada una brilla en una situación distinta. Vamos a verlas todas, con sus trampas incluidas, porque aquí es donde los principiantes se tropiezan más.

El operador &&: "si existe, muéstralo"

La técnica más usada en React es el operador && (AND lógico). Funciona así: pones una condición a la izquierda y el JSX a la derecha. Si la condición es verdadera, React dibuja el JSX. Si es falsa, React no dibuja nada.

```
function PanelUsuario({ usuario }) {  
  return (  
    <div>  
      <h1>Mi cuenta</h1>  
      {usuario && <p>Hola, {usuario.nombre}</p>}  
    </div>  
  )  
}
```

Lee esa línea como una frase: "si usuario existe, muestra el párrafo de saludo". Si usuario es null o undefined (por ejemplo, porque los datos todavía no llegan del servidor), la expresión

completa vale null y React no dibuja nada en ese lugar. Ni un error, ni un hueco raro: simplemente nada.

Este patrón es oro puro cuando trabajas con datos que llegan de internet. Imagina que haces un fetch para traer una lista de productos. Mientras la petición viaja, tu variable datos vale null. Si intentaras dibujar `datos.map(...)` con datos en null, la app tronaría con el famoso error "Cannot read properties of null". Pero con el `&&` te proteges:

```
function Catalogo({ datos }) {  
  return (  
    <div>  
      <h1>Catálogo</h1>  
      {datos && <ListaProductos datos={datos} />}  
    </div>  
  )  
}
```

Mientras datos sea null, el componente ListaProductos ni siquiera se crea. Cuando el fetch termina y datos ya trae el arreglo, React vuelve a dibujar, la condición se vuelve verdadera y la lista aparece como por arte de magia. Por eso en la clase se dice que `&&` "muestra solo si existe": es tu guardia de seguridad contra datos que aún no llegan.

El `&&` también sirve para interruptores de la interfaz: mostrar un mensaje solo si hay notificaciones, mostrar un botón solo si el usuario es administrador, mostrar una insignia solo si el producto está en oferta.

```
{notificaciones.length > 0 && <Insignia numero={notificaciones.length} />}  
{esAdmin && <BotonPanel />}  
{producto.enOferta && <EtiquetaOferta />}
```

Fíjate en el primer ejemplo: no escribí `{notificaciones.length && ...}` sino `{notificaciones.length > 0 && ...}`. Esa diferencia de unas cuantas letras te ahorra uno de los bugs más clásicos y confusos de React. Vamos a ver por qué.

La trampa del cero: cuando `&&` te dibuja un "0" fantasma

Aquí viene la trampa que ha hecho llorar a más de un principiante. En JavaScript, el operador `&&` no devuelve true o false: devuelve uno de los dos operandos. Si el primero es falsy, devuelve el primero tal cual. Los valores falsy en JavaScript son: false, 0, "" (texto vacío), null, undefined y NaN.

Ahora, React tiene una regla: ignora false, null y undefined al dibujar (no muestra nada), pero Sí dibuja los números, incluyendo el 0. Entonces mira lo que pasa con este código inocente:

```
function Carrito({ items }) {  
  return (  
    <div>  
      <h2>Tu carrito</h2>  
      {items.length && <ListaItems items={items} />}  
    </div>  
  )  
}
```

La intención era: "si hay productos en el carrito, muestra la lista". Pero si el carrito está vacío, items.length vale 0. La expresión 0 && <ListaItems .../> se evalúa como 0 (porque 0 es falsy, && devuelve el primer operando). Y React, al recibir un 0, lo dibuja como texto: aparece un "0" solitario y misterioso en tu pantalla, flotando donde debería ir la lista. Tu usuario ve un cero fantasma y tú ves tu dignidad de programador esfumarse.

La solución es nunca poner un número "crudo" a la izquierda del &&. Convierte la condición en un booleano de verdad con una comparación:

```
{items.length > 0 && <ListaItems items={items} />}
```

Ahora la izquierda vale true o false, nunca 0. Con false, React no dibuja nada. Problema resuelto. Otra opción es usar la doble negación (!!) para forzar el booleano: {!!items.length && ...}. Pero la comparación explícita (> 0) se lee mejor y deja clara tu intención.

Regla de oro: a la izquierda del && solo van booleanos. Si tu condición puede valer 0, "" o NaN, conviértela primero. Este es uno de esos detalles que separan al código de juguete del código profesional: el profesional ya se quemó con el cero fantasma una vez y no vuelve a caer.

Ojo con un caso primo del cero: el texto vacío. Si haces {nombre && <p>{nombre}</p>} y nombre es "" (texto vacío), la expresión vale "" y React dibuja... nada visible, pero sí deja un nodo de texto vacío en el DOM. No se ve, pero es mugre. Con la regla de oro (comparar explícitamente o asegurar booleano) también lo evitas.

Ternarios: "si esto, aquello; si no, lo otro"

El `&&` sirve cuando solo hay dos opciones: mostrar algo o no mostrar nada. Pero muchas veces necesitas elegir entre DOS cosas visibles: si el usuario está logueado muestra su nombre, si no muestra el botón de entrar. Para eso está el operador ternario, el `if/else` compacto de JavaScript: `condición ? valorSiTrue : valorSiFalse`.

```
function Encabezado({ usuario }) {
  return (
    <header>
      {usuario ? <p>Hola, {usuario.nombre}</p> : <button>Entrar</button>}
    </header>
  )
}
```

Se lee: "¿hay usuario? muestra el saludo; si no, muestra el botón". Es perfecto para interruptores binarios: encendido/apagado, logueado/no logueado, modo claro/modo oscuro, vista de lista/vista de cuadrícula.

El ternario también brilla para cambiar pequeños pedazos dentro del JSX: un texto, una clase CSS, un icono.

```
<button className={activo ? "btn-activo" : "btn-normal"}>
  {activo ? "Desactivar" : "Activar"}
</button>
```

Pero el ternario tiene un lado oscuro: los ternarios anidados. Cuando necesitas tres o más opciones, la tentación es encadenarlos:

```
{estado === "cargando" ? <Cargando />
  : estado === "error" ? <Error />
  : estado === "vacio" ? <Vacio />
  : <Datos />}
```

¿Puedes leer eso sin entrecerrar los ojos? Funciona, pero es horrible de mantener. Cada nueva condición agrega otro nivel de indentación mental. Dentro de seis meses, ni tú mismo vas a entender qué quisiste decir. Y si un compañero tiene que modificarlo, te va a maldecir en silencio.

La salida profesional: saca la lógica del JSX y usa variables intermedias o una función. El JSX debe ser tonto y legible; la inteligencia vive arriba, en el cuerpo del componente.

```
function Panel({ estado, datos, error }) {
  let contenido
  if (estado === "cargando") {
    contenido = <Cargando />
  } else if (estado === "error") {
    contenido = <Error mensaje={error} />
  } else if (estado === "vacio") {
    contenido = <Vacio />
  } else {
    contenido = <Datos datos={datos} />
  }
  return (
    <div className="panel">
      <h2>Resultados</h2>
      {contenido}
    </div>
  )
}
```

Mira qué diferencia: el return quedó limpio y la lógica se lee de arriba a abajo como una receta. La variable contenido es como un plato donde vas poniendo lo que corresponda según el caso, y al final lo sirves en el JSX. Esta técnica (variable intermedia) es la favorita de los equipos profesionales para casos con más de dos opciones.

Otra variante elegante es usar un objeto como mapa de estados, que verás mucho en código real:

```
const vistas = {
  cargando: <Cargando />,
  error: <Error mensaje={error} />,
  vacio: <Vacio />,
  listo: <Datos datos={datos} />
}
return <div className="panel">{vistas[estado]}</div>
```

Cada técnica tiene su lugar: && para "muestra o no muestras", ternario simple para "esto o lo otro", variable intermedia u objeto mapa para tres o más casos. Aprender a elegir la técnica correcta para cada situación es gran parte del oficio.

Early return: sal temprano, evita la pirámide

Hay una técnica más, favorita de los programadores experimentados: el early return (retorno temprano). La idea: si el componente no puede hacer su trabajo todavía, sal de la función de inmediato devolviendo algo simple, antes de llegar a la lógica principal.

```
function PerfilUsuario({ usuario }) {  
  if (!usuario) {  
    return <p>Cargando perfil...</p>  
  }  
  return (  
    <div className="perfil">  
      <img src={usuario.foto} alt={usuario.nombre} />  
      <h2>{usuario.nombre}</h2>  
      <p>{usuario.bio}</p>  
    </div>  
  )  
}
```

Fíjate en lo que logramos: después del if, JavaScript (y tú) ya SABEN que usuario existe. Todo el JSX de abajo puede usar usuario.nombre, usuario.foto y usuario.bio sin miedo, sin &&, sin signos de interrogación, sin protecciones. El código principal quedó limpio porque el caso problemático se atendió y se despachó al inicio.

Compara con la versión sin early return, donde tendrías que proteger cada línea:

```
function PerfilUsuario({ usuario }) {  
  return (  
    <div className="perfil">  
      {usuario && <img src={usuario.foto} alt={usuario.nombre} />}  
      {usuario && <h2>{usuario.nombre}</h2>}  
      {usuario && <p>{usuario.bio}</p>}  
    </div>  
  )  
}
```

Tres protecciones repetidas para un solo problema. Eso es ruido. El early return dice: "atiende lo urgente primero y quítatelo de encima". Es como el portero de un antro: si no traes identificación, no entras; los que sí la traen pasan directo a divertirse sin que nadie los esté revisando a cada rato.

Puedes encadenar varios early returns para los distintos estados, del más urgente al menos urgente:

```
function PanelDatos({ estado, datos, error }) {
  if (estado === "cargando") {
    return <Cargando />
  }
  if (estado === "error") {
    return <Error mensaje={error} />
  }
  if (!datos || datos.length === 0) {
    return <Vacio />
  }
  return <ListaDatos datos={datos} />
}
```

Cada if es una puerta de salida. Si ninguna se abre, llegas al final con la certeza de que los datos existen y no están vacíos. Este patrón es tan común que tiene nombre propio cuando se usa para validaciones: GUARD CLAUSES (cláusulas de guardia).

Componentes de guardia: el portero educado

Una guard clause es un early return que protege al componente de datos inválidos. Y cuando esa protección se vuelve un patrón repetido, nace el "componente de guardia": un componente pequeño cuya única chamba es decidir si el contenido puede mostrarse o no.

Piensa en una sección de tu app que solo debe verse si el usuario inició sesión. En vez de repetir {usuario && ...} en veinte lugares, creas un guardián:

```
function SoloLogueados({ usuario, children }) {
  if (!usuario) {
    return <p>Inicia sesión para ver este contenido.</p>
  }
  return <>{children}</>
}
```

Y lo usas así:

```
<SoloLogueados usuario={usuario}>
  <PanelSecreto />
  <Configuracion />
</SoloLogueados>
```

El guardián revisa la identificación en la puerta; lo que está adentro (children) ni se entera de la revisión. Este es un ejemplo hermoso de cómo el render condicional y la composición trabajan

juntos: un componente pequeño con una sola responsabilidad (vigilar la puerta) que protege a otros componentes.

Los componentes de guardia aparecen en todas las apps reales con nombres como `RutaProtegida`, `RequierePermiso`, `SoloAdmin`. Si alguna vez ves `<RutaProtegida>` en un código, ya sabes qué hace: es el portero educado que decide quién pasa.

El patrón de los cuatro estados: cargando, vacío, error, datos

Vamos a juntar todo en el patrón más importante de esta sección, el que usarás cada vez que traigas datos de internet. Toda petición de datos vive en uno de cuatro estados, y tu interfaz debe tener una cara para cada uno:

1. **CARGANDO:** la petición va en camino. Muestra un indicador (ruedita, texto "cargando...", esqueleto de contenido). El usuario necesita saber que algo está pasando; el silencio lo pone nervioso.
2. **ERROR:** algo falló (se cayó el internet, el servidor tronó). Muestra qué pasó y un botón de reintentar. Nunca dejes al usuario mirando una pantalla muerta.
3. **VACÍO:** la petición funcionó, pero no hay datos (búsqueda sin resultados, carrito vacío, lista de tareas terminada). Muestra un mensaje amable, no una pantalla en blanco. La pantalla en blanco es el peor mensaje de error: no dice nada.
4. **DATOS:** todo bien, hay datos. Muestra el contenido real.

Aquí está el patrón completo con `early returns`, que es como lo escriben los profesionales:

```

function ListaProductos() {
  const [estado, setEstado] = useState("cargando")
  const [productos, setProductos] = useState([])
  const [error, setError] = useState(null)

  useEffect(() => {
    fetch("https://api.tienda.com/productos")
      .then(respuesta => respuesta.json())
      .then(datos => {
        setProductos(datos)
        setEstado("listo")
      })
      .catch(err => {
        setError(err.message)
        setEstado("error")
      })
  }, [])

  if (estado === "cargando") {
    return <p>Cargando productos...</p>
  }
  if (estado === "error") {
    return (
      <div>
        <p>Ocurrió un error: {error}</p>
        <button onClick={() => window.location.reload()}>Reintentar</button>
      </div>
    )
  }
  if (productos.length === 0) {
    return <p>No hay productos disponibles por ahora.</p>
  }
  return (
    <ul>
      {productos.map(p => <li key={p.id}>{p.nombre}</li>)}
    </ul>
  )
}

```

Recorre el código con calma. El estado inicial es "cargando", así que la primera vez se muestra el mensaje de carga. Cuando el fetch termina, el estado cambia a "listo" (o a "error" si falló) y React vuelve a dibujar. Si llegó vacío el arreglo, se muestra el mensaje amable. Solo al final, cuando hay datos de verdad, se dibuja la lista. Cuatro estados, cuatro caras, cero pantallas en blanco, cero errores por datos nulos.

El error más común de los principiantes es olvidar el caso VACÍO. Piensan en cargando, piensan en error, piensan en datos... y cuando la búsqueda no encuentra nada, la pantalla queda en blanco y el usuario cree que la app se trabó. Grábatelo: toda lista necesita su mensaje de "no hay nada aquí". Es un detalle pequeño que hace ver tu app cien veces más profesional.

Otro detalle fino: fíjate que el estado "vacío" no necesita su propio valor en la variable estado. Se detecta con `productos.length === 0` cuando ya terminó la carga. Eso es porque "vacío" no es algo que el servidor te avise: es algo que deduces al ver los datos. El patrón queda en tres valores de estado ("cargando", "error", "listo") más una revisión de longitud. Limpio y suficiente.

Componentes anidados: dividir para reinar

Si el render condicional es el pegamento, los componentes anidados son los bloques. La idea central de React (la viste en la Unidad 1) es que una interfaz se construye con piezas pequeñas y reutilizables. Pues bien: anidar componentes es exactamente eso llevado a la práctica. Un componente puede dibujar otros componentes dentro de su JSX, y esos a su vez pueden dibujar otros más, formando un árbol.

Mira esta App:

```
function App() {  
  return (  
    <div>  
      <Header />  
      <Contenido />  
      <Footer />  
    </div>  
  )  
}
```

Tres líneas dentro del div, tres componentes. App ya no sabe cómo se ve el encabezado ni el pie de página: solo sabe que existen y en qué orden van. Cada uno de esos componentes puede a su vez contener más componentes:

```
function Contenido() {
  return (
    <main>
      <BarraBusqueda />
      <ListaProductos />
    </main>
  )
}
```

Y ListaProductos dibuja muchas TarjetaProducto. Y TarjetaProducto dibuja ImagenProducto, PrecioProducto y BotonAgregar. Así se forma el **ÁRBOL DE COMPONENTES**: App es la raíz (el tronco), y de ahí cuelgan ramas cada vez más específicas hasta llegar a las hojas (los componentes más pequeños, como un botón o una etiqueta de precio).

Dibuja mentalmente ese árbol para una tienda en línea:

```
App
|-- Header
|   |-- Logo
|   |-- MenuNavegacion
|   `-- CarritoIcono
|-- Contenido
|   |-- BarraBusqueda
|   `-- ListaProductos
|       |-- TarjetaProducto
|       |   |-- ImagenProducto
|       |   |-- PrecioProducto
|       |   `-- BotonAgregar
|       `-- TarjetaProducto ...
`-- Footer
    |-- EnlacesAyuda
    `-- RedesSociales
```

Esta es la famosa analogía del LEGO aplicada en serio: cada cajita es una pieza con forma definida, y las piezas se conectan entre sí para formar estructuras grandes. Si una pieza sale defectuosa (un bug en TarjetaProducto), la cambias sin desarmar toda la construcción. Si necesitas la misma pieza en otro modelo (reutilizar BotonAgregar en otra página), la tomas de la caja y la encajas. Nadie en su sano juicio construiría un castillo de LEGO con una sola pieza gigante e indivisible. Con código pasa lo mismo.

Dividir por responsabilidad: cada pieza hace una sola cosa

La pregunta del millón: ¿cómo sé dónde cortar? ¿Cómo decido qué merece ser su propio componente y qué puede quedarse dentro de otro? La respuesta es el principio de responsabilidad única: cada componente debe hacer UNA sola cosa, y hacerla bien.

Piensa en un restaurante. No hay un empleado que cocine, sirva, cobre, lave platos y atienda el teléfono. Hay un cocinero (cocina), un mesero (sirve), un cajero (cobra). Cada rol tiene una responsabilidad clara. Si el cajero se enferma, el restaurante sigue funcionando con un reemplazo: no tienes que entrenar a alguien en TODO el restaurante, solo en la caja. Los componentes funcionan igual: TarjetaProducto muestra un producto; no debe saber nada del carrito, ni del usuario, ni de la navegación.

Señales de que un pedazo de JSX merece ser su propio componente:

1. SE REPITE. Si el mismo bloque aparece dos o más veces (o sospechas que aparecerá), es componente. El botón de "agregar al carrito" que sale en cada tarjeta: componente. El avatar del usuario que sale en el header y en los comentarios: componente.
2. TIENE UN NOMBRE CLARO. Si puedes señalar un bloque y decir "esto es la tarjeta de producto", "esto es el encabezado", "esto es el formulario de login", entonces ya tienes el nombre del componente. Si no puedes nombrarlo sin decir "esta cosa que hace varias cosas", todavía no está bien dividido.
3. EL ARCHIVO CRECIÓ DEMASIADO. Regla práctica: si el return de tu componente pasa de 50 a 80 líneas, o si el archivo completo pasa de 150 líneas, es hora de extraer piezas. No es ley, es termómetro: el código largo esconde responsabilidades mezcladas.
4. NECESITA SUS PROPIOS COMENTARIOS DE SECCIÓN. Si dentro de tu JSX escribes comentarios como `/* encabezado */`, `/* lista de productos */`, `/* pie */`, cada comentario está gritando "¡soy un componente, sácame de aquí!".
5. MEZCLA NIVELES DE DETALLE. Si en el mismo return tienes el layout general de la página Y el detalle de cómo se dibuja cada estrellita de calificación, estás mezclando el plano de la ciudad con el plano de una casa. La ciudad (App) no necesita saber cuántas estrellas tiene cada producto; eso es chamba de TarjetaProducto.

Vamos a ver una transformación real. Empiezas con un componente monolítico:

```

function App() {
  return (
    <div>
      <header>
        
        <nav>
          <a href="/">Inicio</a>
          <a href="/catalogo">Catálogo</a>
          <a href="/contacto">Contacto</a>
        </nav>
      </header>
      <main>
        <h1>Productos destacados</h1>
        <div className="tarjeta">
          
          <h3>Audífonos Pro</h3>
          <p>$999</p>
          <button>Agregar</button>
        </div>
        <div className="tarjeta">
          
          <h3>Teclado Mecánico</h3>
          <p>$1,499</p>
          <button>Agregar</button>
        </div>
      </main>
      <footer>
        <p>Tienda 2026. Todos los derechos reservados.</p>
      </footer>
    </div>
  )
}

```

Funciona, pero todo está pegado. Ahora divídelo por responsabilidad:


```
function App() {
  return (
    <div>
      <Header />
      <main>
        <h1>Productos destacados</h1>
        <TarjetaProducto imagen="/prod1.jpg" nombre="Audífonos Pro" precio={999} />
        <TarjetaProducto imagen="/prod2.jpg" nombre="Teclado Mecánico"
precio={1499} />
      </main>
      <Footer />
    </div>
  )
}
```

App quedó de 10 líneas legibles: es el plano general. Header, TarjetaProducto y Footer viven en sus propios archivos, cada uno enfocado en lo suyo. Y fíjate en el regalo extra: como TarjetaProducto recibe props, agregar un tercer producto es una sola línea, no un bloque de 6 líneas copiado y pegado. La composición y las props son socias inseparables: dividir componentes casi siempre viene acompañado de pasarles datos por props.

Un componente por archivo: orden en tu proyecto

En proyectos reales, cada componente vive en su propio archivo, y los archivos se organizan en carpetas. La convención más común: una carpeta components/ dentro de src/, con un archivo por componente nombrado igual que el componente.

```
src/
|-- App.jsx
|-- main.jsx
`-- components/
    |-- Header.jsx
    |-- Footer.jsx
    |-- TarjetaProducto.jsx
    |-- ListaProductos.jsx
    |-- BarraBusqueda.jsx
    `-- BotonAgregar.jsx
```

Cada archivo exporta su componente (export default) y quien lo necesita lo importa:

```
// components/TarjetaProducto.jsx
function TarjetaProducto({ imagen, nombre, precio }) {
  return (
    <div className="tarjeta">
      <img src={imagen} alt={nombre} />
      <h3>{nombre}</h3>
      <p>${precio}</p>
      <BotonAgregar nombre={nombre} />
    </div>
  )
}

export default TarjetaProducto
```

```
// App.jsx
import Header from "../components/Header.jsx"
import TarjetaProducto from "../components/TarjetaProducto.jsx"
import Footer from "../components/Footer.jsx"

function App() {
  return (
    <div>
      <Header />
      <main>
        <TarjetaProducto imagen="/prod1.jpg" nombre="Audífonos Pro" precio={999} />
      </main>
      <Footer />
    </div>
  )
}

export default App
```

Tres reglas de nomenclatura que te harán ver profesional desde el día uno:

1. PascalCase para componentes y archivos: TarjetaProducto, BarraBusqueda. Siempre empiezan con mayúscula (así React los distingue de las etiquetas HTML normales).
2. Nombres descriptivos de lo que SON, no de dónde están: TarjetaProducto (bien), DivIzquierdo (mal), CosaAzul (terrible). Si dentro de seis meses lees el nombre y no sabes qué dibuja, el nombre está mal.
3. La carpeta components/ es para piezas reutilizables. Si un componente solo se usa dentro de otro (por ejemplo, EstrellasCalificacion solo vive dentro de TarjetaProducto), algunos

equipos lo ponen en una subcarpeta o junto a su padre. No hay una sola regla sagrada; lo importante es que el equipo la siga igual.

Cuando el proyecto crece, la carpeta `components/` se organiza por feature (característica): `components/productos/`, `components/usuario/`, `components/comunes/` (para los que se usan en todas partes, como Botón o Cargando). No te obsesiones con la estructura perfecta desde el día uno: empieza simple con una sola carpeta `components/` y divídela cuando empiece a tener más de 15 archivos y te cueste encontrar cosas.

Presentacionales vs contenedores: dos personalidades

Conforme divides componentes, notarás que desarrollan dos personalidades distintas. Entenderlas te ayuda a decidir qué lógica va dónde.

Los COMPONENTES PRESENTACIONALES (también les dicen "tontos", de cariño) solo se dedican a verse bien. Reciben datos por props y los dibujan. No tienen estado propio, no hacen fetch, no deciden nada importante. Son como los actores: se aprenden su guion (las props) y lo interpretan.

```
function TarjetaProducto({ nombre, precio, enOferta }) {
  return (
    <div className="tarjeta">
      <h3>{nombre}</h3>
      <p>${precio}</p>
      {enOferta && <span>OFERTA</span>}
    </div>
  )
}
```

Puro dibujo. Le das nombre, precio y si está en oferta, y te devuelve la tarjeta. No sabe de dónde vienen los datos ni a dónde van. Por eso es facilísimo de reutilizar y de probar: con las mismas props, siempre dibuja lo mismo.

Los COMPONENTES CONTENEDORES (los "listos") se dedican a la lógica. Tienen estado, hacen fetch, manejan eventos, y le pasan todo ya digerido a los presentacionales. Son como el director de la obra: no sale en escena, pero coordina todo.

```
function ListaProductos() {
  const [productos, setProductos] = useState([])
  const [estado, setEstado] = useState("cargando")

  useEffect(() => {
    fetch("https://api.tienda.com/productos")
      .then(r => r.json())
      .then(d => { setProductos(d); setEstado("listo") })
      .catch(() => setEstado("error"))
  }, [])

  if (estado === "cargando") return <p>Cargando...</p>
  if (estado === "error") return <p>Error al cargar.</p>
  return (
    <div className="lista">
      {productos.map(p => (
        <TarjetaProducto key={p.id} nombre={p.nombre} precio={p.precio}
        enOferta={p.oferta} />
      ))}
    </div>
  )
}
```

ListaProductos (contenedor) hace el trabajo sucio: fetch, estados, errores. TarjetaProducto (presentacional) solo dibuja. Esta división es poderosa por tres razones: los presentacionales se reutilizan en cualquier parte (la tarjeta sirve en el catálogo, en el carrito, en favoritos), los contenedores se entienden sin distracciones visuales, y cuando algo se ve mal sabes que el bug está en el presentacional, mientras que si los datos están mal sabes qué está en el contenedor.

Ojo: esto es una IDEA GENERAL, no una ley. En la práctica, muchos componentes mezclan un poco de ambas personalidades, y está bien. La utilidad está en preguntarte: "¿este componente está haciendo demasiadas cosas? ¿puedo separar lo que DIBUJA de lo que DECIDE?". Si la respuesta es sí, ya sabes cómo dividirlo.

La analogía del organigrama

Si el LEGO explica las piezas, el organigrama de una empresa explica el árbol. En una empresa, el director general no hace el trabajo de todos: delega en gerentes, que delegan en supervisores, que coordinan a los empleados. Cada nivel sabe lo necesario de su nivel, ni más ni menos. El director no necesita saber cómo archiva cada empleado sus papeles; el empleado no necesita saber la estrategia anual de la empresa.

Tu árbol de componentes es igual: App (el director) decide la estructura general. Los componentes intermedios (gerentes) coordinan secciones. Los componentes hoja (empleados) hacen el trabajo concreto de dibujar. Los datos fluyen de arriba hacia abajo por props, como las instrucciones fluyen del director a los empleados. Y cuando algo falla, sabes en qué nivel buscar: ¿se ve mal? Revisa al empleado (presentacional). ¿los datos no llegan? Revisa al gerente (contenedor).

Props drilling: cuando los datos viajan de mano en mano

Ya sabes que los datos en React fluyen en una sola dirección: de padres a hijos, a través de props. Eso funciona de maravilla cuando el padre dibuja directamente al hijo que necesita el dato. Pero ¿qué pasa cuando el dato vive en el ABUELO y lo necesita el NIETO, con el padre en medio que ni lo usa? Tienes que pasarlo de mano en mano: el abuelo se lo da al padre, el padre se lo pasa al hijo, el hijo se lo pasa al nieto. A ese viaje con escalas se le llama PROPS DRILLING (taladrar con props: vas perforando niveles hasta llegar al que necesita el dato).

Mira el ejemplo clásico de tres niveles. La App tiene el tema visual ("claro" u "oscuro") y el Botón, tres niveles abajo, lo necesita para pintarse del color correcto:

```

function App() {
  const [tema, setTema] = useState("claro")
  return (
    <div>
      <BarraHerramientas tema={tema} />
    </div>
  )
}

function BarraHerramientas({ tema }) {
  return (
    <div className="barra">
      <GrupoBotones tema={tema} />
    </div>
  )
}

function GrupoBotones({ tema }) {
  return (
    <div className="grupo">
      <Boton tema={tema} texto="Guardar" />
      <Boton tema={tema} texto="Cancelar" />
    </div>
  )
}

function Boton({ tema, texto }) {
  return (
    <button className={tema === "oscuro" ? "btn-oscuro" : "btn-claro"}>
      {texto}
    </button>
  )
}

```

Sigue el viaje de la prop `tema`: nace en `App`, viaja a `BarraHerramientas` (que no la usa, solo la pasa), luego a `GrupoBotones` (que tampoco la usa, solo la pasa), y por fin llega a `Botón`, que es el único que de verdad la necesita. `BarraHerramientas` y `GrupoBotones` son "repartidores": firman el paquete y lo pasan al siguiente sin abrirlo.

Con tres niveles, esto se tolera bien. Es explícito (ves exactamente por dónde viaja el dato), es fácil de rastrear y no necesitas ninguna herramienta extra. Para casos así, el props drilling no es un problema: es simplemente cómo funciona React.

Cuándo el drilling empieza a doler

El dolor llega cuando los niveles se multiplican. Imagina una app real donde el dato del usuario logueado lo necesitan componentes a 5 o 6 niveles de profundidad: el avatar en el header, el saludo en el dashboard, los permisos en un formulario, el nombre en los comentarios. El objeto usuario viaja por una docena de componentes intermedios que ni lo tocan. Y entonces pasan tres cosas feas:

1. **REFACTOR DOLOROSO.** Quieres cambiar el nombre de la prop de usuario a sesión, o agregar un dato nuevo que necesita el nieto. Tienes que editar **TODOS** los niveles intermedios, uno por uno, aunque no les importe el dato. Un cambio pequeño se convierte en una cacería por 8 archivos.
2. **COMPONENTES CONTAMINADOS.** Esos componentes intermedios terminan recibiendo props que no usan, solo para pasarlas. Su firma se llena de parámetros ajenos: `function Panel({ título, usuario, tema, idioma, permisos, ... })`. Cuando lees ese componente, no sabes qué props son tuyas y cuáles solo van de paso. Es como un mensajero que carga 10 paquetes y solo uno es para él.
3. **RIGIDEZ.** Si mañana quieres mover el componente nieto a otra rama del árbol (reutilizarlo en otra página), tienes que reconstruir toda la cadena de props en la nueva ubicación. El componente no es independiente: está atado a la ruta exacta por donde le llegan sus datos.

La analogía de las muñecas rusas (matrioshkas) ayuda a visualizarlo: cada muñeca contiene a la siguiente, y para llegar a la más pequeña tienes que abrir todas las de afuera. Si solo quieres ver la muñeca pequeña, abrir 6 muñecas es un ritual cansado. Y si alguien cambia el orden de las muñecas intermedias, tu ritual se rompe.

Entonces, ¿cuándo está BIEN el props drilling y cuándo hay que buscar alternativas? Reglas prácticas:

- 1 a 2 niveles de intermediarios: perfectamente bien. Es lo normal en React.
- Los intermediarios son estables (no los reordenas ni los reutilizas en otro lado): bien.
- El dato cambia poco (tema, idioma, usuario): el costo es bajo.
- 3 o más niveles de intermediarios que solo pasan el dato: empieza a oler mal.
- El dato lo necesitan MUCHOS componentes en distintas ramas del árbol (usuario, tema, idioma, carrito): el drilling se vuelve insostenible. Aquí necesitas otra estrategia.

Alternativas: composición con children y Context (adelanto)

Antes de que te angusties, hay salidas. La primera la ya conoces a medias: la COMPOSICIÓN CON CHILDREN. Muchas veces el intermediario no necesita recibir el dato para pasarlo: puede simplemente dibujar a sus hijos tal como vienen, y el padre le pasa los hijos YA CONFIGURADOS.

Mira la diferencia. Versión con drilling (el intermediario pasa la prop):

```
function App() {
  const tema = "oscuro"
  return <Panel tema={tema} />
}
function Panel({ tema }) {
  return (
    <div className="panel">
      <Boton tema={tema} texto="Aceptar" />
    </div>
  )
}
```

Versión con composición (el intermediario solo dibuja children):

```
function App() {
  const tema = "oscuro"
  return (
    <Panel>
      <Boton tema={tema} texto="Aceptar" />
    </Panel>
  )
}
function Panel({ children }) {
  return <div className="panel">{children}</div>
}
```

¿Ves el truco? Panel ya no sabe nada del tema. Su única chamba es dibujar el contenedor con lo que le pongan dentro. App conecta directamente a Botón con su prop, saltándose al intermediario. El drilling desapareció porque el intermediario dejó de ser repartidor y se volvió un marco: no le importa qué hay dentro del cuadro, solo lo enmarca bonito.

Esta técnica resuelve una cantidad sorprendente de casos de drilling. Cada vez que un componente intermedio solo "pasa" props sin usarlas, pregúntate: "¿puedo pasarle children ya armados en vez de props sueltas?". Si la respuesta es sí, el código queda más limpio y los componentes más independientes.

La segunda salida es el CONTEXT, y es la artillería pesada. Context es una herramienta de React para compartir datos con TODO el árbol sin pasarlos de mano en mano: el abuelo pone el dato en un "contexto" y cualquier nieto, bisnieto o tataranieto lo toma directamente, sin intermediarios. Es cómo instalar un altavoz en la empresa: en vez de que el mensaje pase de jefe en jefe hasta el empleado, el director lo anuncia y todos los que necesitan oírlo lo escuchan directo.

```
UsuarioContexto.Provider value={usuario} // el abuelo lo pone
const usuario = useContext(UsuarioContexto) // el nieto lo toma directo
```

No vamos a profundizar aquí porque Context tiene sus propias reglas y sus propios peligros (usarlo mal puede volver tu app difícil de rastrear), y se merece su propio espacio: lo dominarás en la UNIDAD 8. Por ahora quédate con el mapa completo: para 1-2 niveles usa props normales (drilling leve, sin culpa); si los intermediarios solo pasan datos, prueba composición con children; y si el dato lo necesita media aplicación a gran profundidad, Context te espera en la Unidad 8.

Un último consejo de oficio: no optimices antes de tiempo. Muchos principiantes, en cuanto escuchan que el props drilling "es malo", quieren meter Context en todas partes desde el día uno. Error. El drilling de 2 niveles es simple, explícito y fácil de entender. Context agrega indirección: el dato aparece "por arte de magia" y cuesta más rastrear de dónde viene. La regla profesional es: empieza con props, y solo cuando el drilling duela DE VERDAD (muchos niveles, muchos consumidores), migra a Context. El código simple que funciona le gana al código elegante que nadie entiende.

Errores comunes (y cómo evitarlos)

1. EL COMPONENTE DE 500 LÍNEAS. El clásico. Todo empezó como "solo agregé este botoncito aquí" y seis meses después el archivo es un monstruo que nadie se atreve a tocar. Antídoto: cuando un componente pase de ~150 líneas o su return pase de ~80, extrae piezas. Hazlo en cuanto lo notes, no "después". El "después" nunca llega.

2. PROP DRILLING DE 5 NIVELES. Pasar usuario por App -> Layout -> Dashboard -> Panel -> Tarjeta -> Avatar, con 4 intermediarios que ni lo usan. Antídoto: a los 3 niveles de intermediarios, detente y pregunta si composición con children lo resuelve. Si el dato lo necesitan muchos, planea Context (Unidad 8).
3. CONDICIONALES ILEGIBLES. Ternarios anidados de 4 niveles dentro del JSX, && con lógica compleja a la izquierda, condiciones que mezclan 5 variables. Antídoto: saca la lógica a variables o funciones ANTES del return. El JSX es para dibujar, no para pensar.
4. OLVIDAR EL CASO "VACÍO". La búsqueda que no encuentra nada deja la pantalla en blanco. El carrito vacío no dice nada. La lista de notificaciones vacía es un hueco mudo. Antídoto: cada lista que dibujes necesita su mensaje amable de "no hay nada aquí". Hazlo un hábito, como lavarte las manos.
5. LÓGICA COMPLEJA DENTRO DEL MAP. El .map() que dibuja la lista termina con filtros, ordenamientos y transformaciones adentro. Antídoto: prepara los datos antes (filtra y ordena en variables), y deja el map solo para dibujar. El map es un pincel, no una licuadora.
6. DIVIDIR DEMASIADO PRONTO. El extremo opuesto: crear un componente para cada 3 líneas, terminando con 40 archivos donde era suficiente con 6. Componentes como EnvoltorioDiv o TextoNegrita no aportan nada. Antídoto: un componente debe ganarse su existencia (se repite, tiene nombre claro, tiene lógica propia). Si no, es solo ruido.
7. NOMBRES QUE NO DICEN NADA. Componente1, Cosa, Wrapper, MiDiv, Parte2. Dentro de un mes no sabrás qué hace cada uno. Antídoto: nombra por lo que ES (TarjetaProducto, FormularioLogin). Si te cuesta nombrarlo, probablemente está mal dividido.

Buenas prácticas de composición

- UN COMPONENTE, UNA RESPONSABILIDAD. Antes de escribirlo, completa la frase: "este componente se encarga de ____". Si necesitas la palabra "y" más de una vez, divídelo.
- NOMBRA COMO UN PROFESIONAL: PascalCase, descriptivo, sin abreviaturas raras. TarjetaProducto, no TarjProd.
- UN COMPONENTE POR ARCHIVO, en components/, importado donde se use. Tu yo del futuro te lo agradecerá cuando busque dónde está el bug.
- EL JSX TONTO, LA LÓGICA ARRIBA. Condicionales enredados van en variables o funciones antes del return, no incrustados en el marcado.

- CUBRE LOS CUATRO ESTADOS de toda petición: cargando, error, vacío y datos. Sin excepciones.
- BOOLEANOS A LA IZQUIERDA DEL &&. Nunca pongas un número o texto crudo: compara explícitamente (`length > 0`).
- EMPIEZA CON PROPS, MIGRA A CONTEXT SOLO SI DUELE. Simple y explícito le gana a elegante y misterioso.
- EXTRAE CUANDO SE REPITE. La segunda vez que copies y pegues un bloque de JSX, es un componente esperando nacer.

Ejemplos paso a paso

Ejemplo 1: Panel que muestra cargando, datos o error según el estado

Este es el patrón de los cuatro estados aplicado a un panel de noticias. Simula una petición a internet con un temporizador para que puedas probarlo sin necesidad de un servidor real. Crea el archivo `src/components/PanelNoticias.jsx`:

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

function PanelNoticias() {
  const [estado, setEstado] = useState("cargando")
  const [noticias, setNoticias] = useState([])

  useEffect(() => {
    const temporizador = setTimeout(() => {
      const exito = Math.random() > 0.3
      if (exito) {
        setNoticias([
          { id: 1, titulo: "React 19 ya está aquí" },
          { id: 2, titulo: "Guía de hooks en español" }
        ])
        setEstado("listo")
      } else {
        setEstado("error")
      }
    }, 1500)
    return () => clearTimeout(temporizador)
  }, [])

  if (estado === "cargando") {
    return <p>Cargando noticias...</p>
  }
}
```

```

}

if (estado === "error") {
  return (
    <div>
      <p>No se pudieron cargar las noticias.</p>
      <button onClick={() => window.location.reload()}>
        Reintentar
      </button>
    </div>
  )
}

if (noticias.length === 0) {
  return <p>No hay noticias por ahora. Vuelve más tarde.</p>
}

return (
  <ul>
    {noticias.map(n => (
      <li key={n.id}>{n.titulo}</li>
    ))}
  </ul>
)
}

export default PanelNoticias

```

Línea por línea. La primera línea importa los hooks que necesitamos: `useState` para guardar el estado de la petición y la lista de noticias, `useEffect` para simular la petición cuando el componente aparece en pantalla.

Dentro del componente, creamos dos estados: `estado` (que empieza en "cargando") y `noticias` (que empieza como arreglo vacío). Fíjate que el estado inicial ya cuenta una historia: "todavía no sé qué pasó con la petición, así que asumo que va en camino".

El `useEffect` se ejecuta una sola vez (arreglo de dependencias vacío) cuando el componente se monta. Dentro, `setTimeout` simula que la petición tarda 1.5 segundos. Al terminar, `Math.random() > 0.3` decide al azar: 70% de las veces la petición "tiene éxito" y guardamos dos noticias de ejemplo cambiando el estado a "listo"; 30% de las veces "falla" y el estado pasa a "error". El `return () => clearTimeout(...)` es limpieza: si el componente desaparece antes de los 1.5 segundos, cancelamos el temporizador para no intentar actualizar un componente que ya no existe.

Ahora los early returns, el corazón del ejemplo. Si estado es "cargando", devolvemos el mensaje de carga y salimos: React dibuja eso y nada más. Si es "error", devolvemos el mensaje de error con un botón que recarga la página (en una app real, reintentarías la petición en vez de recargar). Si llegamos al tercer if, sabemos que la petición terminó bien, pero revisamos si el arreglo quedó vacío: de ser así, mostramos el mensaje amable de "no hay nada". Solo si pasamos las tres puertas llegamos al return final, donde dibujamos la lista con map. Cada noticia necesita su key única (usamos n.id).

Para probarlo, importa el componente en tu App.jsx y úsalo: `<PanelNoticias />`. Recarga la página varias veces: unas veces verás las noticias, otras el error. En todos los casos, durante 1.5 segundos verás el "cargando". Para ver el caso vacío, cambia temporalmente el arreglo de noticias a `[]` en la rama de éxito. Este ejemplo es una plantilla: cambia "noticias" por "productos", "usuarios" o lo que sea, y tienes el patrón listo para cualquier pantalla de tu app.

Ejemplo 2: App dividida en Header, Contenido y Footer

Aquí aplicamos la división por responsabilidad con un componente por archivo. Vamos a crear cuatro archivos. Primero, `src/components/Header.jsx`:

```
function Header({ titulo, usuario }) {
  return (
    <header>
      <h1>{titulo}</h1>
      {usuario
        ? <p>Hola, {usuario.nombre}</p>
        : <button>Iniciar sesión</button>}
    </header>
  )
}

export default Header
```

Header recibe dos props: el título del sitio y el usuario (que puede ser null). Usa un ternario para el render condicional: si hay usuario, saluda con su nombre; si no, muestra el botón de entrar. Fíjate que Header no sabe de dónde viene el usuario ni cómo se inició sesión: solo dibuja según lo que recibe. Eso lo hace reutilizable.

Segundo, `src/components/Contenido.jsx`:

```
import TarjetaProducto from "../TarjetaProducto.jsx"

function Contenido({ productos }) {
  if (productos.length === 0) {
    return <p>No hay productos para mostrar.</p>
  }
  return (
    <main>
      <h2>Productos destacados</h2>
      {productos.map(p => (
        <TarjetaProducto
          key={p.id}
          nombre={p.nombre}
          precio={p.precio}
        />
      ))}
    </main>
  )
}

export default Contenido
```

Contenido importa a TarjetaProducto (componente anidado un nivel más abajo). Recibe el arreglo de productos y lo primero que hace es el early return del caso vacío: si no hay productos, muestra el mensaje amable. Si hay, dibuja el título de la sección y mapea cada producto a una TarjetaProducto, pasándole nombre y precio por props. Observa que Contenido no sabe cómo se ve una tarjeta: delega ese detalle. Esa es la división de responsabilidades en acción.

Tercero, src/components/TarjetaProducto.jsx:

```
function TarjetaProducto({ nombre, precio }) {
  return (
    <div className="tarjeta">
      <h3>{nombre}</h3>
      <p>${precio}</p>
      <button>Agregar al carrito</button>
    </div>
  )
}

export default TarjetaProducto
```

El componente hoja: puro presentacional. Recibe nombre y precio, dibuja la tarjeta. No tiene estado, no decide nada. Si mañana quieres cambiar el diseño de las tarjetas, solo tocas este archivo.

Cuarto, src/components/Footer.jsx:

```
function Footer() {
  const anio = new Date().getFullYear()
  return (
    <footer>
      <p>Mi Tienda {anio}. Todos los derechos reservados.</p>
    </footer>
  )
}

export default Footer
```

Footer ni siquiera necesita props: calcula el año actual y dibuja el pie. Finalmente, el director de orquesta, src/App.jsx:

```
import Header from "../components/Header.jsx"
import Contenido from "../components/Contenido.jsx"
import Footer from "../components/Footer.jsx"

function App() {
  const usuario = { nombre: "María" }
  const productos = [
    { id: 1, nombre: "Audífonos Pro", precio: 999 },
    { id: 2, nombre: "Teclado Mecánico", precio: 1499 }
  ]
  return (
    <div>
      <Header titulo="Mi Tienda" usuario={usuario} />
      <Contenido productos={productos} />
      <Footer />
    </div>
  )
}

export default App
```

App define los datos (usuario y productos) y los reparte por props a quien los necesita. Su return es el plano general de la página: encabezado, contenido, pie. Cuenta las líneas: el JSX

de App cabe en 7 líneas y se entiende de un vistazo. Cada pieza vive en su archivo, con su responsabilidad clara. Si quieres agregar una barra lateral mañana, creas BarraLateral.jsx y agregas una línea en App. Así se construyen las apps reales: sumando piezas, no alargando un archivo.

Prueba el caso vacío cambiando productos a []: verás el mensaje amable gracias al early return de Contenido. Prueba usuario en null: verás el botón de entrar gracias al ternario de Header. Cada pieza maneja su propio render condicional sin que App se entere de los detalles.

Ejemplo 3: Tarjeta de usuario con props en 3 niveles (drilling en acción)

Este ejemplo muestra el props drilling funcionando en un caso donde está perfectamente bien usarlo: pocos niveles y componentes estables. La App tiene los datos del usuario; el componente Avatar, tres niveles abajo, necesita la foto. Todo en un solo archivo para que veas el viaje completo de la prop:


```

function App() {
  const usuario = {
    nombre: "Carlos Rivera",
    foto: "/fotos/carlos.jpg",
    rol: "Administrador"
  }
  return (
    <div>
      <h1>Equipo</h1>
      <TarjetaUsuario usuario={usuario} />
    </div>
  )
}

function TarjetaUsuario({ usuario }) {
  return (
    <div className="tarjeta-usuario">
      <EncabezadoTarjeta usuario={usuario} />
      <p>Rol: {usuario.rol}</p>
    </div>
  )
}

function EncabezadoTarjeta({ usuario }) {
  return (
    <div className="encabezado">
      <Avatar usuario={usuario} />
      <h3>{usuario.nombre}</h3>
    </div>
  )
}

function Avatar({ usuario }) {
  return (
    <img
      src={usuario.foto}
      alt={"Foto de " + usuario.nombre}
      className="avatar"
    />
  )
}

export default App

```

Sigue el viaje de la prop usuario paso a paso. Nace en App como un objeto con nombre, foto y rol. App se la pasa a TarjetaUsuario. TarjetaUsuario la usa un poco (muestra el rol) pero también se la pasa a EncabezadoTarjeta sin modificarla: aquí TarjetaUsuario actúa como intermediaria. EncabezadoTarjeta usa el nombre para el h3 y pasa el objeto completo a Avatar: otra intermediaria. Finalmente Avatar, el único que necesita la foto, la toma de usuario.foto y dibuja la imagen.

Cuenta los niveles: App -> TarjetaUsuario -> EncabezadoTarjeta -> Avatar. El dato viaja 3 saltos y hay 2 intermediarios. Según nuestras reglas prácticas, esto está en el límite de lo aceptable: funciona, es explícito y fácil de rastrear. Si agregaras un nivel más (por ejemplo, que Avatar dibujara un MarcoAvatar que dibujara la imagen), sería momento de refactorizar.

Observa también un detalle de diseño: pasamos el objeto usuario COMPLETO en cada nivel, no prop por prop (no hicimos nombre={usuario.nombre} foto={usuario.foto}). Cuando varios datos viajan juntos y pertenecen a la misma "cosa", pasar el objeto completo es más limpio y evita que la firma del componente se llene de parámetros. La excepción: si el componente hijo solo necesita UN dato del objeto, pásale solo ese dato para mantenerlo independiente.

Para practicar, intenta la refactorización con children: modifica EncabezadoTarjeta para que reciba children en vez de usuario, y haz que TarjetaUsuario le pase <Avatar usuario={usuario} /> ya armado. Verás cómo desaparece un nivel de drilling y EncabezadoTarjeta queda más limpio. Ese es el poder de la composición.

Ejemplo 4: Lista de tareas con mensaje amable cuando está vacía

El caso "vacío" merece su propio ejemplo porque es el más olvidado y el que más profesionalismo aporta. Vamos a construir una mini lista de tareas donde el usuario puede agregar y eliminar tareas, y donde el estado vacío tiene su propio diseño amable. Archivo src/components/ListaTareas.jsx:

```
import { useState } from "react"

function ListaTareas() {
  const [tareas, setTareas] = useState([])
  const [texto, setTexto] = useState("")

  function agregarTarea() {
    if (texto.trim() === "") {
      return
    }
    const nueva = { id: Date.now(), texto: texto.trim() }
    setTareas([...tareas, nueva])
  }
}
```

```

    setTexto("")
  }

  function eliminarTarea(id) {
    setTareas(tareas.filter(t => t.id !== id))
  }

  return (
    <div>
      <h2>Mis tareas</h2>
      <input
        value={texto}
        onChange={e => setTexto(e.target.value)}
        placeholder="Escribe una tarea"
      />
      <button onClick={agregarTarea}>Agregar</button>

      {tareas.length === 0 ? (
        <div className="vacio">
          <p>No tienes tareas pendientes.</p>
          <p>Agrega tu primera tarea para empezar.</p>
        </div>
      ) : (
        <ul>
          {tareas.map(t => (
            <li key={t.id}>
              {t.texto}
              <button onClick={() => eliminarTarea(t.id)}>
                Eliminar
              </button>
            </li>
          ))}
        </ul>
      )}
    </div>
  )
}

export default ListaTareas

```

Vamos por partes. Los dos estados: tareas (arreglo que empieza vacío) y texto (lo que el usuario escribe en el input). La función agregarTarea primero valida con un early return: si el texto está vacío (después de quitar espacios con trim), no hace nada. Luego crea la tarea con un id único basado en la hora (Date.now) y la agrega creando un arreglo nuevo con el spread [...tareas, nueva]: nunca mutamos el estado directamente, creamos una copia. Finalmente limpiamos el input.

La función eliminarTarea usa filter para crear un arreglo nuevo sin la tarea del id indicado. Otra vez: sin mutar, creando un arreglo fresco. React detecta el arreglo nuevo y vuelve a dibujar.

El render condicional es el ternario grande de abajo: ¿tareas.length === 0? Si sí, dibujamos el bloque "vacío" con DOS párrafos amables: uno informa, el otro invita a la acción ("agrega tu primera tarea"). Ese segundo párrafo es el toque profesional: no solo dices que no hay nada, le dices al usuario qué hacer. Si no, dibujamos la lista con map, cada tarea con su botón de eliminar.

Fíjate en los paréntesis alrededor de cada rama del ternario: cuando el JSX de una rama ocupa varias líneas, los paréntesis lo hacen legible y evitan errores. Y nota que el formulario (input + botón) está FUERA del ternario: siempre se muestra, haya tareas o no. Solo la zona de la lista cambia según el caso. Esa es una decisión de diseño típica: lo que no depende de la condición va fuera, lo que sí depende va dentro.

Prueba este componente: agrégalo a tu App con <ListaTareas />. Al inicio verás el mensaje amable. Agrega dos tareas: aparece la lista. Elimínalas una por una: cuando eliminas la última, el mensaje amable regresa solo. Esa transición automática (lista -> vacío -> lista) es el render condicional trabajando con el estado: tú solo describes cada caso, React se encarga de cambiar de uno a otro.

Práctica

Ejercicio 1: Semáforo de estados de una petición

Crea un componente SemaforoPeticon que simule cargar una lista de películas. Debe mostrar "Buscando películas..." mientras carga (2 segundos), un mensaje de error con botón de reintentar si falla (usa Math.random para simular el falló como en el Ejemplo 1), "No se encontraron películas" si la lista llega vacía, y la lista de títulos si hay datos. Usa early returns.

SOLUCIÓN: El truco está en ordenar los early returns del caso más urgente al menos urgente: primero cargando, luego error, luego vacío, y al final los datos. El estado inicial debe ser "cargando" para que la primera cara que vea el usuario sea la correcta. El

botón de reintentar puede recargar la página con `window.location.reload()`, que es lo más simple para este ejercicio; en una app real llamarías de nuevo a la función que hace la petición.

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

function SemaforoPeticion() {
  const [estado, setEstado] = useState("cargando")
  const [peliculas, setPeliculas] = useState([])

  useEffect(() => {
    const t = setTimeout(() => {
      if (Math.random() > 0.4) {
        setPeliculas([
          { id: 1, titulo: "Volver al futuro" },
          { id: 2, titulo: "El laberinto del fauno" }
        ])
        setEstado("listo")
      } else {
        setEstado("error")
      }
    }, 2000)
    return () => clearTimeout(t)
  }, [])

  if (estado === "cargando") {
    return <p>Buscando películas...</p>
  }
  if (estado === "error") {
    return (
      <div>
        <p>Falló la conexión. Inténtalo de nuevo.</p>
        <button onClick={() => window.location.reload()}>
          Reintentar
        </button>
      </div>
    )
  }
  if (peliculas.length === 0) {
    return <p>No se encontraron películas.</p>
  }
  return (
    <ul>
      {peliculas.map(p => <li key={p.id}>{p.titulo}</li>)}
    </ul>
  )
}
```

```

    )
  }

export default SemaforoPeticion

```

Para probar el caso vacío, cambia temporalmente el arreglo a [] en la rama de éxito. Observa que el caso vacío se detecta con `peliculas.length === 0` DESPUÉS de confirmar que ya no estamos cargando ni en error: el orden de los `if` importa. Si pusieras la revisión de vacío antes que la de "cargando", al inicio (arreglo vacío + estado cargando) mostrarías "no se encontraron" por error durante 2 segundos. El orden correcto evita ese bug.

Ejercicio 2: Layout reutilizable con Header, Main y Footer

Crea un componente `Layout` que reciba título por props y children, y dibuje un Header con el título, un Main con los children, y un Footer fijo con el texto "Lucas EGA Academy". Luego crea una página `PaginalInicio` que use `Layout` para mostrar un mensaje de bienvenida, y verifica que el Footer se reutiliza sin repetir código.

SOLUCIÓN: La clave es que `Layout` no sepa nada del contenido: solo provee el marco (header, main, footer) y dibuja lo que le pasen por children en el Main. Así puedes crear 20 páginas distintas sin repetir el encabezado ni el pie.

```

// components/Layout.jsx
function Layout({ titulo, children }) {
  return (
    <div>
      <header>
        <h1>{titulo}</h1>
      </header>
      <main>{children}</main>
      <footer>
        <p>Lucas EGA Academy</p>
      </footer>
    </div>
  )
}

export default Layout

```

```
// App.jsx
import Layout from "../components/Layout.jsx"

function PaginaInicio() {
  return (
    <Layout titulo="Bienvenido">
      <p>Esta es la página de inicio.</p>
      <p>El header y el footer vienen del Layout.</p>
    </Layout>
  )
}

export default PaginaInicio
```

Fíjate en cómo se lee `PaginaInicio`: envuelves tu contenido en `<Layout>` y le pasas el título. Todo lo que está entre las etiquetas de apertura y cierre viaja por la prop `children` y aparece dentro del `<main>`. Si mañana creas `PaginaContacto`, reutilizas `Layout` con otro título y otro contenido, y el footer sale gratis. Este es el patrón de layout que usan casi todas las apps React del mundo: un marco compartido y páginas que solo definen su contenido. Además, nota que este ejercicio usa composición con `children` para EVITAR props drilling: el contenido no necesita viajar por props intermedias, va directo al Main.

Ejercicio 3: Perfil de usuario con props en 3 niveles

Crea los componentes `App`, `TarjetaPerfil`, `DatosContacto` y `Teléfono`. `App` tiene un objeto usuario con nombre, email y teléfono. `TarjetaPerfil` recibe usuario y muestra el nombre; `DatosContacto` recibe usuario y muestra el email; `Teléfono` recibe usuario y muestra el teléfono. Dibuja el árbol de componentes en un comentario y marca qué componentes son intermediarios.

SOLUCIÓN: El árbol queda así: `App -> TarjetaPerfil -> DatosContacto -> Teléfono`. `TarjetaPerfil` y `DatosContacto` son intermediarios: reciben el objeto usuario completo y lo pasan al siguiente nivel. `Teléfono` es la hoja que por fin usa el dato del teléfono.

```

function App() {
  const usuario = {
    nombre: "Ana López",
    email: "ana@correo.com",
    telefono: "555-1234"
  }
  return <TarjetaPerfil usuario={usuario} />
}

function TarjetaPerfil({ usuario }) {
  return (
    <div>
      <h2>{usuario.nombre}</h2>
      <DatosContacto usuario={usuario} />
    </div>
  )
}

function DatosContacto({ usuario }) {
  return (
    <div>
      <p>Email: {usuario.email}</p>
      <Telefono usuario={usuario} />
    </div>
  )
}

function Telefono({ usuario }) {
  return <p>Tel: {usuario.telefono}</p>
}

// Árbol:
// App -> TarjetaPerfil (intermediario: usa nombre, pasa usuario)
//   -> DatosContacto (intermediario: usa email, pasa usuario)
//     -> Telefono (hoja: usa telefono)

```

Este es props drilling de manual con 2 intermediarios: aceptable según nuestras reglas. Cada componente usa una parte del objeto y pasa el resto. Como el objeto viaja completo, si mañana Teléfono también necesita el nombre, no hay que tocar a los intermediarios: ya les llega todo. Esa es la ventaja de pasar el objeto completo en vez de prop por prop cuando los datos pertenecen a la misma entidad.

Como reto extra, refactoriza con children: haz que DatosContacto reciba children en vez de usuario, y que TarjetaPerfil le pase <Teléfono usuario={usuario} /> ya armado. El árbol se

acorta y DatosContacto deja de ser repartidor. Compara ambas versiones y decide cuál se lee mejor: esa comparación es exactamente el criterio que usarás en proyectos reales.

Ejercicio 4: Buscador que muestra "sin resultados" si la lista queda vacía

Crea un componente BuscadorFrutas con un arreglo fijo de 6 frutas, un input para filtrar por texto, y la lista filtrada debajo. Si el filtro no coincide con ninguna fruta, muestra el mensaje amigable "Sin resultados para 'X'" donde X es lo que escribió el usuario. El mensaje debe incluir lo que buscó para que sepa por qué no hay nada.

SOLUCIÓN: La idea central es calcular la lista filtrada ANTES del return (lógica arriba, JSX tonto abajo) y luego usar un ternario para elegir entre la lista y el mensaje. El filtro usa toLowerCase en ambos lados para que "MANZANA" encuentre "manzana".

```

import { useState } from "react"

function BuscadorFrutas() {
  const frutas = [
    "manzana", "plátano", "naranja",
    "fresa", "mango", "uva"
  ]
  const [busqueda, setBusqueda] = useState("")

  const filtradas = frutas.filter(f =>
    f.toLowerCase().includes(busqueda.toLowerCase())
  )

  return (
    <div>
      <h2>Buscador de frutas</h2>
      <input
        value={busqueda}
        onChange={e => setBusqueda(e.target.value)}
        placeholder="Busca una fruta"
      />
      {filtradas.length === 0 ? (
        <p>Sin resultados para "{busqueda}"</p>
      ) : (
        <ul>
          {filtradas.map(f => <li key={f}>{f}</li>)}
        </ul>
      )}
    </div>
  )
}

export default BuscadorFrutas

```

Tres detalles a notar. Primero: filtradas se calcula en cada render con el valor actual de búsqueda, así que la lista se actualiza sola mientras escribes, sin botones ni eventos extra. Segundo: el mensaje de "sin resultados" incluye {búsqueda} entre comillas, para que el usuario entienda que el filtro funcionó pero no hubo coincidencias (no es lo mismo "no hay nada" que "no hay nada PARA LO QUE BUSCASTE"). Tercero: usamos la fruta misma como key porque en este arreglo fijo no hay duplicados; en datos reales de un servidor usarías un id.

Prueba escribir "xyz": verás el mensaje con tu texto. Borra el input: la lista completa regresa. Este ejercicio combina estado (useState), derivación de datos (filter antes del return) y render

condicional (ternario para el caso vacío): las tres herramientas trabajando juntas, como en las apps reales.

Ejercicio 5: Dashboard con widgets condicionales según el rol

Crea un componente Dashboard que reciba rol por props ("admin", "editor" o "invitado"). Si el rol es "admin", muestra los widgets PanelUsuarios, Reportes y Configuración. Si es "editor", muestra Reportes y EditorContenido. Si es "invitado", muestra solo un mensaje "Inicia sesión para ver más". Usa una variable intermedia (no ternarios anidados) para elegir el contenido.

SOLUCIÓN: Con tres roles distintos, los ternarios anidados serían un desastre. La variable intermedia contenido deja la lógica legible de arriba a abajo, y el return queda limpio. Los widgets pueden ser componentes simples de una línea para este ejercicio; lo importante es la estructura de decisión.

```

function PanelUsuarios() {
  return <div>Widget: usuarios registrados</div>
}
function Reportes() {
  return <div>Widget: reportes de ventas</div>
}
function Configuracion() {
  return <div>Widget: configuración del sitio</div>
}
function EditorContenido() {
  return <div>Widget: editor de artículos</div>
}

function Dashboard({ rol }) {
  let contenido
  if (rol === "admin") {
    contenido = (
      <>
        <PanelUsuarios />
        <Reportes />
        <Configuracion />
      </>
    )
  } else if (rol === "editor") {
    contenido = (
      <>
        <Reportes />
        <EditorContenido />
      </>
    )
  } else {
    contenido = <p>Inicia sesión para ver más.</p>
  }

  return (
    <div>
      <h2>Panel principal ({rol})</h2>
      {contenido}
    </div>
  )
}

export default Dashboard

```

Observa el fragmento `<> </>` envolviendo los widgets de cada rol: como contenido debe ser UN solo valor, agrupamos los widgets sin agregar etiquetas extra al HTML. También nota que Reportes aparece en dos roles sin duplicar su código: el componente se reutiliza, que es exactamente para lo que sirven. El `else` final atrapa "invitado" y cualquier valor inesperado: ese es tu componente de guardia disfrazado, protegiendo el dashboard de roles desconocidos.

Para probarlo, dibuja tres dashboards en tu App: `<Dashboard rol="admin" />`, `<Dashboard rol="editor" />` y `<Dashboard rol="invitado" />`. Verás tres paneles distintos con el mismo componente. Este patrón (contenido según rol o permiso) aparece en literalmente toda app con login: menús que cambian, botones que aparecen o desaparecen, páginas enteras que se protegen. Ya sabes construirlo.

Resumen de la unidad

- La composición de componentes es el arte de dividir una app en piezas pequeñas, cada una con una sola responsabilidad. Es la habilidad más importante de React: un componente gigante de 500 líneas siempre termina siendo un problema.
- El render condicional decide qué se dibuja según la situación. Las técnicas, de la más corta a la más explícita: el operador `&&` ("muestra solo si existe"), el ternario ("esto o lo otro"), la variable intermedia (para 3 o más casos) y el `early return` (sal temprano del caso problemático).
- A la izquierda del `&&` solo van booleanos. Si pones un número crudo como `items.length`, el 0 se dibuja como texto fantasma en pantalla. Usa comparaciones explícitas: `items.length > 0`.
- Los ternarios anidados son ilegibles: con más de dos opciones, saca la lógica a una variable intermedia o a un objeto mapa antes del `return`. El JSX es para dibujar, no para pensar.
- El `early return` atiende el caso urgente primero y deja el código principal limpio, sin protecciones repetidas. Encadenados, forman las `guard clauses`: puertas de salida ordenadas por urgencia.
- Los componentes de guardia (como `SoloLogueados` o `RutaProtegida`) protegen contenido usando `children`: revisan la condición en la puerta y dejan pasar lo de adentro sin que se entere.
- Toda petición de datos vive en cuatro estados y tu interfaz necesita una cara para cada uno: cargando, error, vacío y datos. El más olvidado (y el que más profesionalismo aporta) es el vacío: nunca dejes una pantalla en blanco.

- Los componentes anidados forman un árbol: App es el tronco, los intermedios coordinan secciones y las hojas dibujan detalles. Divide cuando algo se repite, tiene nombre claro, el archivo creció demasiado o mezcla niveles de detalle.
- Un componente por archivo en la carpeta components/, nombrado en PascalCase por lo que ES (TarjetaProducto, no CosaAzul). Empieza simple y subdivide la carpeta por feature cuando pase de 15 archivos.
- Los presentacionales solo dibujan a partir de props (fáciles de reutilizar y probar); los contenedores manejan estado, fetch y decisiones, y les pasan todo digerido a los presentacionales. Es una idea general, no una ley.
- El props drilling es pasar datos de mano en mano (Padre -> Hijo -> Nieto). Con 1 o 2 intermediarios está perfectamente bien; con 3 o más niveles de repartidores que ni usan el dato, empieza a doler: refactors dolorosos, firmas contaminadas y rigidez.
- La composición con children elimina intermediarios: el padre pasa los hijos ya configurados y el componente del medio solo los enmarca. El Context (Unidad 8) es la artillería pesada para datos que necesita media aplicación a gran profundidad.
- Empieza siempre con props simples y migra solo cuando el drilling duela de verdad. El código simple que funciona le gana al elegante que nadie entiende.
- En la próxima unidad seguirás construyendo sobre esta base: con componentes bien divididos y render condicional dominado, estás listo para manejar formularios y eventos de usuario como un profesional.

Cierra los ojos un momento e imagina la primera app grande que vas a construir: quizá la tienda en línea del proyecto final, quizá una idea propia. Ahora mírala con tus nuevos ojos de rayos X. Ya no ves "una página", ves un árbol: un Layout que enmarca todo, un Header que saluda o pide login según haya usuario, un Contenido que muestra "cargando" mientras viajan los datos, una Lista que dibuja Tarjetas, cada Tarjeta con su imagen, su precio y su botón. Cada pieza en su archivo, cada archivo con su responsabilidad, cada dato viajando por props justo hasta donde se necesita y ni un nivel más. Eso que acabas de imaginar no es un sueño lejano: es exactamente lo que sabes hacer al terminar esta unidad.

Y aquí va un consejo de cierre que vale oro: la composición es un músculo, no un dato. No se aprende de memoria, se entrena. Cada vez que escribas un componente, pregúntate las tres preguntas de esta unidad: ¿qué se muestra y cuándo? (render condicional), ¿esta pieza hace una sola cosa? (componentes anidados), ¿este dato viaja más niveles de los necesarios? (props drilling). Si las respondes con honestidad en cada proyecto, dentro de unos meses mirarás tu primer código y te dará ternura lo mucho que avanzaste. Esa es la señal de que vas por buen camino. Nos vemos en la Unidad 7, donde tus componentes empezarán a escuchar al usuario: formularios, inputs y eventos.

UNIDAD 7: Children, estilos y referencias

Ya llevas seis unidades aprendiendo React, y si has seguido el curso paso a paso, a estas alturas ya sabes crear componentes, pasarles props, manejar estado con `useState` y repetir elementos con listas. Eso te convierte en alguien que puede construir interfaces que funcionan. Pero en esta unidad vas a dar un salto importante: vas a aprender a construir componentes que se sienten profesionales, que se ven bien y que saben manejar esos casos especiales donde React necesita hablar directamente con el navegador. Los tres temas de esta unidad (`children`, estilos y `useRef`) son los que separan una aplicación que apenas funciona de una que está bien construida.

Piensa en lo siguiente: hasta ahora tus componentes han sido como máquinas que reciben instrucciones por props y devuelven algo en pantalla. Eso está bien, pero hay una limitación que quizá ya notaste. Imagina que quieres hacer una tarjeta reutilizable: un cuadro con borde redondeado y sombra donde después puedas meter cualquier cosa (un texto, una imagen, un formulario, lo que sea). Con solo props tendrías que pasarle el contenido como un texto o como una prop especial, y se sentiría rígido, como meter ropa en una caja cerrada con la tapa pegada. Lo natural sería abrir la caja, meter lo que quieras y cerrarla. Eso es exactamente lo que hace la prop especial `children`: te permite abrir el componente, poner cosas adentro y que el componente las muestre donde el decida. `Children` convierte a tus componentes en contenedores flexibles, en cajas abiertas que reciben lo que sea. Las tarjetas, los modales, los layouts de página, las alertas, las barras laterales... todo eso se construye con `children`.

El segundo tema, los estilos, responde a una pregunta que seguro ya te hiciste: mis componentes funcionan, pero se ven planos, se ven feos. Como los pinto. React no inventó una forma rara de poner estilos; usa el CSS que ya conoces, pero con un par de reglas propias que tienes que entender para no tropezar. La más famosa: en JSX no escribes `class`, escribes `className`. Parece un detalle tonto, pero si lo olvidas, tu estilo simplemente no se aplica y te pasas veinte minutos pensando que el CSS está mal cuando el problema es una sola palabra. Además vas a aprender a poner estilos en línea con objetos de JavaScript (esa sintaxis de doble llave que al principio parece un error de dedo), a cambiar estilos según el estado del componente (un botón que se pone rojo cuando hay error, un semáforo que prende la luz correcta) y a entender CSS Modules, que es la forma elegante de evitar que los estilos de un componente manchen a otro.

El tercer tema, `useRef`, es la "salida de emergencia" de React. Casi todo lo que haces en React pasa por el estado y por el renderizado: cambias el estado, React vuelve a dibujar. Pero hay cosas que no necesitan dibujarse de nuevo. Por ejemplo: quieres enfocar un `input` cuando el usuario hace clic en un botón. No necesitas que la pantalla se vuelva a dibujar, solo

necesitas decirle al navegador "oye, pon el cursor aquí". O quieres medir el ancho de un elemento, o guardar el identificador de un temporizador para poder detenerlo después. Para eso existe useRef: una cajita que guarda un valor que sobrevive entre renders pero que no provoca un nuevo render cuando cambia. Es como tener el número de teléfono directo del elemento del DOM, sin pasar por la recepción de React.

Estos tres temas están conectados por una idea comun: hacer componentes más potentes y más reales. Children te da componentes contenedores reutilizables. Los estilos te dan componentes que se ven bien y que reaccionan visualmente. useRef te da acceso quirurgico al DOM para esos detalles que el estado no puede o no debe manejar. Al final de esta unidad vas a poder construir un layout completo para tus páginas, un modal reutilizable que muestre cualquier contenido, botones que cambian de color según lo que pasa, y formularios donde el foco salta al campo correcto automáticamente. Eso ya es nivel de aplicación real, no de ejercicio de juguete.

Una advertencia honesta antes de empezar: useRef es tentador. Cuando descubres que puedes tocar el DOM directamente, dan ganas de usarlo para todo. Resiste esa tentación. React tiene una filosofía: el estado manda, la pantalla es un reflejo del estado. useRef es la excepción, no la regla. Te voy a enseñar exactamente cuando usarlo y cuando no, porque abusar de el es uno de los errores más comunes de quienes están aprendiendo. Lo mismo con los estilos en línea: son prácticos, pero si los usas para todo terminas con un código difícil de mantener. Cada herramienta tiene su lugar, y esta unidad te va a dejar clarísimo cual es.

Así que ponte comodo. Vamos a abrir cajas con children, a pintar con className y style, y a tomar el control directo con useRef. Al final tendrás cinco ejercicios resueltos para practicar todo junto, porque estos tres temas brillan más cuando se combinan: un modal (children) con estilos dinamicos que enfoca automáticamente su primer campo (useRef). Eso es React de verdad. Empecemos.

Explicación detallada

Que es children

En React, children es una prop especial que todo componente recibe automáticamente. No tienes que declararla ni pasarla como las demás props: React la llena solo con lo que pongas entre las etiquetas de apertura y cierre del componente. Mira este ejemplo:

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

Y así se usa:

```
<Tarjeta>
  <p>Contenido</p>
</Tarjeta>
```

Lo que va dentro de `<Tarjeta>...</Tarjeta>`, en este caso el párrafo `<p>Contenido</p>`, llega al componente dentro de la prop `children`. El componente decide donde mostrarlo: en el ejemplo, dentro del `div` con clase "tarjeta". El componente es el marco, `children` es la foto que pones dentro del marco. Puedes cambiar la foto sin cambiar el marco, y puedes usar el mismo marco con mil fotos distintas.

Fíjate en algo importante: `children` no es magia, es simplemente una prop más. La única diferencia es que no la escribes como atributo (no pones `children="algo"`), sino que React la construye con el contenido anidado. Por dentro, es como si React hiciera esto:

```
<Tarjeta children={<p>Contenido</p>} />
```

Pero nadie escribe eso porque la sintaxis de anidado es mucho más legible. La desestructuración `function Tarjeta({ children })` es la forma habitual de sacarla de las props, igual que haces con cualquier otra prop.

Que puede contener children

Children puede ser casi cualquier cosa que React sepa dibujar:

- Texto simple: `<Tarjeta>Hola mundo</Tarjeta>` (`children` es el string "Hola mundo").
- Un elemento: `<Tarjeta><p>Contenido</p></Tarjeta>`.
- Varios elementos: puedes poner muchos hijos, y `children` será un arreglo de elementos.
- Otros componentes: `<Tarjeta><FormularioLogin /></Tarjeta>`.
- Expresiones de JavaScript: `<Tarjeta>{usuario.nombre}</Tarjeta>`.

- Mezclas de todo lo anterior.

Un ejemplo con varios hijos:

```
<Tarjeta>
  <h2>Mi titulo</h2>
  <p>Primer parrafo.</p>
  <p>Segundo parrafo.</p>
  <button>Comprar</button>
</Tarjeta>
```

Aquí `children` es un arreglo con cuatro elementos, y el componente los dibuja todos donde ponga `{children}`. No tienes que hacer nada especial para manejar varios hijos: React los renderiza en orden, tal como los escribiste.

Analogía para que se te grabe: piensa en `children` como el contenido de una caja de regalo. La caja (el componente `Tarjeta`) es siempre la misma por fuera: mismo papel, mismo listón, mismo tamaño. Pero lo que hay adentro cambia en cada regalo. El que envuelve el regalo (quien usa el componente) decide que meter; la caja solo se encarga de verse bonita y de presentar el contenido. Si la caja tuviera que fabricarse de nuevo para cada tipo de regalo, sería un desperdicio. Con `children` fabricas la caja una vez y la reutilizas con cualquier contenido.

Componentes contenedores (wrappers)

El patrón más importante que `children` habilita se llama componente contenedor o `wrapper`: un componente cuya chamba es envolver contenido con estructura o estilo común. Los vas a ver en todas las aplicaciones React del mundo. Los casos típicos:

1. Tarjetas: un borde, una sombra y un padding que se repiten en toda la app.
2. Modales y diálogos: el fondo oscuro y el cuadro centrado son siempre iguales; lo que cambia es el contenido.
3. Layouts: la estructura de la página (encabezado, menú, pie) es fija; el contenido de cada página cambia.
4. Alertas y notificaciones: el estilo de la alerta (color, ícono) lo decide el contenedor; el mensaje lo pone quien la usa.
5. Botoneras y grupos: un contenedor que acomoda varios botones con espacio entre ellos.

La gracia es que el contenedor no necesita saber nada del contenido. La `Tarjeta` no sabe si adentro hay texto, un formulario o un video; solo sabe verse como tarjeta. Eso es separación

de responsabilidades en su forma más pura, y es lo que hace que tu código sea reutilizable de verdad.

Mira un ejemplo de Layout, que es quizá el uso más valioso de children:

```
function Layout({ children }) {  
  return (  
    <div>  
      <header className="encabezado">  
        <h1>Lucas EGA Academy</h1>  
        <nav>  
          <a href="/">Inicio</a>  
          <a href="/cursos">Cursos</a>  
        </nav>  
      </header>  
      <main className="contenido">  
        {children}  
      </main>  
      <footer className="pie">  
        <p>Aprende tecnologia gratis</p>  
      </footer>  
    </div>  
  );  
}
```

Y cada página lo usa así:

```
function PaginaInicio() {
  return (
    <Layout>
      <h2>Bienvenido</h2>
      <p>Esta es la pagina de inicio.</p>
    </Layout>
  );
}

function PaginaCursos() {
  return (
    <Layout>
      <h2>Cursos</h2>
      <ul>
        <li>HTML</li>
        <li>CSS</li>
        <li>React</li>
      </ul>
    </Layout>
  );
}
```

Sin children tendrías que repetir el header y el footer en cada página, o inventar un sistema de props torpe para pasar el contenido. Con children, el layout se escribe una vez y cada página solo se preocupa por lo suyo. Cuando quieras cambiar el menu de navegación, lo cambias en un solo lugar.

Composicion vs props drilling

Aquí hay una idea de diseño que vale oro. Muchas veces necesitas que un componente intermedio pase datos o contenido a un componente más profundo. La forma ingenua es el props drilling: pasar props nivel por nivel, como una cubeta que va de mano en mano. Funciona, pero cada componente intermedio tiene que conocer y reenviar props que no le importan, y si cambia algo hay que tocar todos los niveles.

Children ofrece una alternativa elegante: la composición. En lugar de que el componente padre baje datos por props a través de intermediarios, el que usa los componentes los anida directamente. Ejemplo del problema:

```
// Props drilling: Lista recibe items solo para pasarselos a ItemLista
function Lista({ items }) {
  return (
    <ul>
      {items.map(item => <ItemLista key={item.id} item={item} />)}
    </ul>
  );
}
```

Con composición, Lista no necesita saber nada de los items:

```
function Lista({ children }) {
  return <ul className="lista">{children}</ul>;
}

// Quien la usa compone directamente:
<Lista>
  <ItemLista item={producto1} />
  <ItemLista item={producto2} />
  <ItemLista item={producto3} />
</Lista>
```

Lista se vuelve un contenedor tonto y reutilizable; la decisión de que items mostrar queda donde debe estar, en el componente que tiene los datos. Menos acoplamiento, menos props pasando de mano en mano, código más claro.

Ojo: esto no significa que el props drilling sea siempre malo. Para datos simples que bajan uno o dos niveles, pasar props es perfecto y es lo más legible. La composición con children brilla cuando el contenido es complejo (elementos, componentes enteros) o cuando quieres que el contenedor sea agnostico a lo que envuelve. Regla práctica: si lo que pasas se puede expresar como un valor simple (un texto, un número, un booleano), usa props. Si lo que pasas es estructura (elementos JSX, componentes), usa children.

También puedes combinar ambos: un Modal puede recibir el título por prop y el cuerpo por children:

```
function Modal({ titulo, onCerrar, children }) {
  return (
    <div className="fondo-modal">
      <div className="caja-modal">
        <h2>{titulo}</h2>
        <div className="cuerpo-modal">{children}</div>
        <button onClick={onCerrar}>Cerrar</button>
      </div>
    </div>
  );
}
```

Uso:

```
<Modal titulo="Confirmar compra" onCerrar={cerrar}>
  <p>Estas seguro de comprar este curso?</p>
  <p>Total: $27 USD</p>
</Modal>
```

El título es un dato simple (prop), el cuerpo es estructura variable (children). Esa combinación es el patrón que más vas a usar en la vida real.

Un detalle técnico útil: si necesitas saber cuantos hijos hay o transformarlos, React te da la utilidad `React.Children` con funciones como `React.Children.count(children)` y `React.Children.map(children, ...)`. No la necesitas para lo básico, pero existe cuando quieras inspeccionar o modificar los hijos desde el contenedor, por ejemplo para agregarles una prop extra a cada uno.

Estilos en React a fondo

En React los estilos se ponen con el CSS de toda la vida, pero JSX impone sus propias reglas porque lo que escribes no es HTML puro, es JavaScript disfrazado. Hay dos formas principales de dar estilo: con clases CSS (`className`) y con estilos en línea (`style`). Vamos a dominar las dos y a aprender cuando conviene cada una.

className: la forma más común

En HTML escribes `<div class="caja">`. En JSX eso no funciona: tienes que escribir `<div className="caja">`. La razón es simple: `class` es una palabra reservada en JavaScript (se usa para declarar clases), y como JSX se convierte a JavaScript, usar `class` ahí causaría conflictos. React decidió llamarle `className`, que además es el nombre de la propiedad del DOM que representa el atributo `class`.

Este es el error número uno de todo principiante en React: escribes `class` por costumbre, el navegador no se queja, React te pone una advertencia en la consola, y tu estilo no se aplica. Te quedas mirando tu CSS jurando que está bien escrito, y el problema era una sola palabra. Grabatelo a fuego: en JSX siempre `className`, nunca `class`.

El flujo de trabajo habitual es este: creas un archivo CSS normal, lo importas en tu componente, y usas `className` para aplicar las clases.

```
// Boton.css
.boton {
  padding: 10px 20px;
  border: none;
  border-radius: 8px;
  background-color: #2e7d32;
  color: white;
  cursor: pointer;
}

.boton:hover {
  background-color: #1b5e20;
}
```

```
// Boton.js
import "./Boton.css";

function Boton({ children, onClick }) {
  return (
    <button className="boton" onClick={onClick}>
      {children}
    </button>
  );
}
```

Esa línea `import "./Botón.css"` es la que conecta el CSS con el componente. Las herramientas de React (como Vite o Create React App) se encargan de incluir esos estilos en la página. Así de simple: CSS normal, clases normales, solo cambia `class` por `className` en el JSX.

Ventajas de este enfoque: es el CSS que ya conoces, con pseudoclases (:hover, :focus), media queries para responsive, animaciones con @keyframes, y toda la potencia de siempre. Además mantiene la separación de responsabilidades: la estructura en el JS, la presentación en el CSS. Para proyectos medianos y grandes, esta es la forma recomendada.

El problema que puede aparecer: las clases son globales. Si en otro archivo CSS defines .botón con otro color, los estilos chocan y el último que cargue gana, lo cual es una pesadilla de debugging. Para eso existen los CSS Modules, que vemos más abajo.

Puedes aplicar varias clases a un elemento separandolas con espacio, igual que en HTML:

```
<div className="tarjeta tarjeta-destacada">...</div>
```

Y puedes combinar clases fijas con clases condicionales usando template strings (lo veremos en estilos dinámicos).

Estilos en línea con el atributo style

La segunda forma es pasar un objeto de JavaScript al atributo style. La sintaxis tiene una doble llave que al principio confunde:

```
<p style={{ color: "red", fontSize: "20px" }}>Texto rojo grande</p>
```

Por qué dos llaves. La llave externa es la de JSX: le dice a React "lo que sigue es JavaScript, no texto". La llave interna es la del objeto de JavaScript con los estilos. Desglosado se entiende mejor:

```
// Es lo mismo que hacer esto:
const estilos = { color: "red", fontSize: "20px" };
<p style={estilos}>Texto rojo grande</p>
```

Cuando ves {{ }}, piensa: "llave de JSX + objeto de estilos". Con la variable intermedia solo hay una llave porque el objeto ya existe.

Dos reglas del objeto de estilos:

1. Las propiedades se escriben en camelCase, no con guiones. En CSS escribes background-color; en el objeto escribes backgroundColor. font-size se vuelve fontSize, margin-top se vuelve marginTop. La razón: los guiones no son válidos en nombres de propiedades de JavaScript (a.b-c se interpretaría como una resta), así que React usa la versión camelCase que coincide con las propiedades del DOM en JavaScript.

2. Los valores son strings en la mayoría de los casos: color: "red", fontSize: "20px". Pero hay una excepción cómoda: si el valor es un número, React le agrega "px" automáticamente para las propiedades que usan pixeles. Así que fontSize: 20 es lo mismo que fontSize: "20px". Ojo: esto solo funciona con propiedades de longitud; para otras como zIndex: 10 o opacity: 0.5 el número se usa tal cual, y para porcentajes necesitas el string: width: "50%".

Ejemplo completo:

```
function Caja({ titulo, children }) {  
  const estilosCaja = {  
    border: "2px solid #333",  
    borderRadius: "12px",  
    padding: "16px",  
    backgroundColor: "#f5f5f5",  
    maxWidth: 400  
  };  
  
  return (  
    <div style={estilosCaja}>  
      <h3 style={{ color: "#1a237e", marginTop: 0 }}>{titulo}</h3>  
      {children}  
    </div>  
  );  
}
```

Cuando conviene el estilo en línea: cuando el valor del estilo depende de JavaScript de forma dinámica (un color que viene de una prop, un ancho calculado), o para estilos rápidos de un solo uso en prototipos. Cuando NO conviene: para diseñar componentes completos, porque pierdes pseudoclases (:hover no existe en estilos en línea), media queries, y terminas con objetos gigantes mezclados con tu lógica. Un componente con 40 líneas de objeto de estilos es difícil de leer y de mantener. Regla práctica: estilos en línea para lo dinámico y puntual; archivos CSS para el diseño base.

Estilos condicionales y dinámicos

Aquí es donde React brilla: como el estilo es JavaScript, puede reaccionar al estado. La técnica más usada es el operador ternario dentro de className o style.

Con clases y template strings:

```
function Alerta({ tipo, children }) {
  // tipo puede ser "exito", "error" o "info"
  return (
    <div className={`alerta alerta-${tipo}`}>
      {children}
    </div>
  );
}
```

Si tipo es "error", el className resultante es "alerta alerta-error", y en tu CSS defines cómo se ve cada variante. Limpio y escalable: agregar un nuevo tipo es agregar una clase CSS, sin tocar el componente.

Con el atributo style y ternarios:

```
function Boton({ primario, children }) {
  const estilos = {
    padding: "10px 20px",
    borderRadius: "8px",
    border: "none",
    cursor: "pointer",
    backgroundColor: primario ? "#1565c0" : "#e0e0e0",
    color: primario ? "white" : "#333"
  };

  return <button style={estilos}>{children}</button>;
}
```

Si la prop primario es true, el botón sale azul con texto blanco; si no, gris con texto oscuro. El estilo se recalcula en cada render, así que si primario cambia, el botón cambia de color automáticamente. Eso es reactividad visual sin esfuerzo.

Otro patrón común: construir el className con lógica más elaborada usando un arreglo y join, o simplemente concatenando:

```
function Tarjeta({ destacada, children }) {
  let clases = "tarjeta";
  if (destacada) {
    clases += " tarjeta-destacada";
  }
  return <div className={clases}>{children}</div>;
}
```

Para lógica más compleja existen librerías como `clsx` o `classnames`, pero con `template strings` y `ternarios` tienes suficiente para el 90% de los casos.

Ejemplo que combina estado y estilo dinámico, el típico semáforo:

```
function Semaforo() {
  const [luz, setLuz] = useState("rojo");

  const colorFoco = (nombre) => ({
    width: 60,
    height: 60,
    borderRadius: "50%",
    backgroundColor: luz === nombre ? nombre : "#444",
    margin: "8px"
  });

  return (
    <div>
      <div style={colorFoco("red")}></div>
      <div style={colorFoco("yellow")}></div>
      <div style={colorFoco("green")}></div>
      <button onClick={() => setLuz("red")}>Rojo</button>
      <button onClick={() => setLuz("yellow")}>Amarillo</button>
      <button onClick={() => setLuz("green")}>Verde</button>
    </div>
  );
}
```

Cada foco se pinta con su color solo si es la luz activa; los demás quedan apagados en gris oscuro. Cambias el estado y los estilos se actualizan solos. Nota como la función `colorFoco` devuelve un objeto de estilos distinto según el estado: esa es la esencia de los estilos dinámicos.

CSS Modules: la idea general

El problema de las clases globales se resuelve con CSS Modules. La idea: en lugar de importar un CSS normal, importas un archivo que termina en `.module.css`, y las clases que defines ahí se vuelven locales al componente que las importa. No importa si otro componente usa la misma clase `.botón`: cada uno tendrá la suya sin choques.

```
// Boton.module.css
.boton {
  padding: 10px 20px;
  border-radius: 8px;
}
```

```
// Boton.js
import estilos from "./Boton.module.css";

function Boton({ children }) {
  return <button className={estilos.boton}>{children}</button>;
}
```

Fíjate en la diferencia: importas el CSS como un objeto (estilos) y usas estilos.botón en el className. Por debajo, la herramienta de construcción renombra la clase a algo único como Boton_boton_3a7f, así que nunca choca con otra .botón de otro archivo. Tu escribes nombres simples y descriptivos sin miedo.

Cuando usar cada enfoque, en resumen:

- Archivo CSS normal + className: proyectos pequenos, estilos compartidos globales (como un reset o tipografias base), cuando ya tienes un sistema CSS armado.
- CSS Modules: componentes reutilizables en proyectos medianos o grandes, cuando quieres evitar choques de nombres sin complicarte.
- Estilos en línea (style con objeto): valores dinamicos que dependen de props o estado, ajustes puntuales rápidos.
- Librerías de componentes (como Tailwind o styled-components): existen y son populares, pero son tema de otro curso; con lo que viste aquí puedes construir interfaces completas.

Variables CSS

Un truco que combina lo mejor de los dos mundos: define variables CSS en tu archivo global para los colores y medidas de tu marca, y usalas tanto en clases como en estilos en línea.

```
/* estilos-globales.css */
:root {
  --color-primario: #1565c0;
  --color-exito: #2e7d32;
  --color-error: #c62828;
  --radio-borde: 8px;
}
```

En clases las usas normal: `background-color: var(--color-primario)`. Y en estilos en línea también funcionan porque al final es CSS: `style={{ backgroundColor: "var(--color-primario)" }}`. Así, si mañana cambias el color primario de tu academia, lo cambias en un solo lugar y toda la app se actualiza. Es una forma barata y poderosa de tener un mini sistema de diseño.

useRef a fondo

Que es una referencia

`useRef` es un Hook de React que te devuelve un objeto mutable que sobrevive entre renders. Su forma básica:

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

function MiComponente() {
  const miRef = useRef(null);
  // ...
}
```

`useRef` te entrega un objeto simple con una sola propiedad importante: `.current`. Lo que guardes en `miRef.current` se mantiene mientras el componente exista, aunque el componente se vuelva a dibujar cien veces. Y aquí está la diferencia clave con `useState`: cambiar `.current` NO provoca un nuevo render. Es una cajita silenciosa: guarda cosas sin avisarle a React.

El valor que le pasas a `useRef(valorInicial)` es el valor inicial de `.current`. Por convención se usa `null` cuando la ref va a apuntar a un elemento del DOM, porque al principio el elemento todavía no existe.

Analogía: si `useState` es un pizarrón donde todo lo que escribes lo ve la clase (y cada vez que escribes algo, todos voltean a ver, es decir, hay un re-render), `useRef` es una nota adhesiva pegada debajo de tu escritorio. Está ahí, la puedes leer y cambiar cuando quieras, pero nadie voltear a ver cuando la modificas. A veces necesitas justo eso: guardar algo sin armar alboroto.

El uso principal: acceso directo al DOM

El caso más común de useRef es obtener acceso directo a un elemento del DOM. Lo haces en dos pasos: creas la ref y se la pasas al elemento con el atributo ref.

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

  function enfocar() {
    inputRef.current.focus();
  }

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

Paso a paso: `const inputRef = useRef(null)` crea la cajita con `.current` en `null`. Cuando React dibuja el `<input ref={inputRef} />`, mete el elemento real del DOM dentro de `inputRef.current`. A partir de ese momento, `inputRef.current` es el input de verdad, y puedes llamarle métodos del DOM como `.focus()`, que pone el cursor dentro del campo. Al hacer clic en el botón, se ejecuta `enfocar` y el cursor salta al input mágicamente.

Esto sirve para muchas operaciones directas del DOM:

- Enfocar un campo: `inputRef.current.focus()` (formularios, accesibilidad, modales que enfocan su primer campo al abrirse).
- Medir un elemento: `inputRef.current.offsetWidth`, `.offsetHeight`, o `.getBoundingClientRect()` para saber su tamaño y posición en pantalla.
- Hacer scroll: `elementoRef.current.scrollToView()` para llevar la vista hasta una sección.
- Reproducir medios: `videoRef.current.play()` en un `<video>`.
- Seleccionar texto: `inputRef.current.select()`.

Ejemplo de medición:

```
function CajaMedible() {
  const cajaRef = useRef(null);
  const [ancho, setAncho] = useState(0);

  function medir() {
    if (cajaRef.current) {
      setAncho(cajaRef.current.offsetWidth);
    }
  }

  return (
    <div>
      <div ref={cajaRef} style={{ padding: 20, border: "1px solid black" }}>
        Mide mi ancho
      </div>
      <button onClick={medir}>Medir</button>
      <p>El ancho es: {ancho} pixeles</p>
    </div>
  );
}
```

Nota el if (cajaRef.current): es una buena práctica verificar que la ref ya apunta a algo antes de usarla, porque en el primer render el elemento aun no existe y .current es null. En la práctica, después del primer render la ref ya está llena, pero el chequeo evita errores raros.

Guardar valores mutables sin re-render

El segundo gran uso de useRef no tiene nada que ver con el DOM: guardar cualquier valor mutable que necesites conservar entre renders pero que no deba redibujar la pantalla al cambiar. El ejemplo clásico es el identificador de un intervalo o un timeout.


```

function Cronometro() {
  const [segundos, setSegundos] = useState(0);
  const intervaloRef = useRef(null);

  function iniciar() {
    if (intervaloRef.current !== null) return; // ya esta corriendo
    intervaloRef.current = setInterval(() => {
      setSegundos(s => s + 1);
    }, 1000);
  }

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

  function reiniciar() {
    detener();
    setSegundos(0);
  }

  return (
    <div>
      <p>Tiempo: {segundos} s</p>
      <button onClick={iniciar}>Iniciar</button>
      <button onClick={detener}>Detener</button>
      <button onClick={reiniciar}>Reiniciar</button>
    </div>
  );
}

```

Por qué el id del intervalo va en una ref y no en el estado. Porque el id del intervalo no se dibuja en pantalla: a nadie le importa verlo, solo necesitas guardarlo para poder detener el intervalo después. Si lo guardaras en el estado, cada vez que inicias o detienes habría un render extra innecesario. Con la ref, el valor persiste entre renders (si fuera una variable normal dentro del componente, se perdería en cada render) sin provocar renders de más. Es el lugar perfecto para: ids de intervalos y timeouts, contadores internos que no se muestran, el valor anterior de una prop, o cualquier dato "de trabajo" del componente.

Ojo con un matiz importante: como cambiar `.current` no provoca render, si guardas en una ref algo que SI se debe ver en pantalla, la pantalla no se va a actualizar. Ese es exactamente el error de abusar de `useRef` para lo que debe ser estado, y lo veremos en la sección de errores comunes.

Diferencia entre useRef y useState

Tabla comparativa en texto:

- Propósito: useState guarda datos que se muestran en pantalla y que al cambiar deben redibujar el componente. useRef guarda datos que NO necesitan redibujar nada al cambiar (referencias al DOM, valores internos).
- Re-render: cambiar el estado provoca un nuevo render. Cambiar .current no provoca nada, es silencioso.
- Lectura en el render: el estado se lee directamente (segundos) y siempre tiene el valor actual del render. La ref se lee con .current y tiene el valor más reciente, incluso si cambió después del último render.
- Persistencia: ambos sobreviven entre renders (a diferencia de las variables normales declaradas en el cuerpo del componente, que se reinician en cada render).
- Mutabilidad: el estado nunca se muta directamente, se reemplaza con su función actualizadora (setSegundos). La ref se muta libremente asignando a .current.
- Momento de actualización: el estado se actualiza de forma asincrónica (React lo procesa y re-renderiza). La ref se actualiza de forma sincrónica e inmediata.

Regla de oro para decidir: "estado para lo que se dibuja, ref para lo que no". Preguntate: si este valor cambia, necesita la pantalla enterarse. Si la respuesta es sí (un contador visible, un texto, una lista), va en useState. Si la respuesta es no (el id de un intervalo, una referencia a un input, el valor anterior para comparar), va en useRef.

Un ejemplo donde se ve la diferencia con claridad:

```
// MAL: usar ref para algo visible (no se actualiza la pantalla)
function ContadorMalo() {
  const cuentaRef = useRef(0);
  function incrementar() {
    cuentaRef.current += 1; // la pantalla no se entera
  }
  return <button onClick={incrementar}>Clicks: {cuentaRef.current}</button>;
}

// BIEN: usar estado para algo visible
function ContadorBueno() {
  const [cuenta, setCuenta] = useState(0);
  function incrementar() {
    setCuenta(cuenta + 1); // React re-renderiza y muestra el nuevo valor
  }
  return <button onClick={incrementar}>Clicks: {cuenta}</button>;
}
```

En ContadorMalo haces clic y el número no cambia en pantalla, aunque por dentro la ref si aumento. Es el sintoma clásico de haber elegido mal la herramienta.

La regla: usalo solo cuando el estado no baste

useRef es la salida de emergencia, no la puerta principal. Usalo cuando:

- Necesitas llamar un método del DOM que React no expone como prop declarativa (focus, scroll, play, mediciones).
- Necesitas guardar un valor mutable entre renders sin provocar renders (id de intervalo, timeout).
- Necesitas el valor anterior de algo para compararlo (guardas el anterior en una ref dentro de un efecto).

NO lo uses para:

- Guardar datos que se muestran en pantalla (eso es estado).
- Evitar re-renders por "optimización" prematura (casi siempre es un error de diseno; si la pantalla debe reflejar el dato, el render es necesario y barato).

- Manipular el DOM para cambiar cosas que React ya maneja (como cambiar estilos o textos directamente con `.current.style` o `.current.textContent` cuando podrias hacerlo con estado y renderizado declarativo). Cada vez que tocas el DOM a mano, React y tu código pueden desincronizarse: React cree que la pantalla dice una cosa y tu la cambiaste por detras.

Piensalo así: React es declarativo (dices COMO debe verse la pantalla según el estado), y `useRef` te deja ser imperativo (dar ordenes directas: "enfocate", "midete"). Lo imperativo es necesario a veces, pero si lo usas para todo, pierdes todas las ventajas de React y terminas programando como en los tiempos de jQuery, peleando con el DOM a mano.

Analogias para no olvidar

- Children es la caja de regalo: el componente es la caja (siempre igual por fuera), children es lo que metes adentro (siempre distinto). Fabricas la caja una vez, la reutilizas mil veces.
- Children también es el marco de fotos: el marco no sabe que foto va a contener, solo sabe enmarcarla bien. Tu componente contenedor no debe saber que contenido va a envolver.
- `className` vs `class`: es como el nombre con el que te registraste en la escuela. Te llamas igual, pero en la lista oficial apareces con tu nombre completo. En JSX el "nombre oficial" del atributo es `className`; si escribes `class`, es como gritar un apodo que la lista no reconoce.
- El objeto de style con doble llave `{{ }}`: la llave de afuera es la puerta de JSX ("aquí empieza JavaScript"), la llave de adentro es la caja donde guardas los estilos. Puerta + caja = `{{ }}`.
- `useRef` es el control remoto directo del televisor: normalmente cambias de canal con la app del celular (el estado, que avisa a todos), pero a veces necesitas el control fisico para una función especial que la app no tiene (enfocar, medir). Lo usas puntual, no para todo.
- La ref como nota adhesiva secreta: el estado es el pizarron que todos ven; la ref es tu acordeon personal debajo del escritorio. Útil, pero si lo que escribes ahí lo necesita ver la clase, está en el lugar equivocado.

Errores comunes

1. Olvidar `className` y usar `class`. Sintoma: los estilos no se aplican y la consola muestra una advertencia. Solución: busca `class=` en tu JSX y cambialo por `className=`. Hazlo un habito desde el primer día.

2. Abusar de useRef para lo que debe ser estado. Sintoma: cambias la ref y la pantalla no se actualiza, o tu código se llena de manipulación manual del DOM. Solución: aplica la regla de oro (estado para lo que se dibuja, ref para lo que no). Si dudas, usa estado; es la opción segura.
3. Estilos en línea gigantes. Sintoma: objetos de 30 o 40 líneas mezclados con la lógica, componentes ilegibles, sin :hover ni media queries. Solución: pasa el diseño base a un archivo CSS con className y deja el style en línea solo para lo dinámico.
4. No desestructurar children y luego no saber de donde salió. Sintoma: ves {children} en el return y no entiendes qué es. Solución: recuerda que children viene de las props; escribir function Tarjeta({ children }) deja clarísimo que es una prop especial que React llena con el contenido anidado.
5. Pasar contenido por props cuando debería ser children. Sintoma: props como contenido="

hola

" pasadas como strings, o componentes que reciben media docena de props solo para armar su interior. Solución: si el contenido es estructura JSX, anídalos como children; reserva las props para datos simples.
6. Olvidar que cambiar .current no re-renderiza y luego leer la ref esperando ver el cambio en pantalla. Sintoma: "mi ref sí cambió pero no se ve". Solución: no es un bug, es el diseño de useRef. Lo visible va en estado.
7. Usar la ref antes de que exista el elemento. Sintoma: error "cannot read properties of null". Solución: verifica con if (miRef.current) antes de usarla, o úsala dentro de manejadores de eventos (ahí el elemento ya existe seguro).
8. Manipular el DOM con la ref para cosas que React maneja solo. Sintoma: haces miRef.current.style.color = "red" en lugar de usar estado y estilo dinámico, y después React sobrescribe tu cambio en el siguiente render. Solución: deja que React pinte; usa la ref solo para lo que React no puede hacer declarativamente.

Buenas prácticas

- Diseña primero los contenedores: cuando veas estructura repetida (tarjetas, secciones, modales), extrae un componente con children antes de copiar y pegar por tercera vez. La regla de tres aplica: si lo vas a repetir tres veces, es un componente.
- Combina props simples con children estructural: título, tipo, onClose por props; cuerpo y contenido variable por children. Así tus componentes son flexibles sin ser vagos.
- Nombra tus clases CSS con un prefijo del componente (tarjeta-título, modal-fondo) o usa CSS Modules; evita nombres genéricos como .caja o .botón en CSS global, porque tarde o temprano chocan.

- Centraliza colores y medidas en variables CSS (:root). Tu yo del futuro te lo va a agradecer cuando el cliente pida "cambiale el azul".
- Para estilos dinámicos simples prefiere clases condicionales (className con ternario) sobre objetos gigantes en style; el CSS sigue siendo mejor lugar para el diseño.
- Usa useRef con intención documentada: si otro programador ve tu ref, que entienda en un segundo por qué no es estado. Un comentario corto ayuda: // id del intervalo, no necesita render.
- Verifica la ref antes de usarla (if (ref.current)) cuando no estes seguro de que el elemento ya se dibujo.
- Mantén la filosofía declarativa: que el estado describa la pantalla y que useRef sea la excepción quirúrgica. Si tu componente tiene más refs que estados, algo huele mal; revisa el diseño.

Children avanzado: inspeccionar y transformar hijos

Hasta ahora usamos children como una caja negra: el contenedor lo dibuja tal cual con {children} y no pregunta que hay adentro. Pero a veces el contenedor necesita saber cuantos hijos tiene o hacerles algo a cada uno. Para eso React ofrece la utilidad React.Children, con funciones pensadas para trabajar con la prop children de forma segura (porque children puede ser un solo elemento, un arreglo o texto, y manejarlo a mano es propenso a errores).

Las más útiles:

- React.Children.count(children): te dice cuantos hijos hay. Útil para mostrar "3 elementos" o para decidir un layout según la cantidad.
- React.Children.map(children, función): aplica una función a cada hijo y devuelve el resultado. Ojo: no es el .map de los arreglos; esta versión funciona aunque children sea un solo elemento o texto.
- React.Children.toArray(children): convierte children a un arreglo normal, útil si necesitas filtrar o reordenar.

Ejemplo: una Botonera que separa automáticamente sus botones con espacio, sin que quien la use tenga que agregar márgenes:

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

function Botonera({ children }) {
  const total = Children.count(children);

  return (
    <div>
      <p>{total} botones:</p>
      <div className="botonera">
        {Children.map(children, (hijo, indice) => (
          <span key={indice} className="botonera-item">
            {hijo}
          </span>
        ))}
      </div>
    </div>
  );
}
```

Línea por línea: importamos Children de React. Children.count(children) nos da el número de hijos para mostrarlo. Children.map recorre cada hijo y lo envuelve en un span con clase "botonera-item" (que en el CSS tendrá el margen de separación). Cada hijo envuelto necesita su key, por eso usamos el índice. Quien usa la Botonera solo escribe los botones, y el contenedor se encarga del espaciado y del conteo:

```
<Botonera>
  <button>Guardar</button>
  <button>Cancelar</button>
  <button>Ayuda</button>
</Botonera>
```

Un patrón avanzado que vale la pena conocer de nombre: children como función (render props). En lugar de pasar elementos, pasas una función como children, y el componente la llama con datos para que decida que dibujar:

```
function DatosUsuario({ children }) {  
  const usuario = { nombre: "Esteban", cursos: 22 };  
  return <div>{children(usuario)}</div>;  
}  
  
// Uso: children es una funcion que recibe el usuario  
<DatosUsuario>  
  {(usuario) => <p>Hola, {usuario.nombre}</p>}  
</DatosUsuario>
```

Aquí children no es JSX, es una función que el componente ejecuta pasándole datos. Es un patrón poderoso para componentes que manejan lógica (cargar datos, toggles) y dejan que quien los usa decida la presentación. No lo necesitas todos los días, pero cuando lo veas en librerías ya sabrás que está pasando: children siendo una función en vez de elementos.

Estilos: organización en proyectos reales

Saber las técnicas de estilo es la mitad del trabajo; la otra mitad es organizarlas para que el proyecto no se vuelva un caos. Algunas reglas que usan los equipos profesionales:

1. Un archivo CSS por componente, junto a su archivo JS. Botón.js vive al lado de Botón.css (o Botón.module.css). Así encuentras los estilos del componente sin cazarlos por el proyecto.
2. Un CSS global solo para lo verdaderamente global: el reset de márgenes, la tipografía base, las variables de :root y los estilos del body. Todo lo demás vive con su componente.
3. Nombres de clases con prefijo del componente (bloque-elemento, estilo BEM simplificado): tarjeta, tarjeta-título, tarjeta-cuerpo, modal-fondo, modal-caja. Evita nombres genéricos como .contenido o .título en CSS global, porque algún día otro componente querrá ese nombre y chocarán.
4. Lo responsive va en el CSS, no en JavaScript. Las media queries son la herramienta natural:


```

/* Tarjeta.css */
.tarjeta {
  max-width: 350px;
  padding: 16px;
}

@media (max-width: 600px) {
  .tarjeta {
    max-width: 100%;
    padding: 12px;
  }
}

```

En pantallas chicas la tarjeta ocupa todo el ancho. Intentar hacer esto con style en línea y JavaScript (midiendo `window.innerWidth`) es reinventar la rueda y peor: el CSS lo hace declarativo y el navegador lo optimiza solo.

5. Combina los enfoques sin culpa, cada uno en su capa. Ejemplo de componente bien organizado:

```

import "../TarjetaPerfil.css";

function TarjetaPerfil({ destacada, children }) {
  // Capa 1: clases para el diseño base y variantes (CSS)
  const clase = `perfil ${destacada ? "perfil-destacada" : ""}`;

  // Capa 2: estilos en línea solo para lo dinámico (color que viene de datos)
  return (
    <div className={clase}>
      {children}
    </div>
  );
}

```

El diseño (bordes, sombras, layout, responsive) vive en `TarjetaPerfil.css`; la decisión de que variante toca vive en el JavaScript con el ternario; y si algún día un color viene de una API, ese valor puntual iría en `style`. Tres capas, cada una en su lugar, código fácil de mantener.

6. Evita el `!important` como la plaga. Si un estilo no se aplica, el problema casi siempre es especificidad o el orden de carga, no falta de fuerza bruta. El `!important` tapa el síntoma y crea un problema mayor después, cuando quieras sobrescribir ese estilo y ya nada funcione.

useRef: patrones extra en la vida real

Ya viste los dos usos principales (DOM directo y valores mutables). Aquí van tres patrones extra que aparecen constantemente en código profesional:

1. Guardar el valor anterior de una prop o estado. A veces necesitas comparar "que tenía antes" con "que tengo ahora". Como el estado siempre tiene el valor actual del render, el anterior lo guardas en una ref:

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

function ContadorConHistorial() {
  const [cuenta, setCuenta] = useState(0);
  const anteriorRef = useRef(0);

  useEffect(() => {
    anteriorRef.current = cuenta; // se actualiza despues de cada render
  });

  return (
    <div>
      <p>Actual: {cuenta}</p>
      <p>Anterior: {anteriorRef.current}</p>
      <button onClick={() => setCuenta(cuenta + 1)}>Sumar</button>
    </div>
  );
}
```

El efecto se ejecuta después de cada render y guarda el valor actual en la ref; así, durante el siguiente render, `anteriorRef.current` todavía tiene el valor del render pasado. Es la forma idiomática de "acordarse" del valor previo sin provocar renders extra.

2. Bandera para evitar trabajo repetido. Por ejemplo, evitar que una función de inicialización corra dos veces:

```
function Inicializador() {
  const yaInicioRef = useRef(false);

  function iniciar() {
    if (yaInicioRef.current) {
      return; // ya se hizo, no repetir
    }
    yaInicioRef.current = true;
    console.log("Inicializando una sola vez...");
  }

  return <button onClick={iniciar}>Iniciar</button>;
}
```

Una variable normal no serviría porque se reiniciaría en cada render; el estado provocaría un render innecesario solo para recordar un booleano interno. La ref es el lugar exacto.

3. Limpiar intervalos al desmontar. En el ejercicio del cronómetro el intervalo vive mientras el componente existe, pero si el componente puede desaparecer de la pantalla (por ejemplo, al cerrar un modal), debes limpiar el intervalo para que no siga corriendo en el fondo. El patrón completo usa `useEffect` con limpieza:

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

function Reloj() {
  const [segundos, setSegundos] = useState(0);
  const intervaloRef = useRef(null);

  useEffect(() => {
    intervaloRef.current = setInterval(() => {
      setSegundos((s) => s + 1);
    }, 1000);

    return () => {
      clearInterval(intervaloRef.current); // limpieza al desmontar
    };
  }, []);

  return <p>Segundos: {segundos}</p>;
}
```

El efecto crea el intervalo una sola vez (arreglo vacío `[]`) y devuelve una función de limpieza que React ejecuta cuando el componente desaparece. La ref guarda el id para que la limpieza

sepa que intervalo detener. Este es el patrón profesional completo: ref para el id, efecto para el ciclo de vida, estado para lo visible. Cada pieza en su lugar.

Depurando estilos: cuando no se ve como esperas

Tarde o temprano te va a pasar: escribes el CSS, escribes el className, y la pantalla se ve igual que antes. En lugar de frustrarte, sigue esta lista de verificación en orden, que resuelve el 95% de los casos:

1. Revisa className vs class. Es la causa número uno. Busca class= en tu JSX y cambialo por className=. React te avisa en la consola del navegador, así que abre las herramientas de desarrollo (F12) y lee los warnings: ahí suele estar la respuesta.
2. Verifica que importaste el CSS. Si olvidaste la línea import `"/Botón.css"` en el componente, las clases no existen para la página. El className queda puesto en el HTML pero sin estilos que lo respalden. Revisa la parte de arriba de tu archivo.
3. Inspecciona el elemento en el navegador. Clic derecho sobre el elemento, "Inspeccionar", y mira el panel de estilos: ahí ves que clases tiene aplicadas, que reglas le están llegando y cuáles están tachadas (las tachadas perdieron por especificidad u orden). Esta es la herramienta más poderosa para depurar estilos: te dice exactamente que está pasando.
4. Cuidado con el cache del navegador. A veces cambias el CSS y no ves el cambio porque el navegador guardó la versión vieja. Recarga forzada con Ctrl+Shift+R (o Cmd+Shift+R en Mac) y listo.
5. Revisa la especificidad y el orden. Si dos reglas pelean por el mismo elemento, gana la más específica, y en empate gana la que se cargó al último. Si tu clase no se aplica, quizá otra regla la está sobrescribiendo. En el inspector las reglas tachadas te muestran quien ganó la pelea.
6. Con estilos en línea, revisa el camelCase y las comillas. `background-color` dentro del objeto no funciona (debe ser `backgroundColor`), y un valor como `fontSize: 20px` sin comillas es un error de sintaxis (debe ser `"20px"` o `20`). Los errores de sintaxis en el objeto de estilos suelen romper todo el componente, así que la consola te avisará.
7. Prueba aislar el problema. Si un componente complejo no se ve bien, crea una versión mínima en otro archivo con solo ese elemento y su estilo. Si ahí funciona, el problema está en la interacción con el resto (herencia, contenedores padres con `overflow hidden`, etc.).

Un truco de depuración rápida: ponle temporalmente un borde rojo al elemento sospechoso con `style={{ border: "2px solid red" }}`. Si aparece el borde, el elemento existe y el problema es

tu CSS; si no aparece, el problema es que el elemento ni siquiera se esta dibujando (quizá un condicional o un error en el JSX). Es feo pero efectivo, y cuando lo resuelves quitas la línea.

Ejemplos paso a paso

Ejemplo 1: componente Tarjeta con children

Vamos a construir la tarjeta reutilizable clasica: un contenedor con borde, sombra y padding que puede envolver cualquier contenido.

```
function Tarjeta({ titulo, children }) {  
  return (  
    <div className="tarjeta">  
      <h3 className="tarjeta-titulo">{titulo}</h3>  
      <div className="tarjeta-cuerpo">  
        {children}  
      </div>  
    </div>  
  );  
}
```

Línea por línea: `function Tarjeta({ título, children })` declara el componente y desestructura dos props: `título` (un texto simple) y `children` (el contenido anidado, la prop especial). El `return` devuelve un `div` con clase `"tarjeta"`, que en el CSS tendrá el borde, la sombra y el padding. Dentro hay un `h3` que muestra el título, y un `div` con clase `"tarjeta-cuerpo"` donde se dibuja `{children}`: ahí aparece todo lo que el usuario del componente ponga entre `<Tarjeta>` y `</Tarjeta>`.

El CSS:

```

.tarjeta {
  border: 1px solid #ddd;
  border-radius: 12px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
  padding: 16px;
  max-width: 350px;
  background-color: white;
}

.tarjeta-titulo {
  margin-top: 0;
  color: #1a237e;
}

.tarjeta-cuerpo {
  color: #333;
}

```

Y el uso, que es donde se ve la magia:

```

function App() {
  return (
    <div>
      <Tarjeta titulo="Curso de React">
        <p>Aprende desde cero hasta profesional.</p>
        <p><strong>Precio:</strong> gratis</p>
        <button>Inscribirme</button>
      </Tarjeta>

      <Tarjeta titulo="Curso de Python">
        <p>El lenguaje mas pedido por las empresas.</p>
        <button>Inscribirme</button>
      </Tarjeta>
    </div>
  );
}

```

La misma Tarjeta envuelve contenidos distintos: la primera lleva dos párrafos y un botón, la segunda un párrafo y un botón. El título viaja por prop porque es un dato simple; el cuerpo viaja por children porque es estructura. Si mañana quieres que todas las tarjetas tengan el borde más grueso, cambias una línea de CSS y las dos se actualizan. Eso es reutilización real.

Ejemplo 2: modal reutilizable

Un modal es el ejemplo perfecto de children: el armazon (fondo oscuro, caja centrada, botón de cerrar) siempre es igual; el contenido cambia cada vez.

```
function Modal({ titulo, onCerrar, children }) {  
  return (  
    <div className="modal-fondo" onClick={onCerrar}>  
      <div  
        className="modal-caja"  
        onClick={(e) => e.stopPropagation()}  
      >  
        <div className="modal-encabezado">  
          <h2>{titulo}</h2>  
          <button className="modal-cerrar" onClick={onCerrar}>  
            X  
          </button>  
        </div>  
        <div className="modal-cuerpo">  
          {children}  
        </div>  
      </div>  
    </div>  
  );  
}
```

Línea por línea: el componente recibe título, onCerrar (función para cerrar) y children. El div exterior modal-fondo es la capa oscura que cubre toda la pantalla; tiene onClick={onCerrar} para que al hacer clic fuera de la caja se cierre el modal. El div interior modal-caja es la ventana blanca; su onClick hace e.stopPropagation() para que el clic dentro de la caja no se propague al fondo y no cierre el modal por accidente (detalle importante: sin esto, cualquier clic dentro cerraría la ventana). El encabezado muestra el título y un botón X que también cierra. El cuerpo dibuja {children}: cualquier contenido que le pases.

El CSS:

```
.modal-fondo {
  position: fixed;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background-color: rgba(0, 0, 0, 0.5);
  display: flex;
  justify-content: center;
  align-items: center;
}

.modal-caja {
  background-color: white;
  border-radius: 12px;
  padding: 24px;
  max-width: 450px;
  width: 90%;
}

.modal-encabezado {
  display: flex;
  justify-content: space-between;
  align-items: center;
}

.modal-cerrar {
  border: none;
  background: none;
  font-size: 18px;
  cursor: pointer;
}
```

Y se usa así, combinado con estado para abrir y cerrar:


```
function App() {
  const [abierto, setAbierto] = useState(false);

  return (
    <div>
      <button onClick={() => setAbierto(true)}>
        Ver detalles del curso
      </button>

      {abierto && (
        <Modal titulo="Curso de React" onClose={() => setAbierto(false)}>
          <p>30 clases, 100% gratis, en español.</p>
          <ul>
            <li>Componentes y props</li>
            <li>Estado con useState</li>
            <li>Children, estilos y useRef</li>
          </ul>
          <button onClick={() => setAbierto(false)}>
            Empezar ahora
          </button>
        </Modal>
      )}
    </div>
  );
}
```

El estado abierto controla si el modal existe; cuando es true se dibuja el Modal con su contenido. Fíjate como children permite que cada uso del modal tenga un cuerpo totalmente distinto sin cambiar el componente Modal. Este patrón lo vas a copiar tal cual en tus proyectos.

Ejemplo 3: botón con estilos dinámicos según prop

Ahora combinamos estilos dinámicos: un botón que cambia de apariencia según la prop variante.

```
function Boton({ variante, children, onClick }) {
  // variante puede ser "primario", "peligro" o nada (normal)
  const clase = `boton ${variante ? "boton-" + variante : ""}`;

  return (
    <button className={clase} onClick={onClick}>
      {children}
    </button>
  );
}
```

Línea por línea: el componente recibe variante, children y onClick. La constante clase usa un template string: siempre incluye "botón" (los estilos base) y, si variante tiene valor, agrega "botón-primario" o "botón-peligro". Si variante no se pasa, queda solo "botón" con un espacio extra que no afecta nada. El button usa ese className, muestra {children} como texto del botón y ejecuta onClick al presionarlo.

El CSS con las variantes:

```
.boton {
  padding: 10px 20px;
  border: none;
  border-radius: 8px;
  background-color: #e0e0e0;
  color: #333;
  cursor: pointer;
  font-size: 16px;
}

.boton:hover {
  opacity: 0.85;
}

.boton-primario {
  background-color: #1565c0;
  color: white;
}

.boton-peligro {
  background-color: #c62828;
  color: white;
}
```

Uso:

```
function App() {
  return (
    <div>
      <Boton>Normal</Boton>
      <Boton variante="primario" onClick={() => alert("Guardado")}>
        Guardar
      </Boton>
      <Boton variante="peligro" onClick={() => alert("Eliminado")}>
        Eliminar
      </Boton>
    </div>
  );
}
```

Tres botones, un solo componente, tres apariencias. La lógica de "que color toca" vive en el CSS (donde pertenece el diseño) y el componente solo decide que clase aplicar según la prop. Si mañana agregas la variante "éxito", solo agregas la clase .botón-éxito al CSS; el componente ya sabe construir "botón-" + variante sin cambios. Fíjate también que el texto del botón viaja por children, así cada botón dice lo que quiera.

Ejemplo 4: input con botón "enfocar" usando useRef

Cerramos con useRef en su caso más clásico: enfocar un campo de texto al presionar un botón.

```

import { useRef } from "react";

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

  function enfocarInput() {
    inputRef.current.focus();
  }

  function limpiarYEnfocar() {
    inputRef.current.value = "";
    inputRef.current.focus();
  }

  return (
    <div className="buscador">
      <input
        ref={inputRef}
        type="text"
        placeholder="Buscar cursos..."
      />
      <button onClick={enfocarInput}>Enfocar</button>
      <button onClick={limpiarYEnfocar}>Limpiar y enfocar</button>
    </div>
  );
}

```

Línea por línea: importamos `useRef` de React. Dentro del componente creamos `inputRef = useRef(null)`: la cajita que apuntará al input. La función `enfocarInput` llama `inputRef.current.focus()`: toma el elemento real del DOM y le pone el cursor dentro. La función `limpiarYEnfocar` hace dos cosas imperativas: vacía el campo (`inputRef.current.value = ""`) y luego lo enfoca. En el JSX, el input recibe `ref={inputRef}`: con eso React mete el elemento del DOM en `inputRef.current` cuando se dibuja. Los dos botones llaman a las funciones.

Por qué esto es un buen uso de `useRef`: enfocar un input no necesita redibujar nada; es una orden directa al navegador. Con estado sería torpe: tendrías que inventar un estado "enfocado" que no se dibuja en ningún lado solo para disparar un efecto. La ref es directa y honesta: quiero el elemento, lo tomé, le doy la orden.

Un detalle de buena práctica: aquí tocamos `inputRef.current.value` directamente. Para un buscador real probablemente controlarías el input con estado (`value + onChange`), y usarías la ref SOLO para el focus. La regla se mantiene: el texto visible es estado; el foco es ref. Mezclar ambas cosas en el mismo ejemplo está bien para aprender, pero en producción separa responsabilidades: estado para lo que se dibuja, ref para las ordenes al DOM.

Ejemplo 5: tarjeta de perfil que combina los tres temas

Para cerrar, un ejemplo que mezcla todo: una tarjeta de perfil con `children` para el contenido, estilos dinámicos según la prop `destacada`, y `useRef` para enfocar el campo de mensaje.

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

function TarjetaPerfil({ nombre, destacada, children }) {
  const inputRef = useRef(null);
  const clase = `perfil ${destacada ? "perfil-destacada" : ""}`;

  function enfocarMensaje() {
    inputRef.current.focus();
  }

  return (
    <div className={clase}>
      <h3>{nombre} {destacada && "(destacado)"}</h3>
      <div className="perfil-contenido">
        {children}
      </div>
      <input
        ref={inputRef}
        placeholder="Escribele un mensaje..."
      />
      <button onClick={enfocarMensaje}>
        Escribir mensaje
      </button>
    </div>
  );
}
```

Línea por línea: el componente recibe `nombre` (dato simple por prop), `destacada` (booleano que cambia la apariencia) y `children` (contenido libre). `inputRef` es la ref para el campo de mensaje. La clase se construye con template string: siempre "perfil" más "perfil-destacada" si aplica. En el JSX, el `h3` muestra el nombre y, con render condicional (`destacada &&`

"(destacado)"), una marca extra solo si es destacado. El div perfil-contenido dibuja {children}: ahí va la biografía, los cursos, lo que sea. El input recibe ref={inputRef} y el botón llama a enfocarMensaje, que pone el cursor en el campo.

El CSS:

```
.perfil {
  border: 1px solid #ddd;
  border-radius: 12px;
  padding: 16px;
  max-width: 350px;
  background-color: white;
}

.perfil-destacada {
  border: 2px solid #1565c0;
  box-shadow: 0 4px 16px rgba(21, 101, 192, 0.25);
}
```

Uso:

```
function App() {
  return (
    <TarjetaPerfil nombre="Esteban" destacada={true}>
      <p>Instructor de React y creador de la academia.</p>
      <p>22 cursos gratuitos.</p>
    </TarjetaPerfil>
  );
}
```

Fíjate como cada tema hace exactamente su trabajo: children recibe la biografía sin que la tarjeta sepa que dice; el estilo condicional marca al perfil destacado con borde azul y sombra; useRef le da el toque interactivo de enfocar el mensaje con un clic. Ninguna herramienta invade el terreno de la otra. Cuando escribas componentes así de forma natural, ya piensas como desarrollador React profesional.

Práctica

Ejercicio 1: layout con children para páginas

Enunciado: crea un componente Layout que muestre un encabezado con el nombre de la academia, un menu de navegación con tres enlaces y un pie de página. El contenido de cada página debe pasarse por children. Luego crea dos páginas (Inicio y Cursos) que usen el Layout con contenido distinto.

Solución:

```

function Layout({ children }) {
  return (
    <div className="layout">
      <header className="layout-header">
        <h1>Lucas EGA Academy</h1>
        <nav>
          <a href="#inicio">Inicio</a>
          <a href="#cursos">Cursos</a>
          <a href="#contacto">Contacto</a>
        </nav>
      </header>

      <main className="layout-contenido">
        {children}
      </main>

      <footer className="layout-pie">
        <p>Aprende tecnologia gratis - lucasegaacademy.com</p>
      </footer>
    </div>
  );
}

function PaginaInicio() {
  return (
    <Layout>
      <h2>Bienvenido a la academia</h2>
      <p>22 cursos gratuitos de tecnologia en espanol.</p>
    </Layout>
  );
}

function PaginaCursos() {
  return (
    <Layout>
      <h2>Nuestros cursos</h2>
      <ul>
        <li>HTML5 desde cero</li>
        <li>CSS moderno</li>
        <li>React profesional</li>
      </ul>
    </Layout>
  );
}

```


Explicación: Layout es un contenedor puro: no recibe el contenido por props de datos, sino por children. El header y el footer se escriben una sola vez; el main dibuja {children} donde cada página pone lo suyo. PaginaInicio y PaginaCursos solo se preocupan por su contenido. Si mañana agregas un enlace al menu, lo cambias en Layout y ambas páginas se actualizan. Este es el patrón que usan los frameworks de React para las plantillas de página.

Ejercicio 2: alerta reutilizable con tipos

Enunciado: crea un componente Alerta que reciba una prop tipo ("éxito", "error", "info") y children con el mensaje. Según el tipo, la alerta debe mostrarse de distinto color usando clases CSS condicionales.

Solución:

```
function Alerta({ tipo, children }) {
  const clase = `alerta alerta-${tipo}`;

  return (
    <div className={clase}>
      <span className="alerta-icone">
        {tipo === "exito" ? "OK " : tipo === "error" ? "X " : "i "}
      </span>
      {children}
    </div>
  );
}
```

CSS:

```

.alerta {
  padding: 12px 16px;
  border-radius: 8px;
  margin: 8px 0;
  font-size: 15px;
}

.alerta-exito {
  background-color: #e8f5e9;
  color: #2e7d32;
  border: 1px solid #2e7d32;
}

.alerta-error {
  background-color: #ffebee;
  color: #c62828;
  border: 1px solid #c62828;
}

.alerta-info {
  background-color: #e3f2fd;
  color: #1565c0;
  border: 1px solid #1565c0;
}

```

Uso:

```

function App() {
  return (
    <div>
      <Alerta tipo="exito">
        Tu cuenta se creo correctamente.
      </Alerta>
      <Alerta tipo="error">
        El correo ya esta registrado.
      </Alerta>
      <Alerta tipo="info">
        Tienes 3 cursos pendientes.
      </Alerta>
    </div>
  );
}

```

Explicación: el componente combina los dos temas: children para el mensaje (estructura libre) y estilos condicionales para la apariencia. La clase se construye con template string: ``alerta alerta-${tipo}`` produce "alerta alerta-éxito", "alerta alerta-error", etc. El icono también es condicional con ternarios anidados: según el tipo muestra un texto distinto. Nota que el diseño vive en el CSS y el componente solo decide que clase toca. Agregar un tipo nuevo ("advertencia") sería cuestión de agregar la clase `.alerta-advertencia`, sin tocar el JavaScript (salvo el icono si quieres uno específico).

Ejercicio 3: semaforo visual con estilos condicionales

Enunciado: crea un semaforo con tres focos (rojo, amarillo, verde). Solo el foco activo debe verse encendido con su color; los demás se ven apagados (gris oscuro). Tres botones permiten cambiar la luz activa. Usa el atributo `style` con objetos para los focos.

Solución:

```

import { useState } from "react";

function Semaforo() {
  const [luzActiva, setLuzActiva] = useState("rojo");

  function estiloFoco(color) {
    return {
      width: 70,
      height: 70,
      borderRadius: "50%",
      margin: "10px auto",
      backgroundColor: luzActiva === color ? color : "#3a3a3a",
      boxShadow: luzActiva === color ? `0 0 20px ${color}` : "none"
    };
  }

  const estiloCaja = {
    backgroundColor: "#222",
    padding: 20,
    borderRadius: 16,
    width: 110,
    margin: "0 auto"
  };

  return (
    <div>
      <div style={estiloCaja}>
        <div style={estiloFoco("red")}></div>
        <div style={estiloFoco("yellow")}></div>
        <div style={estiloFoco("green")}></div>
      </div>
      <div style={{ textAlign: "center", marginTop: 16 }}>
        <button onClick={() => setLuzActiva("red")}>Rojo</button>
        <button onClick={() => setLuzActiva("yellow")}>Amarillo</button>
        <button onClick={() => setLuzActiva("green")}>Verde</button>
      </div>
    </div>
  );
}

```

Explicación: el estado `luzActiva` guarda que luz esta prendida ("red", "yellow" o "green"). La función `estiloFoco(color)` devuelve un objeto de estilos distinto según si ese color es el activo: si lo es, `backgroundColor` es el color y se agrega un `boxShadow` brillante del mismo color para el efecto de luz encendida; si no, el foco queda en gris oscuro `#3a3a3a` sin sombra. Como el

objeto se recalcula en cada render, al cambiar el estado los focos se actualizan solos. Este ejercicio muestra por qué los estilos en línea brillan con valores dinámicos: el color y la sombra dependen directamente del estado, y expresarlo con clases requeriría nueve combinaciones de clases. Nota el camelCase (backgroundColor, boxShadow, borderRadius, marginTop) y los números sin "px" (width: 70 equivale a "70px").

Ejercicio 4: cronómetro con useRef para el intervalo

Enunciado: crea un cronómetro con botones Iniciar, Detener y Reiniciar que muestre los segundos transcurridos. El identificador del intervalo debe guardarse en una useRef (no en el estado), porque no es un dato visible.

Solución:

```

import { useState, useRef } from "react";

function Cronometro() {
  const [segundos, setSegundos] = useState(0);
  const [corriendo, setCorriendo] = useState(false);
  const intervaloRef = useRef(null);

  function iniciar() {
    if (intervaloRef.current !== null) {
      return; // ya hay un intervalo corriendo, no crear otro
    }
    intervaloRef.current = setInterval(() => {
      setSegundos((s) => s + 1);
    }, 1000);
    setCorriendo(true);
  }

  function detener() {
    clearInterval(intervaloRef.current);
    intervaloRef.current = null;
    setCorriendo(false);
  }

  function reiniciar() {
    detener();
    setSegundos(0);
  }

  return (
    <div className="cronometro">
      <p className="cronometro-tiempo">{segundos} s</p>
      <button onClick={iniciar} disabled={corriendo}>
        Iniciar
      </button>
      <button onClick={detener} disabled={!corriendo}>
        Detener
      </button>
      <button onClick={reiniciar}>Reiniciar</button>
    </div>
  );
}

```

Explicación: aquí se ve la división de trabajo entre estado y ref con total claridad. segundos y corriendo van en useState porque se dibujan en pantalla (el número y el disabled de los

botones). El id del intervalo va en `intervaloRef` porque es un dato interno: solo sirve para detener el intervalo después, nadie necesita verlo. Si lo guardaras en el estado, iniciar y detener provocarían renders extra sin ningún beneficio visual. El chequeo `if (intervaloRef.current !== null)` evita crear dos intervalos si el usuario presiona Iniciar dos veces. `detener()` limpia el intervalo con `clearInterval` y deja la ref en null para indicar que no hay nada corriendo. `reiniciar()` reutiliza `detener()` y pone los segundos en cero: buen ejemplo de no repetir lógica.

Ejercicio 5: galeria con modal al hacer clic

Enunciado: crea una galeria de tres imagenes (puedes usar divs de colores como imagenes de ejemplo). Al hacer clic en una imagen, se abre un modal reutilizable (con children) que muestra la imagen en grande con su título. El modal se cierra con un botón o haciendo clic fuera.

Solución:

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

function Modal({ titulo, onCerrar, children }) {
  return (
    <div className="modal-fondo" onClick={onCerrar}>
      <div className="modal-caja" onClick={(e) => e.stopPropagation()}>
        <div className="modal-encabezado">
          <h2>{titulo}</h2>
          <button onClick={onCerrar}>X</button>
        </div>
        <div className="modal-cuerpo">{children}</div>
      </div>
    </div>
  );
}

const IMAGENES = [
  { id: 1, titulo: "Amanecer", color: "#ff8a65" },
  { id: 2, titulo: "Oceano", color: "#4fc3f7" },
  { id: 3, titulo: "Bosque", color: "#81c784" }
];

function Galeria() {
  const [seleccionada, setSeleccionada] = useState(null);

  return (
    <div>
      <h2>Mi galeria</h2>
```

```

<div className="galeria-grid">
  {IMAGENES.map((img) => (
    <div
      key={img.id}
      className="galeria-mini"
      style={{ backgroundColor: img.color }}
      onClick={() => setSeleccionada(img)}
    >
      {img.titulo}
    </div>
  ))}
</div>

{seleccionada && (
  <Modal
    titulo={seleccionada.titulo}
    onClose={() => setSeleccionada(null)}
  >
    <div
      className="galeria-grande"
      style={{ backgroundColor: seleccionada.color }}
    >
      {seleccionada.titulo}
    </div>
    <p>Imagen numero {seleccionada.id} de la coleccion.</p>
    <button onClick={() => setSeleccionada(null)}>
      Cerrar
    </button>
  </Modal>
)}
</div>
);
}

```

CSS mínimo:


```

.galeria-grid {
  display: flex;
  gap: 12px;
}

.galeria-mini {
  width: 120px;
  height: 90px;
  border-radius: 8px;
  display: flex;
  align-items: center;
  justify-content: center;
  color: white;
  font-weight: bold;
  cursor: pointer;
}

.galeria-grande {
  height: 220px;
  border-radius: 8px;
  display: flex;
  align-items: center;
  justify-content: center;
  color: white;
  font-size: 24px;
  font-weight: bold;
}

```

Explicación: este ejercicio junta los tres temas de la unidad en una mini aplicación real. El estado seleccionada guarda la imagen elegida (null = modal cerrado): es estado porque controla lo que se dibuja. La cuadrícula usa map para generar las miniaturas, cada una con su color de fondo aplicado con style en línea porque el color viene de los datos (estilo dinámico). Al hacer clic, setSeleccionada(img) abre el modal. El Modal reutilizable recibe el título por prop y el contenido grande por children: la imagen ampliada, una descripción y el botón de cerrar. El modal se cierra de tres formas: botón X, botón Cerrar, o clic en el fondo oscuro (gracias al stopPropagation en la caja). Fíjate como cada tema hace su parte sin estorbar a los demás: children para el contenido variable, estilos dinámicos para los colores de cada imagen, y estado para controlar la visibilidad. Si quisieras agregar autoenfoco al abrir el modal, ahí entraría useRef con inputRef.current.focus(): los tres temas trabajando juntos, como en una app de verdad.

Resumen de la unidad

En esta unidad aprendiste tres herramientas que llevan tus componentes de "funcionan" a "están bien contruidos": children, los estilos en React y useRef.

Children es la prop especial que React llena automáticamente con lo que pongas entre las etiquetas de apertura y cierre de un componente. Te permite crear componentes contenedores (wrappers) que envuelven cualquier contenido: tarjetas, modales, layouts de página, alertas y botoneras. El contenedor no necesita saber que hay adentro; solo se encarga de la estructura y la presentación. Aprendiste que children puede ser texto, elementos, varios elementos, otros componentes o mezclas de todo, y que la composición con children es una alternativa elegante al props drilling cuando el contenido es estructura y no datos simples. La combinación ganadora: datos simples por props, estructura variable por children.

En estilos, aprendiste las dos formas principales. Con className (nunca class en JSX) aplicas clases de archivos CSS normales, que importas en el componente; es la forma más común y te da toda la potencia del CSS: pseudoclasas, media queries y animaciones. Con el atributo style pasas un objeto de JavaScript usando la sintaxis de doble llave `{{ }}`, con propiedades en camelCase (backgroundColor en vez de background-color) y valores como strings o números (los números se convierten a pixeles automáticamente). Dominaste los estilos condicionales y dinámicos con ternarios y template strings, que hacen que la apariencia reaccione al estado: botones que cambian de color según una prop, semáforos que prenden la luz activa. Conociste la idea de CSS Modules (archivos .module.css con clases locales que no chocan entre componentes) y el criterio para elegir cada enfoque: CSS con className para el diseño base, estilos en línea para lo dinámico y puntual, variables CSS en :root para centralizar los colores de tu marca.

Con useRef aprendiste a guardar una referencia mutable que sobrevive entre renders sin provocar re-renders. Sus dos usos legítimos: acceso directo al DOM (crear la ref con useRef(null), pasarla con el atributo ref, y usar ref.current para llamar métodos como .focus(), medir con offsetWidth o hacer scroll) y guardar valores mutables internos como el id de un intervalo. La tabla comparativa te dejó clara la diferencia con useState: el estado es para lo que se dibuja (cambiarlo re-renderiza), la ref es para lo que no (cambiar .current es silencioso). La regla de oro: estado para lo que se dibuja, ref para lo que no; useRef es la salida de emergencia, no la puerta principal.

También te llevaste las analogías para no olvidar (la caja de regalo, el marco de fotos, la nota adhesiva secreta), la lista de errores comunes (usar `class` en vez de `className`, abusar de `useRef` para lo visible, estilos en línea gigantes, manipular el DOM a mano para lo que React maneja solo) y las buenas prácticas (extraer contenedores antes del tercer copiado, prefijos en las clases o CSS Modules, variables CSS centralizadas, verificar la ref antes de usarla).

Los cuatro ejemplos paso a paso (Tarjeta, Modal, Botón con variantes y Buscador con foco) y los cinco ejercicios resueltos (layout, alerta con tipos, semaforo, cronómetro y galería con modal) te mostraron cómo se combinan los tres temas en componentes reales. El patrón final que debes dominar: un contenedor con `children`, vestido con estilos condicionales, que usa `useRef` solo para esos toques directos al DOM que el estado no puede dar.

En la próxima unidad seguiras construyendo sobre esta base. Por ahora, práctica: toma uno de tus componentes viejos, conviértelo en contenedor con `children`, dale variantes de estilo y, si necesita enfocar algo o guardar un intervalo, dale su `useRef`. Esa es la diferencia entre código que funciona y código profesional.

UNIDAD 8: Contexto, hooks personalizados y rutas

Bienvenido a la unidad 8. Si llegaste hasta aquí, ya sabes montar componentes, pasar props, manejar estado con `useState` y traer datos con `useEffect`. Eres capaz de construir una app pequeña de una sola pantalla sin problemas. Pero hay un momento en la vida de todo desarrollador React en el que la app crece: de 3 componentes pasa a 10, de 10 pasa a 30, y de pronto te das cuenta de que las técnicas que te sirvieron al principio ya no te alcanzan. El estado que antes vivía tranquilo en un componente ahora lo necesitan cinco componentes lejanos entre sí. La lógica que repetiste dos veces ahora la repites en ocho archivos. Y tu app de una sola pantalla necesita convertirse en un sitio con varias páginas: inicio, perfil, contacto, cada una con su propia URL.

Esta unidad es el puente entre el React de juguete y el React profesional. Aquí vas a aprender las tres herramientas que todo desarrollador React usa a diario en aplicaciones reales: `useContext`, que te permite compartir datos entre componentes sin pasar props manualmente por cada nivel del árbol; los custom hooks, que te permiten extraer lógica con estado en funciones reutilizables; y React Router, que te permite construir aplicaciones con varias páginas navegables sin recargar el navegador.

Piensa en una app real, por ejemplo una tienda en línea. El usuario inicia sesión en una página y su nombre debe aparecer en el encabezado de TODAS las páginas: inicio, catálogo, carrito, perfil. Con lo que sabes hasta ahora, tendrías que pasar la prop "usuario" desde el componente raíz hasta el encabezado, atravesando todos los componentes intermedios que no la necesitan ni la usan. Eso se llama props drilling, y ya lo viste en unidades anteriores. Funciona con 2 o 3 niveles, pero con 8 niveles se vuelve una pesadilla de mantenimiento: cambias el nombre de una prop y tienes que tocar diez archivos.

Hay una historia que se repite en todos los equipos: el proyecto empieza con tres componentes y todo es felicidad. Las props viajan del padre al hijo sin problema. Un mes después hay veinte componentes y alguien necesita el idioma seleccionado en un rincón profundo del árbol. El desarrollador, con prisa, pasa la prop por cinco niveles "nada más por esta vez". Otro mes después hay que agregar el tema oscuro, y otra prop atraviesa los mismos cinco niveles. Al tercer mes, cambiar cualquier cosa rompe otra: los componentes intermedios reciben doce props que no usan, nadie se acuerda cuál va a dónde, y agregar una pantalla nueva toma días en lugar de horas. Esa historia tiene nombre: deuda de arquitectura. Y se evita con las herramientas de esta unidad, aplicadas a tiempo.

Además, esa misma tienda tiene un tema claro/oscuro. El interruptor vive en el encabezado, pero el cambio de colores debe aplicarse a toda la app. También necesita que el carrito de compras esté disponible en el catálogo (para agregar productos) y en la página del carrito (para verlos y pagar). Todo esto son "datos globales": información que muchos componentes necesitan sin importar dónde estén en el árbol. Para eso existe el contexto.

Ahora piensa en la lógica repetida. En tu app tienes un contador en la página de inicio, otro en el perfil y otro en el carrito. Cada uno con su `useState`, su función de incrementar y su función de decrementar. Copiar y pegar tres veces funciona, pero es frágil: si un día quieres agregar un límite máximo, tienes que acordarte de cambiarlo en los tres lugares. Los custom hooks resuelven esto: extraes la lógica del contador a una función `useContador()` y la usas en los tres componentes. Una sola fuente de verdad, cero duplicación.

Y finalmente las páginas. Hasta ahora tus apps son de una sola vista: todo se muestra al mismo tiempo o se alterna con un estado booleano. Pero los usuarios esperan URLs: quieren compartir el enlace de un producto, usar el botón "atrás" del navegador, marcar una página en favoritos. React Router te da eso: defines rutas como `"/inicio"`, `"/productos"`, `"/contacto"`, y React muestra el componente correcto según la URL, sin recargar la página. La navegación se siente instantánea porque solo cambia lo necesario.

Estas tres herramientas se combinan en la práctica. Una app profesional típica usa React Router para las páginas, `useContext` para los datos globales (usuario, tema, carrito) y custom hooks para la lógica reutilizable (formularios, llamadas a APIs, almacenamiento local). Al terminar esta unidad vas a poder construir exactamente ese tipo de app: un mini sitio completo con navegación, tema claro/oscuro, datos compartidos y lógica reutilizable. Vamos paso a paso, como siempre: primero la explicación detallada de cada herramienta, luego ejemplos completos línea por línea, después práctica con ejercicios resueltos, y al final el resumen.

Explicación detallada

`useContext`: comparte datos sin props drilling

El problema que ya conoces: props drilling

Antes de entender la solución, recordemos el problema. Imagina este árbol de componentes:

App -> Encabezado -> BarraNavegacion -> MenuUsuario -> Avatar

El componente App tiene el dato del usuario logueado. El componente Avatar, cuatro niveles abajo, necesita mostrar la foto del usuario. Con props, el camino es así:

```
function App() {
  const usuario = { nombre: "Ana", foto: "ana.jpg" };
  return <Encabezado usuario={usuario} />;
}

function Encabezado({ usuario }) {
  return <BarraNavegacion usuario={usuario} />;
}

function BarraNavegacion({ usuario }) {
  return <MenuUsuario usuario={usuario} />;
}

function MenuUsuario({ usuario }) {
  return <Avatar usuario={usuario} />;
}

function Avatar({ usuario }) {
  return <img src={usuario.foto} alt={usuario.nombre} />;
}
```

Funciona, pero fíjate en el costó: Encabezado, BarraNavegacion y MenuUsuario reciben la prop "usuario" solo para pasarla al siguiente. No la usan para nada. Si mañana el diseñador pide que el pie de página también muestre el nombre del usuario, tienes que abrir toda la cadena otra vez y agregar la prop en cada nivel. A esto se le llama props drilling: taladrar props a través de niveles que no las necesitan. Con pocos niveles es tolerable; con muchos niveles es una fuente constante de errores y refactorizaciones dolorosas.

El contexto es la respuesta de React a este problema. En lugar de pasar el dato de mano en mano, lo pones en un "contenedor" especial al que cualquier componente descendiente puede acceder directamente, sin intermediarios.

createContext: crear el contenedor

El primer paso es crear el contexto con la función createContext. Piensa en el contexto como una caja etiquetada donde guardas un valor que muchos componentes van a necesitar:

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

const TemaContext = createContext("claro");
```

createContext recibe un valor por defecto, en este caso la cadena "claro". Ese valor por defecto solo se usa si un componente intenta leer el contexto sin que haya un Provider arriba

de él en el árbol (ya veremos qué es el Provider). En la práctica casi siempre vas a tener un Provider, así que el valor por defecto es más un respaldo que otra cosa.

Una convención importante: el nombre de la variable del contexto suele terminar en "Context", como TemaContext o UsuarioContext. No es obligatorio, pero todo el mundo lo hace así y te ayuda a reconocer de un vistazo qué es cada cosa.

Otro detalle crucial: el contexto se crea UNA vez, fuera de los componentes, normalmente en su propio archivo. Nunca lo crees dentro de un componente, porque entonces se crearía un contexto nuevo en cada render y nada funcionaría como esperas. Este es uno de los errores más comunes y lo veremos en la sección de errores.

Provider: el que reparte el valor

Crear el contexto no basta; necesitas un componente Provider que envuelva a todos los componentes que deben recibir el valor. El Provider se obtiene del propio contexto: TemaContext.Provider. Recibe una prop especial llamada "value" con el dato que quieres compartir:

```
function App() {
  const [tema, setTema] = useState("claro");

  return (
    <TemaContext.Provider value={tema}>
      <Encabezado />
      <Contenido />
      <Pie />
    </TemaContext.Provider>
  );
}
```

Todo lo que esté dentro de <TemaContext.Provider> puede leer el valor "tema" directamente, sin importar a cuántos niveles de profundidad esté. Encabezado, Contenido, Pie y todos sus hijos, nietos y bisnietos tienen acceso. Es como poner una bocina en la plaza: todos los que estén dentro la escuchan, sin necesidad de pasar el mensaje de persona en persona.

Y aquí viene lo poderoso: si el valor del Provider cambia (por ejemplo, el usuario cambia de "claro" a "oscuro"), todos los componentes que leen ese contexto se vuelven a renderizar automáticamente con el nuevo valor. No tienes que avisarle a nadie; React se encarga.

useContext: leer el valor donde lo necesites

Para leer el valor del contexto dentro de un componente usas el hook useContext. Le pasas el contexto que creaste y te devuelve el valor actual:

```
import { useContext } from "react";
import { TemaContext } from "../TemaContext";

function Encabezado() {
  const tema = useContext(TemaContext);

  return (
    <header className={tema === "oscuro" ? "header-oscuro" : "header-claro"}>
      <h1>Mi sitio</h1>
    </header>
  );
}
```

Fíjate en lo limpio que quedó: Encabezado ya no recibe ninguna prop. Simplemente pregunta "¿cuál es el tema actual?" y el contexto le responde. Los componentes intermedios (BarraNavegacion, MenuUsuario) desaparecieron de la cadena por completo; ya no tienen que saber que el dato existe.

Compara con la versión de props drilling de antes: eliminamos tres niveles de paso de props. En una app real con 8 o 10 niveles, el ahorro es enorme.

Ejemplo completo: tema claro/oscuro

Vamos a armar el ejemplo clásico paso a paso. Primero, el archivo del contexto:

```
// TemaContext.js
import { createContext } from "react";

export const TemaContext = createContext("claro");
```

Segundo, el componente App que provee el valor y permite cambiarlo:


```
// App.js
import { useState } from "react";
import { TemaContext } from "../TemaContext";
import Encabezado from "../Encabezado";
import Contenido from "../Contenido";

function App() {
  const [tema, setTema] = useState("claro");

  function alternarTema() {
    setTema(tema === "claro" ? "oscuro" : "claro");
  }

  return (
    <TemaContext.Provider value={tema}>
      <Encabezado />
      <Contenido />
      <button onClick={alternarTema}>
        Cambiar a tema {tema === "claro" ? "oscuro" : "claro"}
      </button>
    </TemaContext.Provider>
  );
}
```

Tercero, los componentes que consumen el contexto:

```
// Encabezado.js
import { useContext } from "react";
import { TemaContext } from "../TemaContext";

function Encabezado() {
  const tema = useContext(TemaContext);
  return (
    <header style={{ background: tema === "oscuro" ? "#222" : "#eee" }}>
      <h1>Mi app con contexto</h1>
    </header>
  );
}
```

```
// Contenido.js
import { useContext } from "react";
import { TemaContext } from "../TemaContext";

function Contenido() {
  const tema = useContext(TemaContext);
  return (
    <main style={{ color: tema === "oscuro" ? "#fff" : "#000" }}>
      <p>El tema actual es: {tema}</p>
    </main>
  );
}
```

Cuando el usuario hace clic en el botón, `setTema` cambia el estado en `App`, el `Provider` recibe el nuevo valor y `React` vuelve a renderizar `Encabezado` y `Contenido` con el tema actualizado. Todo sin pasar una sola prop manualmente.

Ejemplo: usuario logueado

Otro caso clásico es la sesión del usuario. Muchas partes de la app necesitan saber quién está logueado: el encabezado muestra su nombre, el perfil muestra sus datos, el carrito asocia la compra a su cuenta:

```
// UsuarioContext.js
import { createContext } from "react";

export const UsuarioContext = createContext(null);
```

```
// App.js
import { useState } from "react";
import { UsuarioContext } from "../UsuarioContext";
import Encabezado from "../Encabezado";
import Perfil from "../Perfil";

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

  function iniciarSesion() {
    setUsuario({ nombre: "Ana", email: "ana@ejemplo.com" });
  }

  return (
    <UsuarioContext.Provider value={{ usuario, setUsuario }}>
      <Encabezado />
      <Perfil />
      {usuario === null && (
        <button onClick={iniciarSesion}>Iniciar sesión</button>
      )}
    </UsuarioContext.Provider>
  );
}
```

Fíjate en un detalle importante: el value del Provider puede ser un objeto con varios valores, aquí pasamos tanto "usuario" como "setUsuario". Así, cualquier componente descendiente no solo puede leer quién está logueado, sino también cambiarlo (por ejemplo, un botón de "cerrar sesión" en el encabezado):

```
// Encabezado.js
import { useContext } from "react";
import { UsuarioContext } from "../UsuarioContext";

function Encabezado() {
  const { usuario, setUsuario } = useContext(UsuarioContext);

  function cerrarSesion() {
    setUsuario(null);
  }

  return (
    <header>
      {usuario ? (
        <div>
          <span>Hola, {usuario.nombre}</span>
          <button onClick={cerrarSesion}>Salir</button>
        </div>
      ) : (
        <span>No has iniciado sesión</span>
      )}
    </header>
  );
}
```

Este patrón (pasar el estado y su función actualizadora juntos en el value) es muy común en apps reales. El contexto se convierte en una mini tienda de datos compartidos.

Múltiples contextos: puedes anidar Providers

Una app real necesita varios datos globales a la vez: el tema, el usuario, el idioma, el carrito. No tienes que meter todo en un solo contexto gigante; lo profesional es crear un contexto por cada dominio y anidar los Providers:

```
function App() {
  return (
    <TemaContext.Provider value={tema}>
      <UsuarioContext.Provider value={usuario}>
        <IdiomaContext.Provider value={idioma}>
          <Encabezado />
          <Contenido />
        </IdiomaContext.Provider>
      </UsuarioContext.Provider>
    </TemaContext.Provider>
  );
}
```

Cada componente lee solo los contextos que necesita. Encabezado podría usar los tres; un botón simple podría no usar ninguno. Esta separación por dominio es una buena práctica: si el manejo del usuario cambia, solo tocas UsuarioContext sin afectar el tema ni el idioma.

Cuándo Sí y cuándo NO usar context

El contexto es poderoso, pero no es la respuesta para todo. Úsalo cuando:

- El dato lo necesitan muchos componentes en diferentes niveles del árbol (tema, usuario, idioma).
- El dato cambia con poca frecuencia (el tema cambia cuando el usuario lo alterna; el usuario cambia al iniciar o cerrar sesión).
- Quieres evitar props drilling real, de 3 o más niveles.

NO lo uses cuando:

- Solo un par de componentes cercanos necesitan el dato. Pasar props entre padre e hijo directo es más simple y más claro. No uses un cañón para matar una mosca.
- El dato cambia muy seguido, como el valor de un input que se actualiza con cada tecla. Cada cambio del value del Provider vuelve a renderizar a TODOS los componentes que consumen ese contexto, lo que puede volver lenta tu app si el dato cambia decenas de veces por segundo.
- Quieres compartir estado entre componentes que no tienen relación padre-hijo en el árbol. El contexto solo llega hacia abajo, a los descendientes del Provider. Si dos componentes están en ramas separadas sin un ancestro común que los envuelva, el contexto no te sirve directamente.

Una regla práctica que usan los profesionales: primero props, y solo cuando las props se vuelvan dolorosas (drilling de varios niveles), migra a contexto. El contexto es una herramienta de escape, no el default.

Errores comunes con useContext

Error 1: olvidar el Provider. Si lees un contexto con useContext pero ningún Provider envuelve tu componente, recibes el valor por defecto de createContext. Tu app no truena, pero muestra datos equivocados o vacíos, y es difícil detectar porque no hay mensaje de error. Siempre verifica que el Provider esté arriba en el árbol.

Error 2: crear el contexto dentro del componente. Esto es un error grave:

```
// MAL: el contexto se recrea en cada render
function App() {
  const MiContexto = createContext("hola");
  return (
    <MiContexto.Provider value="hola">
      <Hijo />
    </MiContexto.Provider>
  );
}
```

Cada vez que App se renderiza, createContext genera un contexto distinto, y los componentes hijos quedan suscritos al contexto viejo. El resultado: los valores no se actualizan o la app se comporta de forma errática. La solución es simple: crea el contexto fuera del componente, a nivel de módulo, o en su propio archivo.

Error 3: poner objetos nuevos en value sin necesidad. Recuerda que en el ejemplo del usuario pasamos value={{ usuario, setUsuario }}. Ese objeto literal se crea nuevo en cada render de App, lo que hace que todos los consumidores se vuelvan a renderizar aunque "usuario" no haya cambiado. En apps pequeñas no importa, pero en apps grandes se optimiza con useMemo. No te preocupes por esto ahora; solo tenlo presente para cuando tus apps crezcan.

Patrón profesional: el proveedor personalizado

Hasta ahora hemos puesto el estado directamente en App y pasado el Provider ahí mismo. Funciona, pero cuando la lógica crece (un carrito con agregar, quitar, vaciar y calcular total), App se llena de código que no le corresponde. El patrón profesional es crear un componente proveedor personalizado que encapsule todo: el estado, las funciones y el Provider. Mira cómo se vería para el carrito:

```

// CarritoContext.js
import { createContext, useState, useContext } from "react";

const CarritoContext = createContext(null);

export function CarritoProvider({ children }) {
  const [carrito, setCarrito] = useState([]);

  function agregarProducto(producto) {
    setCarrito([...carrito, producto]);
  }

  function quitarProducto(id) {
    setCarrito(carrito.filter((p) => p.id !== id));
  }

  function vaciarCarrito() {
    setCarrito([]);
  }

  const total = carrito.reduce((suma, p) => suma + p.precio, 0);

  return (
    <CarritoContext.Provider
      value={{ carrito, agregarProducto, quitarProducto, vaciarCarrito, total }}
    >
      {children}
    </CarritoContext.Provider>
  );
}

export function useCarrito() {
  const contexto = useContext(CarritoContext);
  if (contexto === null) {
    throw new Error("useCarrito debe usarse dentro de CarritoProvider");
  }
  return contexto;
}

```

Línea por línea: creamos el contexto sin exportarlo directamente (nadie necesita tocarlo a mano). CarritoProvider es un componente normal que recibe "children" (todo lo que pongas dentro de él). Dentro vive el estado del carrito y todas las funciones que lo manipulan: agregar, quitar por id, vaciar. El total se calcula con reduce. El Provider reparte un objeto con todo lo necesario y renderiza {children} dentro, es decir, a todos los componentes envueltos.

Finalmente, useCarrito es un custom hook que lee el contexto y, si alguien lo usa fuera del Provider, lanza un error claro en lugar de fallar en silencio. Ese error explícito te ahorra horas de depuración: en vez de preguntarte por qué el carrito sale vacío, el mensaje te dice exactamente qué olvidaste.

El uso en App queda limpio:

```
function App() {
  return (
    <CarritoProvider>
      <Encabezado />
      <Tienda />
      <Pie />
    </CarritoProvider>
  );
}
```

Y cualquier componente usa el hook:

```
function BotonAgregar({ producto }) {
  const { agregarProducto } = useCarrito();
  return <button onClick={() => agregarProducto(producto)}>Agregar</button>;
}
```

Este patrón (contexto + provider personalizado + hook de lectura con validación) es el estándar en la industria. Cuando veas librerías o código de empresas, lo vas a reconocer de inmediato. Úsalo siempre que un contexto tenga más de un par de líneas de lógica.

Custom hooks: tus propios hooks reutilizables

Qué es un custom hook

Un custom hook es una función de JavaScript cuyo nombre empieza con "use" y que puede llamar a otros hooks (useState, useEffect, useContext, etc.). Su propósito es extraer lógica con estado de un componente para reutilizarla en varios componentes.

La idea clave: los hooks que ya conoces (useState, useEffect) son funciones que encapsulan lógica. useState encapsula "guardar un valor y avisar cuando cambia"; useEffect encapsula "hacer algo cuando el componente aparece o cuando algo cambia". Un custom hook hace lo mismo pero con TU lógica: encapsula el comportamiento que se repite en tu app.

El ejemplo más simple del mundo:


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

function useContador() {
  const [n, setN] = useState(0);

  function incrementar() {
    setN(n + 1);
  }

  function decrementar() {
    setN(n - 1);
  }

  return [n, incrementar, decrementar];
}
```

Y se usa así en cualquier componente:

```
function ContadorA() {
  const [n, incrementar, decrementar] = useContador();
  return (
    <div>
      <p>Contador A: {n}</p>
      <button onClick={incrementar}>+</button>
      <button onClick={decrementar}>-</button>
    </div>
  );
}

function ContadorB() {
  const [n, incrementar, decrementar] = useContador();
  return (
    <div>
      <p>Contador B: {n}</p>
      <button onClick={incrementar}>+</button>
      <button onClick={decrementar}>-</button>
    </div>
  );
}
```

Cada componente que llama a useContador obtiene su PROPIO estado independiente. ContadorA y ContadorB no comparten el número; cada uno tiene su "n". Esto es fundamental: un custom hook no comparte estado entre componentes, comparte LÓGICA. Cada llamada al

hook crea una instancia fresca del estado. Si quieres estado compartido entre componentes, eso es trabajo del contexto (o de un manejador de estado), no de los custom hooks.

La regla del nombre: debe empezar con "use"

React identifica los hooks por su nombre. Si tu función se llama useContador, React sabe que es un hook y le aplica las reglas de los hooks (solo llamarlo en el nivel superior del componente, solo en componentes o en otros hooks). Si la llamas "contador" o "crearContador", React la trata como una función normal y las llamadas a useState dentro de ella van a fallar o comportarse mal, además de que las herramientas de desarrollo no podrán advertirte de errores.

No es una sugerencia de estilo; es un requisito técnico. Todo custom hook empieza con "use", sin excepciones: useFetch, useLocalStorage, useToggle, useFormulario, useVentana.

Extraer lógica paso a paso: de componente a hook

La forma más natural de crear un custom hook es extraerlo de un componente que ya funciona. Mira este componente que trae datos de una API:

```

function ListaUsuarios() {
  const [datos, setDatos] = useState(null);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("https://api.ejemplo.com/usuarios")
      .then((res) => res.json())
      .then((json) => {
        setDatos(json);
        setCargando(false);
      })
      .catch((err) => {
        setError(err.message);
        setCargando(false);
      });
  }, []);

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

```

Ahora necesitas la misma lógica en otro componente que trae productos. En lugar de copiar y pegar, extraes la parte de la lógica (todo menos el JSX) a un hook:

```
// useFetch.js
import { useState, useEffect } from "react";

function useFetch(url) {
  const [datos, setDatos] = useState(null);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

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

  return [datos, cargando, error];
}

export default useFetch;
```

Fíjate en los cambios: la URL ahora es un parámetro del hook, el `useEffect` depende de `[url]` para volver a traer datos si la URL cambia, y el hook devuelve los tres valores en un arreglo. Los componentes quedan así de limpios:

```
function ListaUsuarios() {
  const [datos, cargando, error] = useFetch("https://api.ejemplo.com/usuarios");

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

function ListaProductos() {
  const [datos, cargando, error] = useFetch("https://api.ejemplo.com/productos");

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

La lógica de fetch, cargando y error vive en un solo lugar. Si mañana quieres agregar reintentos automáticos o un timeout, lo cambias una vez en useFetch y todos los componentes se benefician.

Hooks que usan otros hooks

Un custom hook puede llamar a otros custom hooks, y también a useContext. Esto permite componer comportamientos. Por ejemplo, un hook useTema que simplifica la lectura del contexto del tema:

```
// useTema.js
import { useContext } from "react";
import { TemaContext } from "../TemaContext";

function useTema() {
  const tema = useContext(TemaContext);
  return tema;
}
```

Parece trivial, pero tiene ventajas reales: si un día cambias cómo funciona el tema (por ejemplo, agregas validación o un valor por defecto distinto), solo tocas useTema. Además, los componentes quedan más legibles: `const tema = useTema()` dice más que `const tema = useContext(TemaContext)`.

Otro ejemplo de composición: un hook `useContadorConLimite` que usa la lógica de `useContador` y le agrega un tope máximo:

```
function useContadorConLimite(maximo) {  
  const [n, incrementar, decrementar] = useContador();  
  
  function incrementarConLimite() {  
    if (n < maximo) {  
      incrementar();  
    }  
  }  
  
  return [n, incrementarConLimite, decrementar];  
}
```

Los hooks se apilan como bloques de construcción. Esta composición es una de las razones por las que el ecosistema React es tan productivo.

Librerías de hooks: no reinventes la rueda

Muchos custom hooks útiles ya existen en librerías open source. Las más conocidas son `react-use` y `ahooks`, con decenas de hooks listos: `useLocalStorage`, `useDebounce`, `useMedia` (para saber el tamaño de la pantalla), `usePrevious`, y muchos más. Cuando necesites un comportamiento común, busca primero si ya existe un hook probado antes de escribir el tuyo. Eso sí: para aprender, escribir tus propios hooks es el mejor ejercicio; para producción, una librería bien mantenida te ahorra bugs.

Buenas prácticas con custom hooks

Guarda cada hook en su propio archivo dentro de una carpeta `hooks/`. Por ejemplo: `src/hooks/useFetch.js`, `src/hooks/useContador.js`. El nombre del archivo coincide con el nombre del hook. Así, cuando tu proyecto crezca, cualquiera puede encontrar la lógica reutilizable en un solo lugar.

Un hook debe hacer una sola cosa y hacerla bien. `useFetch` trae datos; no debe también manejar el tema o validar formularios. Si un hook crece demasiado, divídelo en hooks más pequeños y compónlos.

Devuelve valores de forma consistente. Si devuelves un arreglo como [datos, cargando, error], documentas el orden. Si devuelves muchos valores, considera devolver un objeto { datos, cargando, error } para que quien lo use no dependa del orden. Ambas formas son válidas; lo importante es ser consistente en tu proyecto.

Errores comunes con custom hooks

Error 1: el hook no empieza con "use". Ya lo dijimos, pero vale repetirlo porque es el error número uno de los principiantes. `function contador() { const [n, setN] = useState(0); ... }` va a dar problemas. Siempre "use" al inicio.

Error 2: creer que el estado se comparte. Si dos componentes usan `useContador`, cada uno tiene su propio contador independiente. Si necesitas que ambos vean el mismo número, el hook no es la herramienta; necesitas contexto o elevar el estado a un padre común.

Error 3: llamar al hook dentro de condicionales o bucles. Los custom hooks siguen las mismas reglas que los hooks normales: solo en el nivel superior del componente o de otro hook. Nada de `if (condición) { useFetch(url); }`.

Error 4: poner lógica de presentación en el hook. El hook devuelve datos y funciones; el JSX se queda en el componente. Si tu hook empieza a devolver elementos `<div>`, probablemente estás mezclando responsabilidades.

Tres custom hooks que vas a usar toda la vida

Además de `useFetch`, hay tres hooks pequeños que aparecen en casi todos los proyectos. Vale la pena conocerlos a fondo porque te van a ahorrar muchísimo código repetido.

El primero es `useToggle`, que ya viste en los ejercicios: maneja un booleano con su función para alternarlo. Sirve para modales, menús desplegables, secciones que se expanden y se colapsan, y el modo claro/oscuro. Cada vez que escribas `const [abierto, setAbierto] = useState(false)` más una función que hace `setAbierto(!abierto)`, detente: ahí hay un `useToggle` esperando nacer.

El segundo es `useLocalStorage`, también visto en la práctica: sincroniza un estado con el `localStorage` del navegador. Sus usos típicos son recordar el tema elegido por el usuario, guardar el borrador de un formulario para que no se pierda si recarga la página, o persistir las preferencias de la app. Un detalle fino: la inicialización perezosa con la función dentro de `useState` es importante porque leer `localStorage` en cada render sería un desperdicio; con la función solo se lee una vez, al montar el componente.

El tercero es `usePrevious`, que guarda el valor anterior de una variable entre renders:

```
// src/hooks/usePrevious.js
import { useRef, useEffect } from "react";

function usePrevious(valor) {
  const ref = useRef(null);

  useEffect(() => {
    ref.current = valor;
  });

  return ref.current;
}

export default usePrevious;
```

Línea por línea: `useRef` crea una "caja" mutable que persiste entre renders sin provocar re-renderizados cuando cambia. El `useEffect` sin arreglo de dependencias se ejecuta después de CADA render, y ahí guardamos el valor actual en la caja. Pero el hook devuelve `ref.current` ANTES de que el efecto se ejecute, es decir, devuelve el valor de la caja tal como quedó en el render anterior. Resultado: `usePrevious(valor)` te da lo que valía "valor" en el render pasado. Sirve para comparar ("¿el usuario cambió?" , "¿el contador subió o bajó?") y para animaciones que necesitan saber de dónde vienen.

Estos tres hooks caben en una tarde de práctica y los vas a reutilizar en todos tus proyectos. Cuando domines este trío más `useFetch`, tendrás un kit de herramientas personal que te hará mucho más rápido.

React Router: navegación entre páginas sin recargar

El problema: apps de una sola vista

Hasta ahora, todas tus apps viven en una sola URL. Si quieres mostrar contenido diferente, usas un estado booleano o condicional:


```
function App() {
  const [pagina, setPagina] = useState("inicio");

  return (
    <div>
      <button onClick={() => setPagina("inicio")}>Inicio</button>
      <button onClick={() => setPagina("contacto")}>Contacto</button>
      {pagina === "inicio" ? <Inicio /> : <Contacto />}
    </div>
  );
}
```

Esto funciona para demos, pero tiene limitaciones graves en el mundo real: la URL nunca cambia, así que el usuario no puede compartir el enlace de la página de contacto, no puede usar el botón "atrás" del navegador para volver, y no puede marcar la página en favoritos. Además, si recarga la página, vuelve al inicio y pierde dónde estaba. Los usuarios esperan que cada "página" tenga su propia URL, como en cualquier sitio web.

React Router resuelve esto: es una librería que sincroniza la interfaz con la URL del navegador. Tú defines qué componente se muestra en cada ruta, y React Router se encarga del resto.

Instalación

React Router no viene incluido en React; es una librería externa que se instala con npm. En la carpeta de tu proyecto, ejecuta:

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

Este comando descarga la librería y la agrega a las dependencias de tu proyecto. A partir de aquí puedes importarla en tus archivos. El paquete se llama react-router-dom porque está pensado para aplicaciones web (DOM); existe una versión para React Native, pero no la necesitas aquí.

Los cuatro protagonistas: BrowserRouter, Routes, Route y Link

Vamos con cada uno:

BrowserRouter es el componente que envuelve toda tu app y activa el sistema de rutas. Usa la API de historial del navegador para mantener la URL sincronizada sin recargar la página. Normalmente envuelve al componente App en el archivo principal:

```
// main.jsx
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Routes es el contenedor donde defines todas tus rutas. Dentro de él pones componentes Route, uno por cada página.

Route define una ruta individual con dos props principales: path (la URL) y element (el componente a mostrar):

```
// App.jsx
import { Routes, Route } from "react-router-dom";
import Inicio from "./paginas/Inicio";
import Nosotros from "./paginas/Nosotros";
import Contacto from "./paginas/Contacto";

function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Inicio />} />
        <Route path="/nosotros" element={<Nosotros />} />
        <Route path="/contacto" element={<Contacto />} />
      </Routes>
    </div>
  );
}
```

Con esto, cuando la URL es "/" se muestra el componente Inicio; cuando es "/nosotros" se muestra Nosotros; cuando es "/contacto" se muestra Contacto. React Router compara la URL actual con los path definidos y renderiza el primero que coincida.

Link es el componente para navegar entre rutas. Genera un enlace que cambia la URL sin recargar la página:

```
import { Link } from "react-router-dom";

function Menu() {
  return (
    <nav>
      <Link to="/">Inicio</Link>
      <Link to="/nosotros">Nosotros</Link>
      <Link to="/contacto">Contacto</Link>
    </nav>
  );
}
```

La prop se llama "to" en lugar de "href". Al hacer clic, React Router intercepta la navegación, actualiza la URL y muestra el componente correspondiente, todo sin recargar.

Link vs <a>: por qué no debes recargar

Esta es una de las distinciones más importantes de la unidad. En HTML normal, los enlaces se hacen con . Si usas <a> dentro de una app React con Router, el navegador hace lo que siempre hace: pide la página completa al servidor y recarga todo. Pierdes el estado de la app (el usuario logueado, el tema, el carrito), la navegación se siente lenta y rompes la experiencia de "app" que React Router te da.

Link, en cambio, le dice al navegador: "no recargues, yo me encargo". React Router actualiza la URL con la API de historial y cambia solo el componente que corresponde. El estado se conserva, la transición es instantánea y el botón "atrás" del navegador funciona porque el historial sí se actualizó.

Regla de oro: dentro de una app con React Router, para navegar entre tus propias rutas SIEMPRE usa Link (o NavLink). Reserva <a> solo para enlaces externos que salen de tu app, como <https://lucasegaacademy.com>.

NavLink: estilos para la ruta activa

NavLink es el primo de Link con un superpoder: sabe cuándo su ruta está activa (es decir, cuando la URL actual coincide con su "to") y te permite aplicar estilos diferentes en ese caso. Es perfecto para menús de navegación donde quieres resaltar la sección actual:

```
import { NavLink } from "react-router-dom";

function Menu() {
  return (
    <nav>
      <NavLink
        to="/"
        style={({ isActive }) => ({
          color: isActive ? "red" : "blue",
          fontWeight: isActive ? "bold" : "normal"
        })}
      >
        Inicio
      </NavLink>
      <NavLink
        to="/nosotros"
        style={({ isActive }) => ({
          color: isActive ? "red" : "blue",
          fontWeight: isActive ? "bold" : "normal"
        })}
      >
        Nosotros
      </NavLink>
    </nav>
  );
}
```

La prop style recibe una función que recibe un objeto con isActive. Si la ruta está activa, pintamos el enlace de rojo y en negritas; si no, de azul normal. También puedes usar className como función con el mismo objeto. Es un detalle pequeño que hace que tu navegación se sienta profesional.

Rutas anidadas: la idea

Las rutas anidadas permiten que una página tenga sub-páginas que comparten un diseño común. Piensa en un panel de administración: tiene un menú lateral fijo y un área de contenido que cambia entre "usuarios", "productos" y "configuración". Con rutas anidadas defines una ruta padre con un layout y rutas hijas que se muestran dentro de él:

```

<Routes>
  <Route path="/panel" element={<PanelLayout />}>
    <Route path="usuarios" element={<Usuarios />} />
    <Route path="productos" element={<Productos />} />
  </Route>
</Routes>

```

La URL `/panel/usuarios` muestra `PanelLayout` con el componente `Usuarios` dentro. El componente padre usa `<Outlet />` (que también viene de `react-router-dom`) para indicar dónde se muestran las rutas hijas:

```

import { Outlet, Link } from "react-router-dom";

function PanelLayout() {
  return (
    <div>
      <aside>
        <Link to="/panel/usuarios">Usuarios</Link>
        <Link to="/panel/productos">Productos</Link>
      </aside>
      <main>
        <Outlet />
      </main>
    </div>
  );
}

```

`Outlet` es como un marcador de posición: "aquí va el contenido de la ruta hija". El menú lateral se muestra siempre; solo cambia lo que está dentro de `Outlet`. No necesitas dominar las rutas anidadas hoy, pero debes conocer la idea porque las verás en proyectos reales.

Página 404: `path="*"`

¿Qué pasa si el usuario escribe una URL que no existe, como `/página-inventada`? Sin una ruta que la capture, React Router no muestra nada: pantalla en blanco. La solución es una ruta comodín con `path="*"`, que coincide con cualquier URL no capturada por las rutas anteriores:

```
<Routes>
  <Route path="/" element={<Inicio />} />
  <Route path="/nosotros" element={<Nosotros />} />
  <Route path="/contacto" element={<Contacto />} />
  <Route path="*" element={<NoEncontrada />} />
</Routes>
```

El asterisco significa "cualquier otra cosa". Es importante ponerla AL FINAL, porque Routes renderiza la primera coincidencia: si la pones al principio, atraparía todas las URLs y tus demás rutas nunca se mostrarían. El componente NoEncontrada típicamente muestra un mensaje amable ("Página no encontrada") con un Link de regreso al inicio.

Analogías para no olvidar

Piensa en el contexto como el sistema de sonido de un edificio: el Provider es la bocina central que emite el mensaje, y cada componente con useContext es una persona con un radio sintonizado a esa frecuencia. No importa en qué piso esté; si está dentro del edificio (dentro del Provider), escucha el mensaje.

Piensa en un custom hook como una receta de cocina escrita en una tarjeta. Cada cocinero (componente) que toma la tarjeta prepara su propio platillo con sus propios ingredientes (su propio estado). La receta se comparte; el platillo no.

Piensa en React Router como el sistema de señalización de un museo: cada sala tiene un nombre en la puerta (path) y un contenido (element). El visitante (usuario) camina de sala en sala (Link) sin salir del edificio (sin recargar). Y si abre una puerta que no existe, encuentra el cartel de "sala no encontrada" (la ruta *).

Buenas prácticas

Separa tus contextos por dominio: un archivo por contexto (TemaContext.js, UsuarioContext.js), cada uno con su responsabilidad clara. Si un contexto necesita lógica más elaborada (por ejemplo, el carrito con agregar, quitar y vaciar), crea un componente proveedor personalizado que encapsule el estado y lo exponga, en lugar de poner toda la lógica en App.

Guarda tus custom hooks en src/hooks/, un archivo por hook, con el nombre del archivo igual al nombre del hook. Guarda tus páginas en src/páginas/ (o src/pages/) separadas de los componentes reutilizables en src/componentes/. Esta organización hace que cualquier persona que abra tu proyecto entienda dónde está cada cosa.

Para las rutas, define los path como constantes si se repiten en varios lugares, y mantén la ruta 404 siempre al final. Usa Link para navegación interna y <a> solo para URLs externas.

useNavigate: navegar desde el código

Link y NavLink sirven cuando el usuario hace clic en un enlace. Pero a veces necesitas navegar desde el código, sin clic: por ejemplo, después de que el usuario envía un formulario de login con éxito, quieres mandarlo a su perfil automáticamente. Para eso existe el hook useNavigate:

```
import { useNavigate } from "react-router-dom";

function FormularioLogin() {
  const navigate = useNavigate();

  function manejarEnvio(e) {
    e.preventDefault();
    // ...validar usuario y contraseña...
    navigate("/perfil");
  }

  return (
    <form onSubmit={manejarEnvio}>
      <input type="text" placeholder="Usuario" />
      <input type="password" placeholder="Contraseña" />
      <button type="submit">Entrar</button>
    </form>
  );
}
```

Línea por línea: useNavigate devuelve una función (por convención se llama navigate). Al llamarla con una ruta, React Router navega a esa URL sin recargar, igual que si el usuario hubiera hecho clic en un Link. También puedes navegar hacia atrás con navigate(-1), que equivale al botón "atrás" del navegador, útil para un botón de "volver".

La regla para decidir: si hay un elemento visible en el que el usuario hace clic para ir a otra página, usa Link o NavLink. Si la navegación es consecuencia de una acción (login exitoso, formulario guardado, compra completada), usa navigate. Ambos hacen lo mismo por debajo; solo cambia quién inicia la navegación.

Contexto y rendimiento: lo que debes saber

Hay un detalle del contexto que todo profesional conoce: cada vez que el value del Provider cambia, TODOS los componentes que consumen ese contexto se vuelven a renderizar, aunque solo usen una parte del valor. Mira este caso:

```
function App() {
  const [usuario, setUsuario] = useState(null);
  const [tema, setTema] = useState("claro");

  return (
    <DatosContext.Provider value={{ usuario, setUsuario, tema, setTema }}>
      <Encabezado />
      <Contenido />
    </DatosContext.Provider>
  );
}
```

Aquí metimos usuario y tema en un solo contexto. El problema: cuando el usuario cambia el tema, el objeto value es nuevo (porque tema cambió), así que React re-renderiza a todos los consumidores, incluyendo los que solo usan "usuario" y ni se enteraron del cambio de tema. En una app pequeña no lo notas; en una app con cientos de componentes consumiendo el contexto, cada cambio provoca una ola de re-renderizados innecesarios y la app se siente lenta.

La solución es la que ya vimos: separa los contextos por dominio. Un UsuarioContext y un TemaContext independientes. Así, cambiar el tema solo re-renderiza a los que leen el tema, y cambiar el usuario solo afecta a los que leen el usuario. Esta es la razón práctica detrás de la buena práctica de "contextos separados por dominio": no es solo organización bonita, es rendimiento real.

Un segundo detalle: el objeto que pasas en value se crea nuevo en cada render del componente que contiene el Provider. Si App se re-renderiza por cualquier motivo (por ejemplo, otro estado que no tiene nada que ver), el value es un objeto nuevo y todos los consumidores se re-renderizan aunque los datos sean idénticos. Para apps grandes, la solución es memorizar el value con useMemo:

```
const valor = useMemo(
  () => ({ usuario, setUsuario }),
  [usuario]
);
```

Con useMemo, el objeto solo se recrea cuando "usuario" cambia de verdad; setUsuario es estable entre renders, así que no necesita estar en las dependencias para este caso simple. No necesitas aplicar esto desde el día uno: es una optimización para cuando tu app crezca y notes lentitud. Pero entender el mecanismo (value nuevo = re-render de consumidores) te evita sorpresas y te hace escribir contextos mejor diseñados desde el principio.

En resumen: un contexto por dominio, value con solo lo necesario, y useMemo cuando el rendimiento lo pida. Con esas tres reglas, el contexto escala sin problemas a aplicaciones grandes.

Repaso rápido: qué herramienta usar y cuándo

Con tres herramientas nuevas es fácil confundirse, así que aquí va una guía de decisión que puedes consultar siempre:

¿Necesitas que un dato llegue a muchos componentes en distintos niveles? Si son 1 o 2 niveles, usa props. Si son 3 o más y los intermedios no usan el dato, usa contexto.

¿El dato cambia con cada tecla que el usuario escribe? No lo pongas en contexto; el re-renderizado masivo te va a dar problemas de rendimiento. Mantenlo en estado local con useState.

¿La misma lógica con estado aparece en dos o más componentes? Extráela a un custom hook. Si solo aparece en uno, déjala en el componente; no crees hooks por adelantado para casos que quizá nunca se repitan.

¿Necesitas varias "páginas" con URL propia? Usa React Router. Si solo alternas vistas dentro de la misma pantalla sin necesidad de URL compartible, un estado condicional basta.

¿Dos componentes necesitan ver el MISMO estado (no una copia)? Eso no lo resuelve un custom hook; necesitas elevar el estado a un padre común o usar contexto.

Memoriza esta guía y te vas a ahorrar la mayoría de los errores de arquitectura que cometen los principiantes: contexto para datos globales estables, hooks para lógica reutilizable, router para páginas.

Ejemplos paso a paso

Ejemplo 1: contexto de tema con interruptor

Vamos a construir un interruptor de tema claro/oscuro completo, explicado línea por línea. Son tres archivos.

Archivo 1: el contexto.

```
// src/TemaContext.js
import { createContext } from "react";

export const TemaContext = createContext("claro");
```

Línea por línea: importamos createContext de react. Creamos el contexto con valor por defecto "claro" y lo exportamos para usarlo en otros archivos. Este archivo no tiene componentes; solo define el "canal" por donde viajará el dato.

Archivo 2: la app que provee el tema.

```
// src/App.jsx
import { useState } from "react";
import { TemaContext } from "../TemaContext";
import Interruptor from "../Interruptor";
import Tarjeta from "../Tarjeta";

function App() {
  const [tema, setTema] = useState("claro");

  return (
    <TemaContext.Provider value={tema}>
      <div>
        <h1>Mi app</h1>
        <Interruptor cambiarTema={setTema} temaActual={tema} />
        <Tarjeta />
      </div>
    </TemaContext.Provider>
  );
}

export default App;
```

Línea por línea: useState guarda el tema actual, que empieza en "claro". El Provider envuelve todo y reparte el valor "tema". Interruptor recibe setTema por props porque es hijo directo (aquí las props están bien, no hay drilling). Tarjeta no recibe nada: leerá el tema del contexto aunque esté a cualquier profundidad.

Archivo 3: los consumidores.

```
// src/Interruptor.jsx
import { useContext } from "react";
import { TemaContext } from "../TemaContext";

function Interruptor({ cambiarTema, temaActual }) {
  const tema = useContext(TemaContext);

  return (
    <button onClick={() => cambiarTema(tema === "claro" ? "oscuro" : "claro")}>
      Tema actual: {tema} (clic para cambiar)
    </button>
  );
}

export default Interruptor;
```

```
// src/Tarjeta.jsx
import { useContext } from "react";
import { TemaContext } from "../TemaContext";

function Tarjeta() {
  const tema = useContext(TemaContext);
  const estilos = {
    background: tema === "oscuro" ? "#333" : "#fff",
    color: tema === "oscuro" ? "#fff" : "#000",
    padding: "20px",
    border: "1px solid #ccc"
  };

  return (
    <div style={estilos}>
      <h2>Tarjeta de ejemplo</h2>
      <p>Esta tarjeta cambia con el tema: {tema}</p>
    </div>
  );
}

export default Tarjeta;
```

Línea por línea de Tarjeta: importa useContext y el contexto. Llama a useContext(TemaContext) y guarda el valor en "tema". Construye un objeto de estilos que depende del tema. Renderiza la tarjeta con esos estilos. Cuando el usuario hace clic en el botón del Interruptor, el estado cambia en App, el Provider reparte el nuevo valor y la tarjeta se repinta sola. Cero props atravesando niveles.

Ejemplo 2: custom hook useFetch reutilizable

Ahora el hook useFetch completo, con dos componentes que lo usan para URLs distintas.

```
// src/hooks/useFetch.js
import { useState, useEffect } from "react";

function useFetch(url) {
  const [datos, setDatos] = useState(null);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    setCargando(true);
    setError(null);

    fetch(url)
      .then((respuesta) => {
        if (!respuesta.ok) {
          throw new Error("Error en la petición: " + respuesta.status);
        }
        return respuesta.json();
      })
      .then((json) => {
        setDatos(json);
        setCargando(false);
      })
      .catch((err) => {
        setError(err.message);
        setCargando(false);
      });
  }, [url]);

  return { datos, cargando, error };
}

export default useFetch;
```

Línea por línea: el hook recibe la URL como parámetro. Declara tres estados: datos (empieza en null), cargando (empieza en true) y error (empieza en null). El useEffect se ejecuta cuando el componente se monta y cada vez que la URL cambia ([url] en el arreglo de dependencias). Dentro, reinicia cargando y error, hace el fetch, verifica que la respuesta sea exitosa, convierte a JSON, guarda los datos y apaga "cargando". Si algo falla, guarda el mensaje de error y también apaga "cargando". Devuelve un objeto con los tres valores para que el componente elija cuáles usar por nombre, sin depender del orden.

Ahora dos componentes que lo reutilizan:

```
// src/ListaUsuarios.jsx
import useFetch from "../hooks/useFetch";

function ListaUsuarios() {
  const { datos, cargando, error } = useFetch("https://api.ejemplo.com/usuarios");

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

  return (
    <ul>
      {datos.map((u) => (
        <li key={u.id}>{u.nombre}</li>
      ))}
    </ul>
  );
}

export default ListaUsuarios;
```

```
// src/ListaProductos.jsx
import useFetch from "../hooks/useFetch";

function ListaProductos() {
  const { datos, cargando, error } = useFetch("https://api.ejemplo.com/productos");

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

  return (
    <ul>
      {datos.map((p) => (
        <li key={p.id}>{p.nombre} - ${p.precio}</li>
      ))}
    </ul>
  );
}

export default ListaProductos;
```

Cada componente llama al hook con su propia URL y obtiene su propio estado independiente. La lógica de fetch, manejo de errores y estado de carga está escrita una sola vez. Si mañana agregas un timeout o reintentos, lo haces en useFetch y ambos componentes mejoran automáticamente.

Ejemplo 3: app con 3 rutas y menú de navegación

Primero instala la librería en la terminal, dentro de la carpeta de tu proyecto:

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

Verás que npm descarga el paquete y actualiza tu package.json. Ahora el código. Archivo principal:

```
// src/main.jsx
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Línea por línea: importamos BrowserRouter de react-router-dom. Envolvemos App con BrowserRouter para activar el sistema de rutas en toda la aplicación. Sin este envoltorio, los componentes Routes, Route y Link no funcionan.

Las páginas (componentes simples, uno por archivo):

```
// src/paginas/Inicio.jsx
function Inicio() {
  return (
    <div>
      <h2>Bienvenido al inicio</h2>
      <p>Esta es la página principal de nuestro mini sitio.</p>
    </div>
  );
}

export default Inicio;
```

```
// src/paginas/Nosotros.jsx
function Nosotros() {
  return (
    <div>
      <h2>Nosotros</h2>
      <p>Somos un equipo que aprende React en Lucas EGA Academy.</p>
    </div>
  );
}

export default Nosotros;
```

```
// src/paginas/Contacto.jsx
function Contacto() {
  return (
    <div>
      <h2>Contacto</h2>
      <p>Escríbenos a hola@ejemplo.com</p>
    </div>
  );
}

export default Contacto;
```

```
// src/paginas/NoEncontrada.jsx
import { Link } from "react-router-dom";

function NoEncontrada() {
  return (
    <div>
      <h2>404: Página no encontrada</h2>
      <p>La dirección que buscas no existe.</p>
      <Link to="/">Volver al inicio</Link>
    </div>
  );
}

export default NoEncontrada;
```

El menú y las rutas en App:


```
// src/App.jsx
import { Routes, Route, NavLink } from "react-router-dom";
import Inicio from "../paginas/Inicio";
import Nosotros from "../paginas/Nosotros";
import Contacto from "../paginas/Contacto";
import NoEncontrada from "../paginas/NoEncontrada";

function App() {
  const estiloEnlace = ({ isActive }) => ({
    color: isActive ? "red" : "blue",
    fontWeight: isActive ? "bold" : "normal",
    marginRight: "15px"
  });

  return (
    <div>
      <nav>
        <NavLink to="/" style={estiloEnlace}>Inicio</NavLink>
        <NavLink to="/nosotros" style={estiloEnlace}>Nosotros</NavLink>
        <NavLink to="/contacto" style={estiloEnlace}>Contacto</NavLink>
      </nav>

      <hr />

      <Routes>
        <Route path="/" element={<Inicio />} />
        <Route path="/nosotros" element={<Nosotros />} />
        <Route path="/contacto" element={<Contacto />} />
        <Route path="*" element={<NoEncontrada />} />
      </Routes>
    </div>
  );
}

export default App;
```

Línea por línea: importamos Routes, Route y NavLink. Definimos una función estiloEnlace que recibe { isActive } y devuelve estilos: rojo y negritas si la ruta está activa, azul normal si no. El nav muestra los tres enlaces; el activo se resalta automáticamente. Routes contiene las cuatro rutas: cada path muestra su componente. La ruta "*" va al final y atrapa cualquier URL desconocida mostrando la página 404. Prueba escribir "/inventada" en la URL: verás el mensaje de no encontrada con su enlace de regreso.

Ejemplo 4: página de detalle simulada (rutas dinámicas)

En las apps reales, muchas páginas dependen de un dato de la URL, como `/producto/42` donde 42 es el id del producto. React Router maneja esto con parámetros de ruta: defines `path="/producto/:id"` y lees el valor con el hook `useParams`.

Primero una lista de productos con enlaces:

```
// src/paginas/Productos.jsx
import { Link } from "react-router-dom";

const productos = [
  { id: 1, nombre: "Teclado", precio: 500 },
  { id: 2, nombre: "Mouse", precio: 300 },
  { id: 3, nombre: "Monitor", precio: 3500 }
];

function Productos() {
  return (
    <div>
      <h2>Productos</h2>
      <ul>
        {productos.map((p) => (
          <li key={p.id}>
            <Link to={"/producto/" + p.id}>{p.nombre}</Link>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default Productos;
```

Línea por línea: definimos un arreglo de productos simulados (en una app real vendrían de una API). Mapeamos el arreglo a una lista; cada elemento tiene un `Link` que apunta a `/producto/` más el id. Al hacer clic, la URL cambia a por ejemplo `/producto/2` sin recargar.

Ahora la página de detalle que lee el parámetro:

```
// src/paginas/DetalleProducto.jsx
import { useParams, Link } from "react-router-dom";

const productos = [
  { id: 1, nombre: "Teclado", precio: 500, descripcion: "Teclado mecánico en español." },
  { id: 2, nombre: "Mouse", precio: 300, descripcion: "Mouse óptico inalámbrico." },
  { id: 3, nombre: "Monitor", precio: 3500, descripcion: "Monitor de 24 pulgadas Full HD." }
];

function DetalleProducto() {
  const { id } = useParams();
  const producto = productos.find((p) => p.id === Number(id));

  if (!producto) {
    return (
      <div>
        <p>Producto no encontrado.</p>
        <Link to="/productos">Volver a productos</Link>
      </div>
    );
  }

  return (
    <div>
      <h2>{producto.nombre}</h2>
      <p>Precio: ${producto.precio}</p>
      <p>{producto.descripcion}</p>
      <Link to="/productos">Volver a productos</Link>
    </div>
  );
}

export default DetalleProducto;
```

Línea por línea: importamos useParams, el hook que lee los parámetros de la URL. const { id } = useParams() extrae el valor que ocupa el lugar de ":id" en el path; si la URL es "/producto/2", id vale "2" (siempre llega como texto, por eso lo convertimos con Number). Buscamos el producto en el arreglo; si no existe mostramos un mensaje con enlace de regreso. Si existe, mostramos sus datos. Las rutas en App quedan así:

```
<Routes>
  <Route path="/" element={<Inicio />} />
  <Route path="/productos" element={<Productos />} />
  <Route path="/producto/:id" element={<DetalleProducto />} />
  <Route path="*" element={<NoEncontrada />} />
</Routes>
```

Con un solo Route atendemos infinitos productos: "/producto/1", "/producto/2", "/producto/3"... cada uno muestra su detalle. Este patrón (lista -> detalle con parámetro) es la columna vertebral de casi cualquier app con contenido: tiendas, blogs, redes sociales.

Práctica

Ejercicio 1: contexto de idioma ES/EN

Enunciado: crea un contexto IdiomaContext que guarde el idioma actual ("es" o "en") y la función para cambiarlo. Un componente Saludo debe mostrar "Hola" si el idioma es "es" y "Hello" si es "en". Un botón debe alternar entre los dos idiomas. Toda la app debe estar envuelta en el Provider.

Solución:

```
// src/IdiomaContext.js
import { createContext } from "react";

export const IdiomaContext = createContext("es");
```

```
// src/App.jsx
import { useState } from "react";
import { IdiomaContext } from "../IdiomaContext";
import Saludo from "../Saludo";

function App() {
  const [idioma, setIdioma] = useState("es");

  function alternarIdioma() {
    setIdioma(idioma === "es" ? "en" : "es");
  }

  return (
    <IdiomaContext.Provider value={{ idioma, setIdioma }}>
      <Saludo />
      <button onClick={alternarIdioma}>
        Cambiar a {idioma === "es" ? "inglés" : "español"}
      </button>
    </IdiomaContext.Provider>
  );
}

export default App;
```

```
// src/Saludo.jsx
import { useContext } from "react";
import { IdiomaContext } from "../IdiomaContext";

function Saludo() {
  const { idioma } = useContext(IdiomaContext);

  return <h1>{idioma === "es" ? "Hola" : "Hello"}</h1>;
}

export default Saludo;
```

Explicación: el contexto se crea con valor por defecto "es". El Provider pasa un objeto con el idioma y setIdioma, siguiendo el patrón que vimos con el usuario logueado. Saludo desestructura { idioma } del contexto y elige el texto según su valor. El botón alterna el estado en App y todo se actualiza solo. Nota que Saludo no recibe props: podría estar a diez niveles de profundidad y seguiría funcionando igual.

Ejercicio 2: hook useLocalStorage

Enunciado: crea un custom hook useLocalStorage(clave, valorInicial) que guarde un estado en el localStorage del navegador. Debe leer el valor guardado al iniciar (o usar valorInicial si no hay nada) y escribir en localStorage cada vez que el estado cambie. Úsalo en un componente para guardar el nombre del usuario en un input.

Solución:

```
// src/hooks/useLocalStorage.js
import { useState, useEffect } from "react";

function useLocalStorage(clave, valorInicial) {
  const [valor, setValor] = useState(() => {
    const guardado = localStorage.getItem(clave);
    if (guardado === null) {
      return valorInicial;
    }
    return JSON.parse(guardado);
  });

  useEffect(() => {
    localStorage.setItem(clave, JSON.stringify(valor));
  }, [clave, valor]);

  return [valor, setValor];
}

export default useLocalStorage;
```

```
// src/NombreUsuario.jsx
import useLocalStorage from "../hooks/useLocalStorage";

function NombreUsuario() {
  const [nombre, setNombre] = useLocalStorage("nombre-usuario", "");

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

export default NombreUsuario;
```

Explicación: el `useState` recibe una función inicializadora (lazy initialization): solo se ejecuta una vez, al montar. Ahí leemos `localStorage`; si no hay nada guardado devolvemos `valorInicial`, si hay algo lo convertimos de texto a valor con `JSON.parse`. El `useEffect` escribe en `localStorage` cada vez que "valor" o "clave" cambien, usando `JSON.stringify` para guardar cualquier tipo de dato. El hook devuelve `[valor, setValor]` igual que `useState`, así que usarlo se siente idéntico. Si recargas la página, el nombre sigue ahí: la persistencia funciona. Este hook es uno de los más útiles en la vida real.

Ejercicio 3: hook `useToggle`

Enunciado: crea un custom hook `useToggle(valorInicial)` que maneje un booleano con una función para alternarlo. Debe devolver `[valor, alternar]`. Úsalo en un componente que muestre u oculte un texto secreto.

Solución:

```
// src/hooks/useToggle.js
import { useState } from "react";

function useToggle(valorInicial = false) {
  const [valor, setValor] = useState(valorInicial);

  function alternar() {
    setValor(!valor);
  }

  return [valor, alternar];
}

export default useToggle;
```

```
// src/Secreto.jsx
import useToggle from "../hooks/useToggle";

function Secreto() {
  const [visible, alternar] = useToggle(false);

  return (
    <div>
      <button onClick={alternar}>
        {visible ? "Ocultar secreto" : "Mostrar secreto"}
      </button>
      {visible && <p>El secreto es: React es increíble.</p>}
    </div>
  );
}

export default Secreto;
```

Explicación: el hook encapsula el patrón tan repetido de "booleano + función que lo niega". El parámetro `valorInicial` tiene un valor por defecto (`false`) para que `useToggle()` funcione sin argumentos. El componente desestructura `[visible, alternar]` y lo usa de forma declarativa: el texto del botón y la visibilidad del párrafo dependen de "visible". Cada componente que use `useToggle` tiene su propio booleano independiente. Parece simple, y lo es: esa es la gracia de los custom hooks, convertir patrones repetidos en una línea.

Ejercicio 4: mini sitio con Inicio/Nosotros/Contacto + 404

Enunciado: instala react-router-dom y crea un mini sitio con tres páginas (Inicio, Nosotros, Contacto), un menú con NavLink que resalte la página activa y una página 404 para rutas inexistentes. El menú debe estar siempre visible.

Solución:

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

```
// src/main.jsx
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

```
// src/paginas/Inicio.jsx
function Inicio() {
  return (
    <div>
      <h2>Inicio</h2>
      <p>Bienvenido a nuestro mini sitio hecho con React Router.</p>
    </div>
  );
}

export default Inicio;
```

```
// src/paginas/Nosotros.jsx
function Nosotros() {
  return (
    <div>
      <h2>Nosotros</h2>
      <p>Aprendemos React paso a paso en Lucas EGA Academy.</p>
    </div>
  );
}

export default Nosotros;
```

```
// src/paginas/Contacto.jsx
function Contacto() {
  return (
    <div>
      <h2>Contacto</h2>
      <p>Escríbenos cuando quieras: hola@ejemplo.com</p>
    </div>
  );
}

export default Contacto;
```

```
// src/paginas/NoEncontrada.jsx
import { Link } from "react-router-dom";

function NoEncontrada() {
  return (
    <div>
      <h2>Error 404</h2>
      <p>Esa página no existe.</p>
      <Link to="/">Ir al inicio</Link>
    </div>
  );
}

export default NoEncontrada;
```

```

// src/App.jsx
import { Routes, Route, NavLink } from "react-router-dom";
import Inicio from "../paginas/Inicio";
import Nosotros from "../paginas/Nosotros";
import Contacto from "../paginas/Contacto";
import NoEncontrada from "../paginas/NoEncontrada";

function App() {
  function estilos({ isActive }) {
    return {
      color: isActive ? "white" : "black",
      background: isActive ? "black" : "transparent",
      padding: "8px 12px",
      marginRight: "10px",
      textDecoration: "none"
    };
  };

  return (
    <div>
      <nav>
        <NavLink to="/" style={estilos}>Inicio</NavLink>
        <NavLink to="/nosotros" style={estilos}>Nosotros</NavLink>
        <NavLink to="/contacto" style={estilos}>Contacto</NavLink>
      </nav>
      <hr />
      <Routes>
        <Route path="/" element={<Inicio />} />
        <Route path="/nosotros" element={<Nosotros />} />
        <Route path="/contacto" element={<Contacto />} />
        <Route path="*" element={<NoEncontrada />} />
      </Routes>
    </div>
  );
}

export default App;

```

Explicación: el menú vive fuera de Routes, así que está siempre visible sin importar la página. NavLink con la función de estilos resalta la ruta activa con fondo negro y texto blanco. Las tres rutas normales atienden sus URLs y la ruta "*" al final captura todo lo demás. Prueba navegar y luego escribir una URL inventada: verás la 404. Fíjate que en ningún momento la página recarga: el estado se conserva y la navegación es instantánea.

Ejercicio 5: carrito simple con context

Enunciado: crea un CarritoContext que guarde un arreglo de productos y una función agregarProducto. Un componente ListaTienda muestra 3 productos con botón "Agregar". Un componente VerCarrito muestra los productos agregados y el total. Ambos deben leer el mismo contexto.

Solución:

```
// src/CarritoContext.js
import { createContext } from "react";

export const CarritoContext = createContext(null);
```

```
// src/App.jsx
import { useState } from "react";
import { CarritoContext } from "../CarritoContext";
import ListaTienda from "../ListaTienda";
import VerCarrito from "../VerCarrito";

function App() {
  const [carrito, setCarrito] = useState([]);

  function agregarProducto(producto) {
    setCarrito([...carrito, producto]);
  }

  return (
    <CarritoContext.Provider value={{ carrito, agregarProducto }}>
      <h1>Mi tiendita</h1>
      <ListaTienda />
      <hr />
      <VerCarrito />
    </CarritoContext.Provider>
  );
}

export default App;
```

```

// src/ListaTienda.jsx
import { useContext } from "react";
import { CarritoContext } from "../CarritoContext";

const productos = [
  { id: 1, nombre: "Playera", precio: 200 },
  { id: 2, nombre: "Gorra", precio: 150 },
  { id: 3, nombre: "Taza", precio: 100 }
];

function ListaTienda() {
  const { agregarProducto } = useContext(CarritoContext);

  return (
    <div>
      <h2>Productos</h2>
      <ul>
        {productos.map((p) => (
          <li key={p.id}>
            {p.nombre} - ${p.precio}{" "}
            <button onClick={() => agregarProducto(p)}>Agregar</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default ListaTienda;

```

```
// src/VerCarrito.jsx
import { useContext } from "react";
import { CarritoContext } from "../CarritoContext";

function VerCarrito() {
  const { carrito } = useContext(CarritoContext);
  const total = carrito.reduce((suma, p) => suma + p.precio, 0);

  return (
    <div>
      <h2>Carrito ({carrito.length})</h2>
      {carrito.length === 0 ? (
        <p>El carrito está vacío.</p>
      ) : (
        <ul>
          {carrito.map((p, i) => (
            <li key={i}>{p.nombre} - ${p.precio}</li>
          ))}
        </ul>
      )}
      <p><strong>Total: ${total}</strong></p>
    </div>
  );
}

export default VerCarrito;
```

Explicación: el estado del carrito vive en App y se comparte vía contexto. ListaTienda solo necesita agregarProducto; VerCarrito solo necesita carrito; cada uno lee lo suyo del mismo contexto. agregarProducto crea un arreglo nuevo con el spread operator ([...carrito, producto]) en lugar de modificar el existente, porque React necesita un arreglo nuevo para detectar el cambio. El total se calcula con reduce. Este ejemplo junta todo lo de la unidad: estado compartido global (contexto) que en una app real combinarías con rutas (página de tienda, página de carrito) y hooks personalizados.

Ejercicio 6 (reto): mini app que combina las tres herramientas

Enunciado: construye una mini app con dos páginas ("/" con un contador y "/saludo" con un mensaje). Usa un custom hook useContador para la lógica del contador, un contexto TemaContext para el tema claro/oscuro disponible en ambas páginas, y React Router para navegar entre ellas. Debe incluir la página 404.

Solución:

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

```
// src/hooks/useContador.js
import { useState } from "react";

function useContador(valorInicial = 0) {
  const [cuenta, setCuenta] = useState(valorInicial);

  function incrementar() {
    setCuenta(cuenta + 1);
  }

  function reiniciar() {
    setCuenta(valorInicial);
  }

  return [cuenta, incrementar, reiniciar];
}

export default useContador;
```

```

// src/TemaContext.js
import { createContext, useState, useContext } from "react";

const TemaContext = createContext(null);

export function TemaProvider({ children }) {
  const [tema, setTema] = useState("claro");

  function alternarTema() {
    setTema(tema === "claro" ? "oscuro" : "claro");
  }

  return (
    <TemaContext.Provider value={{ tema, alternarTema }}>
      {children}
    </TemaContext.Provider>
  );
}

export function useTema() {
  const contexto = useContext(TemaContext);
  if (contexto === null) {
    throw new Error("useTema debe usarse dentro de TemaProvider");
  }
  return contexto;
}

```



```
// src/paginas/ContadorPagina.jsx
import useContador from "../hooks/useContador";
import { useTema } from "../TemaContext";

function ContadorPagina() {
  const [cuenta, incrementar, reiniciar] = useContador(0);
  const { tema } = useTema();

  return (
    <div style={{ background: tema === "oscuro" ? "#222" : "#fff",
      color: tema === "oscuro" ? "#fff" : "#000",
      padding: "20px" }}>
      <h2>Contador: {cuenta}</h2>
      <button onClick={incrementar}>Incrementar</button>
      <button onClick={reiniciar}>Reiniciar</button>
    </div>
  );
}

export default ContadorPagina;
```

```
// src/paginas/SaludoPagina.jsx
import { useTema } from "../TemaContext";

function SaludoPagina() {
  const { tema, alternarTema } = useTema();

  return (
    <div style={{ background: tema === "oscuro" ? "#222" : "#fff",
      color: tema === "oscuro" ? "#fff" : "#000",
      padding: "20px" }}>
      <h2>Hola, esta es la página de saludo</h2>
      <p>Tema actual: {tema}</p>
      <button onClick={alternarTema}>Cambiar tema</button>
    </div>
  );
}

export default SaludoPagina;
```

```
// src/paginas/NoEncontrada.jsx
import { Link } from "react-router-dom";

function NoEncontrada() {
  return (
    <div>
      <h2>404: no existe esta página</h2>
      <Link to="/">Volver al inicio</Link>
    </div>
  );
}

export default NoEncontrada;
```

```
// src/App.jsx
import { Routes, Route, Link } from "react-router-dom";
import { TemaProvider } from "../TemaContext";
import ContadorPagina from "../paginas/ContadorPagina";
import SaludoPagina from "../paginas/SaludoPagina";
import NoEncontrada from "../paginas/NoEncontrada";

function App() {
  return (
    <TemaProvider>
      <nav>
        <Link to="/">Contador</Link>{" | "}
        <Link to="/saludo">Saludo</Link>
      </nav>
      <hr />
      <Routes>
        <Route path="/" element={<ContadorPagina />} />
        <Route path="/saludo" element={<SaludoPagina />} />
        <Route path="*" element={<NoEncontrada />} />
      </Routes>
    </TemaProvider>
  );
}

export default App;
```

Explicación: aquí conviven las tres herramientas sin estorbarse. El custom hook useContador encapsula la lógica del contador con valor inicial configurable y función de reinicio; la página del contador lo usa sin saber nada del tema ni del router. El contexto del tema sigue el patrón

profesional: TemaProvider encapsula el estado y useTema valida que se use dentro del Provider; ambas páginas leen el tema y la página de saludo lo alterna. El Router define las dos rutas más la 404, y el menú con Link vive dentro del Provider para que todo tenga acceso al tema. Fíjate en el orden de los envoltorios en App: TemaProvider está por fuera de Routes, así que todas las páginas quedan dentro del contexto. Si lo pusieras al revés (el Provider dentro de una página), las demás páginas no tendrían acceso al tema. Este esqueleto (provider global + router + hooks) es exactamente la base de una app profesional real; a partir de aquí solo crece en contenido.

Resumen de la unidad

En esta unidad aprendiste las tres herramientas que convierten tus apps de juguete en aplicaciones profesionales.

useContext te permite compartir datos entre componentes sin props drilling. Creas el contexto con createContext (una vez, fuera de los componentes), lo repartes con el Provider usando la prop value, y lo lees donde lo necesites con useContext. Es ideal para datos globales que cambian poco: tema, usuario logueado, idioma. Puedes anidar varios Providers para separar dominios. Pero no lo uses para todo: si el dato solo lo necesitan padre e hijo, las props son más simples; y si cambia con cada tecla, el contexto puede volver lenta tu app porque cada cambio re-renderiza a todos los consumidores.

Los custom hooks te permiten extraer lógica con estado en funciones reutilizables que empiezan con "use". No comparten estado entre componentes: comparten lógica; cada componente que llama al hook obtiene su propia instancia del estado. Viste cómo extraer un hook paso a paso desde un componente (useFetch), cómo un hook puede usar otros hooks (useTema, composición), y las buenas prácticas: un archivo por hook en src/hooks/, una sola responsabilidad por hook. Los errores típicos son olvidar el "use" en el nombre, creer que el estado se comparte y llamar hooks dentro de condicionales.

React Router te da navegación entre páginas sin recargar. Se instala con npm install react-router-dom. BrowserRouter envuelve tu app y activa el sistema; Routes contiene tus Route con path y element; Link navega sin recargar (nunca uses <a> para rutas internas); NavLink resalta la ruta activa; la ruta path="/" al final captura las URLs inexistentes con una página 404; y con rutas dinámicas como /producto/:id más el hook useParams puedes atender infinitas páginas de detalle con un solo Route.

Las tres herramientas se combinan en apps reales: Router para las páginas, contexto para los datos globales y custom hooks para la lógica reutilizable. En la práctica hiciste un interruptor

de tema con contexto, un `useFetch` reutilizable, un mini sitio con tres rutas y 404, una página de detalle con parámetros, y resolviste cinco ejercicios: contexto de idioma, hook `useLocalStorage`, hook `useToggle`, mini sitio completo y carrito con contexto.

Lo que sigue: con estas herramientas ya puedes construir el esqueleto completo de una aplicación real. En las próximas unidades verás cómo traer datos de APIs reales, manejar formularios como profesional y optimizar el rendimiento de tus apps. Sigue practicando: cada hook que extraigas y cada ruta que definas te acerca al nivel de un desarrollador React profesional.

Un último consejo antes de cerrar: no intentes memorizar cada API de memoria. Lo que sí debes llevarte grabado son los patrones: el Provider que envuelve y reparte, el hook que encapsula lógica reutilizable, y la ruta que conecta una URL con un componente. Con esos tres patrones en la cabeza, la documentación oficial se vuelve una simple referencia para consultar detalles, no un muro que memorizar. La diferencia entre un principiante y un profesional no es cuántas funciones recuerda, sino cuántos patrones reconoce al ver un problema nuevo. Esta unidad te dio tres de los más importantes.

UNIDAD 9: Rutas dinámicas, validación y rendimiento

Ya sabes usar rutas, estados, formularios básicos y efectos. Con eso puedes armar una app que se ve bien en una demo. Pero hay una diferencia grande entre una demo que funciona y un producto que alguien usaría de verdad todos los días. Esta unidad se trata exactamente de eso: de cruzar la línea que separa el prototipo del producto profesional.

Piensa en lo que pasa cuando una app crece. Al principio todo es estático: tres páginas, dos formularios, una lista de diez elementos. Todo funciona porque todo es pequeño. Pero un día el catálogo tiene diez mil productos, el formulario de registro recibe cien envíos al día y los usuarios se quejan de que "la página se siente lenta". En ese momento, las habilidades básicas ya no alcanzan. Necesitas tres cosas: URLs dinámicas que lleven a contenido específico, formularios que guíen al usuario en lugar de frustrarlo, y código que no desperdicie recursos haciendo cálculos repetidos.

Empecemos con las URLs. Imagina que entras a una tienda en línea y ves un producto que te encanta. Quieres compartirlo con un amigo. ¿Qué le mandas? Le mandas la dirección de la página del producto: algo como `tienda.com/producto/42`. Esa dirección es dinámica: la estructura es la misma para todos los productos, pero el número cambia según el producto. Si tu app solo tuviera rutas estáticas, tendrías que crear una página separada para cada producto, lo cual es imposible cuando tienes miles. Las rutas dinámicas resuelven esto con un truco elegante: defines una sola ruta con un parámetro, como `/producto/:id`, y React Router te entrega el valor de ese parámetro con el hook `useParams`. Con ese id puedes buscar el producto en tu lista, en tu base de datos o en una API, y mostrar su detalle. Una sola página sirve para infinitos productos. Además, estas URLs tienen otra ventaja enorme: se pueden compartir, se pueden guardar en marcadores y funcionan si el usuario recarga la página. Eso es lo que hace que una app se sienta real y no como un juguete.

El segundo tema son los formularios. Un formulario básico guarda lo que escribe el usuario y lo manda. Eso está bien para un ejemplo de clase. Pero un formulario profesional hace mucho más: valida los datos antes de enviarlos, muestra mensajes de error claros junto a cada campo, marca qué campos ya tocó el usuario para no regañarlo antes de tiempo, confirma que las dos contraseñas coinciden, verifica que el correo tenga un formato válido y deshabilita el botón de enviar mientras haya errores. Piensa en la diferencia desde el punto de vista del usuario: en un formulario malo escribes tu correo, le das enviar y la página simplemente no hace nada, o peor, te manda a otra página con un error genérico. En un formulario bueno, en cuanto sales del campo del correo aparece un mensajito: "Escribe un correo válido". Sabes

exactamente qué está mal y cómo arreglarlo, sin adivinar. Esa retroalimentación inmediata es la diferencia entre un usuario que completa su registro y uno que cierra la pestaña. En esta unidad vas a aprender el patrón profesional completo: un estado para los valores, un estado para los errores, una función `validar()` que revisa todo y devuelve los errores encontrados, y renderizado condicional para mostrar cada mensaje solo cuando corresponde.

El tercer tema es el rendimiento, y aquí hay una verdad que casi nadie te dice al principio: la mayoría de las veces no necesitas optimizar nada. React ya es rápido. Pero hay casos específicos donde sí importa, y es mejor que los reconozcas. Imagina una lista con cinco mil productos y un campo de búsqueda. Cada vez que el usuario escribe una letra, el componente se vuelve a renderizar, y si tu código filtra los cinco mil productos en cada renderizado, la app se va a sentir lenta y trabada. El hook `useMemo` resuelve esto guardando en memoria el resultado de un cálculo pesado y reutilizándolo mientras sus dependencias no cambien. Es cómo hacer la tarea una vez y guardar la respuesta en una libreta: si te preguntan lo mismo de nuevo, lees la libreta en lugar de resolver todo otra vez. Pero ojo: `useMemo` también tiene un costo, y usarlo en todos lados "por si acaso" es un error clásico de principiante que termina complicando el código sin ningún beneficio real. Vas a aprender exactamente cuándo sí usarlo y cuándo no.

Estas tres habilidades están conectadas por una idea común: pensar en el usuario final. Las rutas dinámicas le dan URLs que puede compartir. Los formularios con validación le dan una experiencia sin frustración. La optimización le da una app que responde rápido. Juntas convierten tu prototipo en un producto.

Vale la pena detenerse un momento en por qué estos tres temas van juntos en una misma unidad y no separados. En la práctica, casi nunca aparecen aislados. Piensa en el flujo de una tienda real: el usuario llega a `/producto/42` (ruta dinámica), decide comprarlo y llena el formulario de pago (validación), y mientras tanto el catálogo con miles de productos se filtra sin trabarse (`useMemo`). Una sola página de checkout usa las tres cosas a la vez. Si las aprendieras por separado, tardarías más en ver cómo se combinan; aquí las ves como lo que son: las tres patas de una app que se siente profesional.

También hay una progresión de mentalidad. Las rutas dinámicas te enseñan a pensar en la URL como parte de la interfaz: la dirección ya no es un detalle técnico, es algo que el usuario ve, comparte y guarda. Los formularios avanzados te enseñan a pensar en los estados intermedios: no solo "vacío" y "enviado", sino "tocado", "con error", "válido", "cargando". Y `useMemo` te enseña a pensar en el costo de tu código: cada línea que escribes se ejecuta muchas veces, y vale la pena saber cuáles cuestan caro. Son tres formas de madurar como desarrollador, y las tres caben en esta unidad.

Al terminar esta unidad vas a poder construir páginas de detalle que lean su contenido desde la URL, formularios de registro y login con validación profesional campo por campo, y listas grandes que se filtran sin trabarse. Son exactamente las piezas que usan las apps reales todos los días. Vamos a ello, paso a paso, sin prisa y sin saltarnos nada.

Explicación detallada

useParams: leer la dirección para decidir qué mostrar

Hasta ahora las rutas que conoces son estáticas: /inicio, /contacto, /productos. Cada una muestra siempre lo mismo. Eso funciona para páginas fijas, pero el contenido real de una app casi nunca es fijo: hay cientos de productos, decenas de artículos de un blog, miles de usuarios. No puedes escribir una ruta para cada uno. La solución son las rutas dinámicas.

Una ruta dinámica tiene una parte fija y una parte variable. La parte variable se marca con dos puntos en la definición de la ruta. Mira este ejemplo:

```
<Route path="/producto/:id" element={<DetalleProducto />} />
```

Esa ruta coincide con /producto/1, /producto/42, /producto/999 y cualquier otra dirección que tenga ese formato. La parte después de /producto/ se captura como un parámetro llamado id. Es como un molde: la forma es la misma, pero el contenido cambia.

Dentro del componente DetalleProducto, el hook useParams te entrega esos parámetros en un objeto:

```
import { useParams } from 'react-router-dom';

function DetalleProducto() {
  const { id } = useParams();
  return <h2>Estás viendo el producto {id}</h2>;
}
```

Si el usuario visita /producto/42, la variable id vale "42". Así de directo. Pero aquí viene el primer detalle importante que causa muchos errores: el valor que te da useParams siempre es un string, siempre texto. Aunque la URL diga /producto/42, el id no es el número 42, es el texto "42". Esto importa muchísimo cuando lo comparas con datos.

Imagina que tienes un arreglo de productos donde cada producto tiene un id numérico:

```
const productos = [
  { id: 1, nombre: 'Teclado mecánico', precio: 899 },
  { id: 2, nombre: 'Mouse inalámbrico', precio: 449 },
];
```

Si haces `productos.find(p => p.id === id)`, la comparación `1 === "1"` da `false` en JavaScript porque el operador `===` no convierte tipos. El producto nunca se encuentra y tu página muestra un error aunque el producto sí exista. Este es uno de los errores más comunes con `useParams` y es facilísimo de cometer. La solución es convertir el parámetro a número antes de comparar:

```
const { id } = useParams();
const producto = productos.find(p => p.id === Number(id));
```

`Number("42")` devuelve 42, y ahora sí `42 === 42` es `true`. Alternativas válidas son `parseInt(id, 10)` o el operador unario `+id`, que también convierte a número. Elige la que te resulte más legible, pero haz la conversión siempre que tus datos usen ids numéricos. Si tus ids son texto (como un slug "teclado-mecanico"), entonces no necesitas convertir nada.

Otro detalle: ¿qué pasa si el id no existe? Si el usuario visita `/producto/9999` y no hay ningún producto con ese id, `find` devuelve `undefined`. Si intentas leer `producto.nombre` cuando `producto` es `undefined`, tu app se rompe con un error en pantalla. Un producto profesional nunca deja que eso pase: muestra una página de "no encontrado". El patrón es este:

```
const { id } = useParams();
const producto = productos.find(p => p.id === Number(id));

if (!producto) {
  return <h2>Producto no encontrado</h2>;
}

return (
  <div>
    <h2>{producto.nombre}</h2>
    <p>Precio: ${producto.precio}</p>
  </div>
);
```

Ese `return` temprano es una buena práctica que vas a ver en todas las apps serias: primero manejas el caso malo, y el resto del componente solo se ejecuta cuando los datos existen. Así evitas una cascada de verificaciones.

En una app real, los datos casi nunca están en un arreglo local: vienen de un servidor. El patrón con fetch es igual de simple y usa el useEffect que ya conoces:

```
function DetalleProducto() {
  const { id } = useParams();
  const [producto, setProducto] = useState(null);
  const [cargando, setCargando] = useState(true);

  useEffect(() => {
    setCargando(true);
    fetch('https://api.ejemplo.com/productos/' + id)
      .then(res => res.json())
      .then(datos => {
        setProducto(datos);
        setCargando(false);
      });
  }, [id]);

  if (cargando) return <p>Cargando...</p>;
  if (!producto) return <p>Producto no encontrado</p>;

  return <h2>{producto.nombre}</h2>;
}
```

Fíjate en el arreglo de dependencias [id]: si el usuario navega de /producto/1 a /producto/2 sin salir de la página (por ejemplo, con un enlace de "producto relacionado"), el componente no se desmonta, solo cambia el parámetro. Sin [id] en las dependencias, el efecto no se volvería a ejecutar y seguirías viendo el producto 1. Este es otro error clásico: olvidar que el parámetro puede cambiar mientras el componente sigue montado.

Las rutas dinámicas también sirven para navegar hacia ellas. Con el componente Link construyes URLs dinámicas concatenando el id:

```
{productos.map(p => (
  <Link key={p.id} to={'/producto/' + p.id}>
    {p.nombre}
  </Link>
))}
```

Cada enlace lleva a la página de detalle de ese producto. Este es el flujo completo de una app real: una lista con enlaces, una ruta dinámica que captura el id, un componente que lee el parámetro y busca los datos. Lista -> detalle -> dato. Una vez que dominas este flujo, puedes construir catálogos, blogs, perfiles de usuario y paneles de administración.

Puedes tener más de un parámetro en la misma ruta. Por ejemplo, /categoría/:categoriald/producto/:productold te da dos valores: `const { categoriald, productold } = useParams();`. Úsalo cuando la jerarquía lo pida, pero no abuses: rutas muy anidadas se vuelven difíciles de mantener.

Analogía para que no se te olvide: una ruta dinámica es como un casillero de paquetería. Todos los casilleros se ven igual por fuera, pero cada uno tiene un número distinto. Tú no construyes un edificio nuevo para cada paquete: construyes un edificio con muchos casilleros y usas el número para abrir el correcto. `useParams` es la llave que lee el número del casillero.

Formularios avanzados: validar antes de enviar y guiar al usuario

Un formulario básico hace dos cosas: guarda lo que escribe el usuario y lo envía. Un formulario profesional hace eso y además valida, informa y previene errores. La validación es el proceso de revisar los datos antes de enviarlos y decirle al usuario qué está mal, dónde está mal y cómo arreglarlo.

Hay dos decisiones de diseño al inicio: ¿un estado por campo o un solo objeto con todos los campos? Con dos o tres campos, estados separados son simples:

```
const [nombre, setNombre] = useState('');
const [email, setEmail] = useState('');
```

Pero cuando el formulario crece a seis, ocho o diez campos, tener diez `useState` se vuelve un enredo. La alternativa profesional es un solo objeto:

```
const [valores, setValores] = useState({
  nombre: '',
  email: '',
  password: '',
  confirmar: ''
});
```

Y una función genérica que actualiza cualquier campo usando su atributo `name`:

```
function manejarCambio(e) {
  const { name, value } = e.target;
  setValores({ ...valores, [name]: value });
}
```

Cada input lleva name="email", name="password", etc., y todos comparten el mismo onChange. Esto escala muy bien: agregar un campo nuevo es agregar una línea al objeto inicial y un input en el JSX. Nada más.

Ahora la validación. La forma profesional es tener una función validar() que recibe los valores y devuelve un objeto con los errores encontrados. Si un campo está bien, no aparece en el objeto; si está mal, aparece con su mensaje:

```
function validar(valores) {
  const errores = {};

  if (!valores.nombre.trim()) {
    errores.nombre = 'El nombre es obligatorio';
  }

  if (!valores.email) {
    errores.email = 'El correo es obligatorio';
  } else if (!/^\S+@\S+\.\S+/.test(valores.email)) {
    errores.email = 'Escribe un correo válido';
  }

  if (!valores.password) {
    errores.password = 'La contraseña es obligatoria';
  } else if (valores.password.length < 8) {
    errores.password = 'Mínimo 8 caracteres';
  }

  if (valores.confirmar !== valores.password) {
    errores.confirmar = 'Las contraseñas no coinciden';
  }

  return errores;
}
```

Fíjate en varias cosas. Primero, trim() en el nombre: sin eso, un usuario podría escribir solo espacios y pasaría la validación. Segundo, la expresión regular /^\S+@\S+\.\S+/. es una validación de email simple pero suficiente: exige algo antes del @, algo después, un punto y algo después del punto. No es perfecta según los estándares más estrictos, pero para un

formulario real funciona bien y es fácil de entender. Tercero, la validación de confirmar contra password: es el caso clásico de validación cruzada entre campos, algo que solo puedes hacer si tienes todos los valores juntos (otra razón para preferir el objeto único).

¿Dónde se usa validar()? En el submit, antes de enviar:

```
function manejarEnvio(e) {
  e.preventDefault();
  const erroresEncontrados = validar(valores);
  setErrores(erroresEncontrados);

  if (Object.keys(erroresEncontrados).length === 0) {
    // todo válido: enviar datos
    console.log('Enviando:', valores);
  }
}
```

Si el objeto de errores está vacío, todo está bien y se envían los datos. Si tiene claves, hay errores y no se envía nada. Simple y predecible.

Pero validar solo al enviar no es suficiente para una buena experiencia. El usuario quiere saber qué está mal mientras escribe, no después de darle al botón. Por eso los errores se muestran junto a cada campo con renderizado condicional:

```
<div>
  <input name="email" value={valores.email} onChange={manejarCambio} />
  {errores.email && <p className="error">{errores.email}</p>}
</div>
```

Si errores.email existe, se muestra el mensaje; si no, no se muestra nada. Cada campo tiene su propio mensaje, justo debajo de él. El usuario sabe exactamente qué arreglar.

Aquí aparece un problema de experiencia: si muestras los errores desde el primer momento, el usuario ve el formulario lleno de mensajes en rojo antes de escribir nada. Eso se siente agresivo. La solución profesional es el concepto de touched (tocado): llevas registro de qué campos el usuario ya visitó, y solo muestras el error de un campo si ya lo tocó o si ya intentó enviar el formulario.

```
const [tocados, setTocados] = useState({});

function manejarBlur(e) {
  const { name } = e.target;
  setTocados({ ...tocados, [name]: true });
}
```

El evento onBlur se dispara cuando el usuario sale de un campo. Entonces el mensaje se muestra así:

```
{tocados.email && errores.email && <p className="error">{errores.email}</p>}
```

Para que la validación sea en vivo, recalculas los errores cada vez que cambian los valores, no solo al enviar. Puedes hacerlo con un useEffect:

```
useEffect(() => {
  setErrores(validar(valores));
}, [valores]);
```

O calculando los errores directamente durante el renderizado (sin guardarlos en estado), que es aún más simple y evita un estado duplicado:

```
const errores = validar(valores);
```

Esta última forma es la favorita de muchos desarrolladores experimentados: los errores son un valor derivado de los valores, así que no necesitan su propio estado. Se calculan solos en cada renderizado. Menos estado, menos bugs.

Otra buena práctica: deshabilitar el botón de enviar mientras haya errores.

```
const hayErrores = Object.keys(errores).length > 0;
<button type="submit" disabled={hayErrores}>Registrarse</button>
```

Esto previene envíos inválidos desde la interfaz. Ojo: esto no sustituye la validación en el submit, porque el usuario podría tener JavaScript desactivado o manipular el HTML. La regla de oro es: la validación del cliente es para la experiencia del usuario; la validación real de seguridad siempre vive en el servidor. Nunca confíes solo en lo que valida el navegador.

Validaciones comunes que vas a usar una y otra vez: campo requerido (verificar que no esté vacío después de trim), formato de email (la regex simple de arriba), longitud mínima

(password.length < 8), longitud máxima (una biografía de máximo 200 caracteres), solo números (para teléfono o código postal), rango numérico (cantidad entre 1 y 10) y confirmación de contraseña (comparar dos campos). Con esas siete cubres el 95% de los formularios reales.

Errores comunes en formularios: validar solo al enviar sin dar feedback en vivo (el usuario adivina qué está mal), mostrar errores antes de que el usuario toque nada (experiencia agresiva), guardar cada error en su propio estado en lugar de un objeto (código repetitivo), olvidar el trim y aceptar espacios en blanco como válidos, y no prevenir el comportamiento por defecto del formulario con e.preventDefault(), lo que recarga la página y borra todo.

Analogía: un buen formulario es como un maestro paciente que revisa tu examen contigo pregunta por pregunta y te dice exactamente dónde te equivocaste. Un mal formulario es como un maestro que solo te dice "reprobado" sin explicarte nada. Sé el maestro paciente.

useMemo: no repitas cálculos que ya hiciste

Memoización es una palabra elegante para una idea simple: guardar el resultado de un cálculo costoso para no repetirlo. Imagina que tienes que sumar los números del 1 al 1000. La primera vez haces la suma completa. Si alguien te pregunta el resultado otra vez, no sumas de nuevo: recuerdas que ya lo calculaste. Eso es memoización.

En React, los componentes se vuelven a renderizar por muchas razones: cambió un estado, el componente padre se renderizó, etc. Si dentro del componente hay un cálculo pesado, ese cálculo se repite en cada renderizado aunque los datos no hayan cambiado. useMemo evita eso.

La sintaxis:

```
const total = useMemo(() => suma(lista), [lista]);
```

Tiene dos partes: una función que hace el cálculo y un arreglo de dependencias. React ejecuta la función la primera vez y guarda el resultado. En los siguientes renderizados, revisa las dependencias: si [lista] no cambió, devuelve el resultado guardado sin ejecutar la función. Si lista sí cambió, vuelve a calcular y guarda el nuevo resultado. Es el mismo concepto de dependencias que ya conoces de useEffect.

El ejemplo clásico es filtrar una lista grande:

```
function Catalogo({ productos }) {
  const [busqueda, setBusqueda] = useState('');

  const filtrados = useMemo(() => {
    console.log('Filtrando...');
    return productos.filter(p =>
      p.nombre.toLowerCase().includes(busqueda.toLowerCase())
    );
  }, [productos, busqueda]);

  return (
    <div>
      <input value={busqueda} onChange={e => setBusqueda(e.target.value)} />
      {filtrados.map(p => <p key={p.id}>{p.nombre}</p>)}
    </div>
  );
}
```

Sin `useMemo`, cada vez que el componente se renderiza por cualquier motivo (por ejemplo, porque el padre cambió un estado no relacionado), se filtran todos los productos otra vez. Con `useMemo`, el filtrado solo se repite si productos o búsqueda cambian. Si el componente se renderiza por otra razón, se reutiliza el resultado anterior. Con una lista de diez elementos no notas la diferencia; con cinco mil, la diferencia es entre una app fluida y una app trabada.

Ahora la parte importante que casi nadie enseña bien: cuándo NO usar `useMemo`. La mayoría de los cálculos en una app normal son baratos: sumar diez números, filtrar veinte elementos, concatenar strings. Para eso, `useMemo` no solo es innecesario, sino que añade costó: React tiene que guardar el valor, comparar las dependencias en cada renderizado y mantener esa lógica. Es como contratar un guardia de seguridad para cuidar una caja vacía: el guardia cuesta más que lo que protege.

Regla práctica: usa `useMemo` cuando el cálculo sea genuinamente pesado (miles de elementos, operaciones matemáticas complejas, transformaciones de datos grandes) o cuando el valor se use como dependencia de otro hook y necesite una referencia estable. Si no estás seguro de que el cálculo sea pesado, probablemente no lo es: no lo optimices. La optimización prematura es uno de los errores más comunes y más caros en tiempo de desarrollo: complica el código, introduce bugs sutiles (dependencias olvidadas) y no mejora nada medible.

Hablando de bugs sutiles: el error más común con `useMemo` es poner dependencias incorrectas o incompletas. Si tu cálculo usa búsqueda pero no la pones en el arreglo, React reutilizará un resultado viejo cuando búsqueda cambie, y la lista no se actualizará. El síntoma

es confuso: "cambió el texto pero la lista no se filtra". La regla es la misma que en `useEffect`: todas las variables que usa el cálculo y que pueden cambiar deben estar en las dependencias.

Dos menciones breves para que conozcas el panorama completo, sin profundizar porque eso viene después: `React.memo` es un componente de orden superior que evita que un componente se vuelva a renderizar si sus props no cambiaron. Es memoización pero a nivel de componente completo, no de un valor. Y `useCallback` es como `useMemo` pero para funciones: guarda la función misma en lugar del resultado de ejecutarla, útil cuando pasas callbacks a componentes optimizados con `React.memo`. La idea general es la misma en los tres: no repitas trabajo que ya hiciste.

Analogía final: `useMemo` es como cocinar para toda la semana el domingo. Cocinas una vez (cálculo pesado) y guardas los tupperes en el refri (memoria). Cada día solo calientas lo que necesitas (reutilizas el resultado). Solo vuelves a cocinar si se acabó la comida o si cambió el menú (cambiaron las dependencias). Pero no tiene sentido cocinar para toda la semana si solo vas a comer una quesadilla: el esfuerzo no se justifica.

Buenas prácticas de la unidad

Uno: convierte siempre los parámetros de `useParams` al tipo que necesites antes de comparar. Dos: maneja el caso de "no encontrado" con un return temprano, nunca dejes que la app se rompa con un id inválido. Tres: incluye el parámetro en las dependencias del `useEffect` que carga datos. Cuatro: en formularios, un objeto para valores y otro para errores escala mejor que estados separados. Cinco: muestra errores solo en campos tocados o después del primer intentó de envío. Seis: calcula los errores como valor derivado cuando puedas, en lugar de duplicarlos en estado. Siete: deshabilita el submit si hay errores, pero valida de nuevo al enviar. Ocho: usa `useMemo` solo para cálculos pesados medibles, no por costumbre. Nueve: revisa dos veces las dependencias de `useMemo`. Diez: la validación del navegador es para la experiencia; la validación de verdad vive en el servidor.

Más a fondo: navegación programática y parámetros en la práctica

`useParams` no vive solo: se combina con otros hooks de React Router para flujos completos. Uno muy común es redirigir cuando el id no existe, en lugar de mostrar un mensaje dentro de la misma página. Para eso sirve `useNavigate`:


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

function Detalle() {
  const { id } = useParams();
  const navigate = useNavigate();
  const producto = productos.find(p => p.id === Number(id));

  useEffect(() => {
    if (producto === undefined) {
      // damos un momento para que el estado inicial se resuelva
    }
  }, []);
}
```

En realidad, el patrón más limpio para redirigir es usar el componente Navigate:

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

function Detalle() {
  const { id } = useParams();
  const producto = productos.find(p => p.id === Number(id));

  if (!producto) {
    return <Navigate to="/no-encontrado" replace />;
  }
  // ...
}
```

Con replace, la redirección no deja la URL mala en el historial: si el usuario le da "atrás", no vuelve a la página inexistente. Define una ruta /no-encontrado con un diseño amable y un enlace de regreso. Las apps profesionales casi nunca muestran una pantalla en blanco con un error técnico; siempre hay una página 404 diseñada.

Otro compañero frecuente de useParams es useSearchParams, que lee los query strings: la parte de la URL después del ?. En /productos?orden=precio&página=2, los parámetros de ruta son distintos de los query strings. useParams lee /producto/:id; useSearchParams lee ?orden=precio. No los confundas: si el dato identifica el recurso (qué producto), va en la ruta; si es un filtro o una opción de vista (cómo ordenarlo), va en el query string.

También puedes validar el formato del parámetro antes de buscar. Si tus ids son numéricos y alguien visita /producto/hola, Number("hola") da NaN y find no encuentra nada, así que cae en el "no encontrado". Eso está bien. Pero si quieres ser explícito:

```
const idNumerico = Number(id);
if (Number.isNaN(idNumerico)) {
  return <Navigate to="/no-encontrado" replace />;
}
```

Esto deja claro en el código que un id no numérico es inválido, y evita que un NaN silencioso viaje por tu lógica.

Cuando los datos vienen de una API, combina useParams con un estado de error separado del estado de "no encontrado". No es lo mismo que el servidor diga "no existe" a que la red falle:

```
const [producto, setProducto] = useState(null);
const [estado, setEstado] = useState('cargando'); // cargando | listo |
no-encontrado | error

useEffect(() => {
  setEstado('cargando');
  fetch('https://api.ejemplo.com/productos/' + id)
    .then(res => {
      if (res.status === 404) {
        setEstado('no-encontrado');
        return null;
      }
      if (!res.ok) throw new Error('Fallo la red');
      return res.json();
    })
    .then(datos => {
      if (datos) {
        setProducto(datos);
        setEstado('listo');
      }
    })
    .catch(() => setEstado('error'));
}, [id]);

if (estado === 'cargando') return <p>Cargando producto...</p>;
if (estado === 'no-encontrado') return <Navigate to="/no-encontrado" replace />;
if (estado === 'error') return <p>Algo falló. Intenta de nuevo.</p>;
```

Cuatro estados, cuatro pantallas. El usuario siempre sabe qué pasa: esperando, listo, no existe o falló. Esa es la diferencia entre una app que se siente profesional y una que deja una pantalla en blanco cuando algo sale mal.

Más a fondo: validación como función pura y reutilizable

La función `validar()` tiene una propiedad valiosa: es pura. Recibe valores y devuelve errores, sin tocar estado ni efectos. Eso significa que la puedes probar sin renderizar nada, reutilizar en varios formularios y hasta compartir entre el cliente y el servidor si usas JavaScript en ambos lados. Cuando un formulario crece, conviene sacarla del componente a su propio archivo:

```
// validaciones.js
export function validarRegistro(v) { /* ... */ }
export function validarLogin(v) { /* ... */ }
```

Y en el componente solo la importas. Otra mejora natural es crear un hook personalizado que encapsule todo el manejo del formulario:

```
function useFormulario(valoresIniciales, funcionValidar) {
  const [valores, setValores] = useState(valoresIniciales);
  const [tocados, setTocados] = useState({});
  const errores = funcionValidar(valores);

  function cambio(e) {
    const { name, value } = e.target;
    setValores({ ...valores, [name]: value });
  }
  function blur(e) {
    setTocados({ ...tocados, [e.target.name]: true });
  }
  return { valores, errores, tocados, cambio, blur };
}
```

Con esto, cada formulario nuevo necesita solo sus valores iniciales y su función de validación; todo el mecanismo se reutiliza. Es el mismo patrón que usan las librerías de formularios por dentro, solo que simplificado. No necesitas una librería para formularios medianos: este hook casero cubre la mayoría de los casos.

Hablemos de validación asíncrona, porque en apps reales aparece: verificar que un nombre de usuario no esté ya registrado requiere preguntarle al servidor. No puedes hacerlo dentro de `validar()` porque esa función es síncrona. El patrón es un estado separado:

```
const [usuarioDisponible, setUsuarioDisponible] = useState(null);

useEffect(() => {
  if (valores.usuario.length < 4) return;
  const t = setTimeout(() => {
    fetch('https://api.ejemplo.com/usuarios/existe?nombre=' + valores.usuario)
      .then(r => r.json())
      .then(d => setUsuarioDisponible(!d.existe));
  }, 500);
  return () => clearTimeout(t);
}, [valores.usuario]);
```

El `setTimeout` de 500 ms es un `debounce`: espera a que el usuario deje de escribir antes de preguntar al servidor, para no saturarlo con una petición por letra. El `return () => clearTimeout(t)` cancela la petición pendiente si el usuario sigue escribiendo. Es un patrón avanzado pero muy común; guárdalo en tu caja de herramientas.

Por último, accesibilidad: asocia cada mensaje de error con su campo usando `aria-describedby`, y usa el atributo `required` en el HTML como primera línea de defensa. La validación con JavaScript mejora la experiencia, pero el HTML semántico ayuda a los lectores de pantalla y funciona incluso si tu script falla.

Más a fondo: `useMemo` y la estabilidad de referencias

Hay un segundo uso de `useMemo` menos obvio pero igual de importante: dar estabilidad a una referencia. En JavaScript, cada vez que ejecutas `{ a: 1 }` creas un objeto nuevo, aunque tenga el mismo contenido. Si un componente crea un objeto o arreglo dentro del renderizado y lo pasa como dependencia a un `useEffect`, el efecto se ejecuta en cada renderizado porque la referencia siempre es "nueva":

```
// problema: filtros es un objeto nuevo en cada renderizado
const filtros = { texto, categoria };
useEffect(() => {
  buscar(filtros);
}, [filtros]); // se ejecuta siempre, aunque texto y categoria no cambien
```

La solución es memoizar el objeto:

```
const filtros = useMemo(() => ({ texto, categoria }), [texto, categoria]);
useEffect(() => {
  buscar(filtros);
}, [filtros]); // solo se ejecuta si texto o categoria cambian
```

Ahora filtros conserva su referencia mientras texto y categoría no cambien, y el efecto solo corre cuando debe. Este es un uso legítimo y común de useMemo que no tiene nada que ver con cálculos pesados: se trata de identidad, no de velocidad.

¿Cómo sabes si un cálculo es lo bastante pesado para justificar useMemo? Mide. Las herramientas de desarrollo de React incluyen un Profiler que muestra cuánto tarda cada renderizado. Si un componente tarda más de unos 16 milisegundos en renderizar (el presupuesto para 60 cuadros por segundo), el usuario puede notar el tirón. Optimiza lo que el Profiler señale, no lo que tu intuición sospeche. La intuición sobre rendimiento es famosa por equivocarse.

Y un recordatorio sobre React.memo y useCallback, los parientes que mencionamos antes. React.memo envuelve un componente y evita renderizarlo si sus props son iguales a las anteriores. Pero hay un detalle: si le pasas una función definida dentro del padre, esa función es nueva en cada renderizado, así que React.memo no sirve de nada. Ahí entra useCallback, que memoiza la función misma:

```
const manejarClic = useCallback(() => {
  console.log('clic en', id);
}, [id]);
```

useCallback(fn, deps) es equivalente a useMemo(() => fn, deps): guarda la función y solo crea una nueva si las dependencias cambian. El trío completo trabaja junto: useCallback estabiliza la función, React.memo evita el renderizado innecesario del hijo, y useMemo evita recalcular valores. No necesitas dominarlos hoy a fondo, pero ya sabes qué buscar cuando los necesites.

Dónde se usa esto en apps reales

Para que no se quede en teoría, veamos dónde aparece cada tema en productos que usas todos los días.

Las rutas dinámicas están en todas partes: cuando abres un video de YouTube la URL lleva un identificador único del video; cuando ves un producto en Amazon la dirección incluye su código; cuando lees un artículo de noticias la URL lleva el slug del artículo. Todos esos sitios definen una sola plantilla de página y la llenan según el parámetro. Además, las rutas

dinámicas hacen posible el SEO: Google indexa cada URL como una página distinta, así que `/producto/teclado-mecanico` puede aparecer en los resultados de búsqueda. Sin rutas dinámicas no hay SEO para contenido generado por usuarios, y sin SEO muchos negocios no existen.

Los formularios con validación deciden si un negocio gana o pierde clientes. Los estudios de experiencia de usuario muestran que los formularios largos y confusos son una de las principales causas de abandono en registros y pagos. Cada campo que validas con un mensaje claro reduce la fricción; cada error críptico la aumenta. Las tiendas en línea invierten mucho en esto: validación de tarjeta en vivo, formato automático del número telefónico, mensajes específicos por campo. No es decoración, es dinero.

El `useMemo` aparece en los lugares donde hay datos masivos: paneles de administración con miles de filas, editores con vista previa en vivo, gráficas que recalculan con cada filtro, y buscadores sobre catálogos enormes. También aparece en un lugar inesperado: las apps móviles hechas con React Native, donde el hardware es más limitado y cada cálculo repetido se nota más en la batería y en la fluidez.

Fíjate que los tres temas comparten una filosofía: respetar los recursos del usuario. Las rutas dinámicas respetan su tiempo (puede compartir y volver a lo que veía). La validación respeta su paciencia (le dice qué está mal sin hacerlo adivinar). La memoización respeta su dispositivo (no lo pone a calcular lo mismo cien veces). Programar pensando en la persona del otro lado de la pantalla es lo que distingue a un desarrollador que entrega features de uno que construye productos.

Errores comunes y cómo depurarlos

Repasemos los errores más frecuentes de la unidad con su síntoma, su causa y su arreglo, porque reconocerlos rápido es una habilidad en sí misma.

Error 1: comparar el id string con números. Síntoma: la página de detalle siempre muestra "no encontrado" aunque el id existe. Causa: `productos.find(p => p.id === id)` compara número con string y `===` nunca convierte tipos. Arreglo: `const producto = productos.find(p => p.id === Number(id));`. Cómo depurarlo: pon un `console.log(typeof id, id)` justo después de `useParams` y verás "string" en la consola. Ese log de dos segundos te ahorra media hora.

Error 2: olvidar el parámetro en las dependencias del `useEffect`. Síntoma: navegas de `/producto/1` a `/producto/2` y la página sigue mostrando el producto 1. Causa: el componente no se desmonta al cambiar solo el parámetro, así que el efecto con `[]` no se vuelve a ejecutar. Arreglo: pon `[id]` en las dependencias. Regla mental: si el efecto usa un valor que puede cambiar, ese valor va en el arreglo. Sin excepciones.

Error 3: validar solo al enviar. Síntoma: el usuario le da al botón, nada pasa (o todo se borra) y no sabe por qué. Causa: los errores se calculan en el submit pero nunca se muestran junto a los campos. Arreglo: calcula los errores en cada renderizado y muéstralos con renderizado condicional debajo de cada input.

Error 4: mostrar errores antes de tiempo. Síntoma: el formulario aparece lleno de mensajes rojos antes de escribir nada y el usuario se siente regañado. Causa: mostrar errores sin revisar touched. Arreglo: el estado tocados con onBlur, y mostrar cada error solo si el campo fue tocado o ya se intentó enviar.

Error 5: dependencias incompletas en useMemo. Síntoma: cambias el texto del buscador y la lista no se actualiza, o se actualiza con un valor viejo. Causa: el cálculo usa una variable que no está en el arreglo de dependencias, así que React reutiliza el resultado guardado. Arreglo: revisa qué variables usa la función de useMemo y ponlas todas en el arreglo. Si el linter de React te marca una advertencia de dependencias, hazle caso: casi siempre tiene razón.

Error 6: optimizar prematuramente. Síntoma: el código está lleno de useMemo y useCallback, es difícil de leer y nadie sabe si realmente algo mejoró. Causa: aplicar memoización "por si acaso". Arreglo: quita los useMemo que no protejan un cálculo mediblemente pesado y mide con el Profiler antes de volver a agregarlos. El código simple que funciona siempre le gana al código "optimizado" que nadie entiende.

Error 7: olvidar e.preventDefault() en el submit. Síntoma: al enviar el formulario la página se recarga y se pierde todo. Causa: el comportamiento por defecto de un form HTML es recargar la página. Arreglo: primera línea del manejador de envío: e.preventDefault();. Es un error de principiante que hasta los intermedios cometen cuando tienen prisa.

Error 8: aceptar espacios en blanco como válidos. Síntoma: un usuario se registra con el nombre " " y tu base de datos queda con basura. Causa: validar !valores.nombre sin trim. Arreglo: usa valores.nombre.trim() en la validación. Tres letras que evitan datos sucios.

Cuando algo falle y no sepas por qué, sigue este orden: primero lee el mensaje de error completo (React suele decirte el archivo y la línea), segundo agrega console.log en los puntos clave (qué vale el parámetro, qué devuelve find, qué contiene el objeto de errores), tercero revisa tipos con typeof, y cuarto simplifica: comenta la mitad del código y ve si el problema sigue. Depurar es un proceso, no un don.

Ejemplos paso a paso

Los siguientes ejemplos están diseñados para que los escribas tú mismo en tu editor, no solo para que los leas. La diferencia entre leer código y escribirlo es enorme: al escribirlo te topas con los errores de dedo, las importaciones olvidadas y las llaves mal cerradas, que es justo donde se aprende de verdad. Te recomiendo este método: primero lee el ejemplo completo para entender la idea, luego ciérralo y trata de escribirlo de memoria, y al final compara con el original. Si te atorras, vuelve a leer solo la parte donde te atoraste. Cinco ejemplos te esperan, cada uno combinando los temas de la unidad de una forma distinta.

Ejemplo 1: detalle de producto con useParams

Vamos a construir la página de detalle de un producto. Primero la lista con enlaces, luego la ruta dinámica y al final el componente que lee el parámetro. Este es el flujo completo lista -> detalle.

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

const productos = [
  { id: 1, nombre: 'Teclado mecánico', precio: 899, desc: 'Switches azules, retroiluminado.' },
  { id: 2, nombre: 'Mouse inalámbrico', precio: 449, desc: 'Sensor óptico, 6 botones.' },
  { id: 3, nombre: 'Monitor 24 pulgadas', precio: 3299, desc: 'Full HD, 75 Hz.' },
];

function Lista() {
  return (
    <div>
      <h2>Productos</h2>
      {productos.map(p => (
        <p key={p.id}>
          <Link to={'/producto/' + p.id}>{p.nombre}</Link>
          {' - $' + p.precio}
        </p>
      ))}
    </div>
  );
}
```

Línea por línea: importamos lo necesario de react-router-dom. El arreglo productos simula una base de datos con ids numéricos. En Lista recorreremos el arreglo con map y por cada producto

creamos un Link. El atributo to se construye concatenando '/producto/' con el id, así que el primer enlace apunta a /producto/1, el segundo a /producto/2, etc. El key={p.id} es obligatorio en listas.

```
function Detalle() {
  const { id } = useParams();
  const producto = productos.find(p => p.id === Number(id));

  if (!producto) {
    return (
      <div>
        <h2>Producto no encontrado</h2>
        <Link to="/">Volver a la lista</Link>
      </div>
    );
  }

  return (
    <div>
      <h2>{producto.nombre}</h2>
      <p>{producto.desc}</p>
      <p>Precio: ${producto.precio}</p>
      <Link to="/">Volver a la lista</Link>
    </div>
  );
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Lista />} />
        <Route path="/producto/:id" element={<Detalle />} />
      </Routes>
    </BrowserRouter>
  );
}
```

En Detalle: useParams nos da el objeto con los parámetros y desestructuramos id. Como viene como string, lo convertimos con Number(id) antes de compararlo con los ids numéricos del arreglo. find devuelve el producto o undefined. El if (!producto) es el return temprano: si el id no existe (por ejemplo /producto/99), mostramos un mensaje amable y un enlace de regreso en lugar de romper la app. Si el producto existe, mostramos sus datos. En App definimos las

dos rutas: la lista en / y el detalle en /producto/:id. Prueba visitar /producto/2 y luego cambiar la URL a /producto/99 para ver los dos casos.

Ejemplo 2: formulario de registro con validación completa

Este es el patrón profesional completo: objeto de valores, objeto de errores, función validar, campos tocados y botón deshabilitado.

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

function validar(v) {
  const e = {};
  if (!v.usuario.trim()) e.usuario = 'El usuario es obligatorio';
  else if (v.usuario.length < 4) e.usuario = 'Mínimo 4 caracteres';
  if (!v.email) e.email = 'El correo es obligatorio';
  else if (!/\S+@\S+\.\S+/.test(v.email)) e.email = 'Correo no válido';
  if (!v.password) e.password = 'La contraseña es obligatoria';
  else if (v.password.length < 8) e.password = 'Mínimo 8 caracteres';
  if (v.confirmar !== v.password) e.confirmar = 'No coinciden';
  return e;
}

function Registro() {
  const [valores, setValores] = useState({ usuario: '', email: '', password: '',
confirmar: '' });
  const [tocados, setTocados] = useState({});
  const [enviado, setEnviado] = useState(false);

  const errores = validar(valores);
  const hayErrores = Object.keys(errores).length > 0;

  function cambio(e) {
    const { name, value } = e.target;
    setValores({ ...valores, [name]: value });
  }

  function blur(e) {
    setTocados({ ...tocados, [e.target.name]: true });
  }

  function mostrar(campo) {
    return (tocados[campo] || enviado) && errores[campo];
  }

  function envio(e) {
```

```

    e.preventDefault();
    setEnviado(true);
    setTocados({ usuario: true, email: true, password: true, confirmar: true });
    if (!hayErrores) {
        alert('Registro exitoso. Bienvenido, ' + valores.usuario);
    }
}

return (
    <form onSubmit={envio}>
        <div>
            <input name="usuario" placeholder="Usuario" value={valores.usuario}
onChange={cambio} onBlur={blur} />
            {mostrar('usuario') && <p>{errores.usuario}</p>}
        </div>
        <div>
            <input name="email" placeholder="Correo" value={valores.email}
onChange={cambio} onBlur={blur} />
            {mostrar('email') && <p>{errores.email}</p>}
        </div>
        <div>
            <input name="password" type="password" placeholder="Contraseña"
value={valores.password} onChange={cambio} onBlur={blur} />
            {mostrar('password') && <p>{errores.password}</p>}
        </div>
        <div>
            <input name="confirmar" type="password" placeholder="Confirmar contraseña"
value={valores.confirmar} onChange={cambio} onBlur={blur} />
            {mostrar('confirmar') && <p>{errores.confirmar}</p>}
        </div>
        <button type="submit" disabled={hayErrores}>Registrarse</button>
    </form>
);
}

```

Vamos por partes. `validar(v)` revisa cada campo y acumula mensajes en el objeto `e`: usuario obligatorio y mínimo 4 caracteres, email obligatorio con formato válido, contraseña obligatoria de mínimo 8, y confirmar debe ser idéntica a password. Nota que los errores se calculan como valor derivado: `const errores = validar(valores);` se ejecuta en cada renderizado, así que siempre están actualizados sin necesidad de otro estado.

La función `mostrar(campo)` es el corazón de la experiencia: devuelve el mensaje solo si el campo ya fue tocado (el usuario salió de él con `onBlur`) o si ya se intentó enviar el formulario. Así no regañamos al usuario antes de tiempo. En `envió` marcamos todo como tocado para

que, si le da al botón con errores, aparezcan todos los mensajes de una vez. El botón se deshabilita con `disabled={hayErrores}` mientras haya cualquier error.

Ejemplo 3: lista filtrable con useMemo

Ahora el rendimiento. Simulamos un catálogo grande generando productos y los filtramos con `useMemo` para que el filtrado pesado no se repita en cada renderizado.

```

import { useState, useMemo } from 'react';

function generarProductos(n) {
  const arr = [];
  for (let i = 1; i <= n; i++) {
    arr.push({ id: i, nombre: 'Producto ' + i, precio: (i * 37) % 5000 });
  }
  return arr;
}

const catalogo = generarProductos(5000);

function Buscador() {
  const [texto, setTexto] = useState('');
  const [contador, setContador] = useState(0);

  const filtrados = useMemo(() => {
    console.log('Filtrando 5000 productos...');
    return catalogo.filter(p =>
      p.nombre.toLowerCase().includes(texto.toLowerCase())
    );
  }, [texto]);

  return (
    <div>
      <input placeholder="Buscar..." value={texto} onChange={e =>
        setTexto(e.target.value)} />
      <button onClick={() => setContador(contador + 1)}>
        Clics: {contador}
      </button>
      <p>Resultados: {filtrados.length}</p>
      {filtrados.slice(0, 20).map(p => (
        <p key={p.id}>{p.nombre} - ${p.precio}</p>
      ))}
    </div>
  );
}

```

generarProductos crea 5000 productos una sola vez, fuera del componente, para que no se regeneren en cada renderizado. Dentro de Buscador hay dos estados: texto para la búsqueda y contador para un botón que no tiene nada que ver con la lista. El botón existe a propósito: cada clic cambia contador, el componente se vuelve a renderizar, y sin useMemo el filtrado de 5000 productos se repetiría en cada clic aunque el texto no cambió.

Con `useMemo`, la función de filtrado solo se ejecuta cuando texto cambia, porque es la única dependencia. Abre la consola del navegador y prueba: escribe en el buscador y verás "Filtrando..." una vez por cada letra; luego dale clic al botón varias veces y verás que el mensaje ya no aparece, porque React reutiliza el resultado guardado. Mostramos solo los primeros 20 resultados con `slice` para no saturar la página. Las dependencias están completas: el cálculo usa catálogo (constante, nunca cambia) y texto (sí cambia), así que `[texto]` es correcto.

Ejemplo 4: confirmación de contraseña con validación cruzada

Este ejemplo se enfoca en un caso que confunde a muchos: validar que dos campos coincidan, con feedback en vivo.

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

function CambioPassword() {
  const [nueva, setNueva] = useState('');
  const [confirmar, setConfirmar] = useState('');
  const [tocadoConfirmar, setTocadoConfirmar] = useState(false);

  const muyCorta = nueva.length > 0 && nueva.length < 8;
  const noCoinciden = tocadoConfirmar && confirmar !== nueva;
  const valida = nueva.length >= 8 && confirmar === nueva && nueva !== '';

  return (
    <form onSubmit={e => e.preventDefault()}>
      <div>
        <input type="password" placeholder="Nueva contraseña" value={nueva}
          onChange={e => setNueva(e.target.value)} />
        {muyCorta && <p>Mínimo 8 caracteres</p>}
      </div>
      <div>
        <input type="password" placeholder="Confirma la contraseña"
          value={confirmar}
          onChange={e => setConfirmar(e.target.value)}
          onBlur={() => setTocadoConfirmar(true)} />
        {noCoinciden && <p>Las contraseñas no coinciden</p>}
      </div>
      {valida && <p>Todo bien, puedes guardar.</p>}
      <button type="submit" disabled={!valida}>Guardar</button>
    </form>
  );
}
```

Aquí usamos estados separados porque son solo dos campos. `muyCorta` solo aparece si ya escribió algo pero no llega a 8: no mostramos error con el campo vacío para no ser agresivos. `noCoinciden` solo aparece después de que el usuario salió del campo de confirmación (`onBlur` activa `tocadoConfirmar`), evitando el mensaje molesto de "no coinciden" mientras apenas va escribiendo la primera contraseña. La variable `valida` resume todo: longitud suficiente y ambas iguales. El botón se habilita solo cuando `valida` es `true`, y aparece un mensaje positivo cuando todo está bien. Dar feedback positivo también es parte de un buen formulario: el usuario siente que avanza.

Ejemplo 5: carrito con total memoizado y cupón validado

Este ejemplo junta los tres temas: un carrito cuyo total se calcula con `useMemo`, un campo de cupón con validación, y un enlace dinámico a la página de pagó.

```
import { useState, useMemo } from 'react';
import { Link } from 'react-router-dom';

const CUPONES = { BIENVENIDA10: 0.10, ESTUDIANTE20: 0.20 };

function Carrito() {
  const [items, setItems] = useState([
    { id: 1, nombre: 'Curso React', precio: 499, cantidad: 1 },
    { id: 2, nombre: 'Curso CSS', precio: 399, cantidad: 2 },
  ]);
  const [cupon, setCupon] = useState('');
  const [cuponAplicado, setCuponAplicado] = useState('');

  const subtotal = useMemo(() => {
    console.log('Calculando subtotal...');
    return items.reduce((acc, i) => acc + i.precio * i.cantidad, 0);
  }, [items]);

  const descuento = CUPONES[cuponAplicado] || 0;
  const total = subtotal * (1 - descuento);

  const errorCupon = cupon && !CUPONES[cupon.toUpperCase()]
    ? 'Cupón no válido'
    : '';

  function aplicarCupon() {
    if (!errorCupon && cupon) {
      setCuponAplicado(cupon.toUpperCase());
    }
  }
}
```

```

return (
  <div>
    <h2>Tu carrito</h2>
    {items.map(i => (
      <p key={i.id}>{i.nombre} x{i.cantidad} - ${i.precio * i.cantidad}</p>
    ))}
    <p>Subtotal: ${subtotal}</p>
    <div>
      <input placeholder="Cupón" value={cupon} onChange={e =>
setCupon(e.target.value)} />
      <button onClick={aplicarCupon}>Aplicar</button>
      {errorCupon && <p>{errorCupon}</p>}
      {cuponAplicado && <p>Cupón {cuponAplicado} aplicado: -{descuento * 100}%</p>}
    </div>
    <h3>Total: ${total}</h3>
    <Link to={'/pago/' + cuponAplicado.toLowerCase()}>Ir a pagar</Link>
  </div>
);
}

```

Línea por línea: CUPONES es un objeto que mapea códigos válidos a su descuento. El subtotal usa useMemo con reduce: solo se recalcula si items cambia, no cuando el usuario escribe en el campo del cupón (ese estado no está en las dependencias, y el cálculo no lo usa). El descuento se lee del cupón aplicado; si no hay, es 0. La validación del cupón es en vivo: en cuanto el texto no coincide con un código válido (comparando en mayúsculas para no ser quisquillosos), aparece el error. aplicarCupon solo guarda el cupón si no hay error. El total es un valor derivado simple que no necesita memoización porque la multiplicación es barata: aquí se ve el criterio en acción. Al final, el Link construye una URL dinámica con el cupón aplicado, mostrando cómo un dato validado viaja hacia la siguiente página. Este ejemplo resume la unidad: datos que fluyen, validación que guía y cálculos que no se repiten.

Práctica

Antes de empezar: cada ejercicio trae su solución completa explicada, pero el aprendizaje está en intentarlo primero sin verla. Ponte un límite de tiempo honesto, unos 20 o 30 minutos por ejercicio, y escribe tu versión aunque quede imperfecta. Después compara con la solución: no busques que tu código sea idéntico, busca que resuelva lo mismo y que entiendas cada línea de la solución. Si tu versión usa otro enfoque válido, también cuenta como acierto. Los seis ejercicios cubren los tres temas de la unidad por separado y combinados, de menor a mayor dificultad.

Ejercicio 1: blog con lista y detalle por id

Crea un mini blog: un arreglo de 4 artículos (id, título, resumen, contenido). La página principal muestra la lista con enlaces, y cada artículo tiene su página de detalle en /artículo/id que muestra el contenido completo. Si el id no existe, muestra "Artículo no encontrado".

Solución:

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

const articulos = [
  { id: 1, titulo: 'Qué es React', resumen: 'Una intro amable.', contenido: 'React es una librería para construir interfaces...' },
  { id: 2, titulo: 'useState explicado', resumen: 'Estado sin miedo.', contenido: 'useState guarda valores que cambian...' },
  { id: 3, titulo: 'Rutas con React Router', resumen: 'Navega tu app.', contenido: 'React Router maneja las páginas...' },
  { id: 4, titulo: 'useEffect sin dolor', resumen: 'Efectos claros.', contenido: 'useEffect corre código después del render...' },
];

function Lista() {
  return (
    <div>
      <h2>Blog</h2>
      {articulos.map(a => (
        <div key={a.id}>
          <Link to={'/articulo/' + a.id}><h3>{a.titulo}</h3></Link>
          <p>{a.resumen}</p>
        </div>
      ))}
    </div>
  );
}

function Detalle() {
  const { id } = useParams();
  const articulo = articulos.find(a => a.id === Number(id));
  if (!articulo) {
    return (
      <div>
        <h2>Artículo no encontrado</h2>
        <Link to="/">Volver al blog</Link>
      </div>
    );
  }
}
```

```

return (
  <div>
    <h2>{articulo.titulo}</h2>
    <p>{articulo.contenido}</p>
    <Link to="/">Volver al blog</Link>
  </div>
);
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Lista />} />
        <Route path="/articulo/:id" element={<Detalle />} />
      </Routes>
    </BrowserRouter>
  );
}

```

Explicación: es el mismo patrón del ejemplo 1 aplicado a artículos. La clave está en Number(id) para comparar con los ids numéricos y en el return temprano cuando find devuelve undefined. Prueba /artículo/3 y luego /artículo/99 para ver ambos caminos.

Ejercicio 2: formulario de login con validación de email

Crea un login con email y password. Valida que el email tenga formato válido y que la contraseña no esté vacía. Muestra un mensaje de error debajo de cada campo solo después de que el usuario lo haya tocado. Deshabilita el botón mientras haya errores.

Solución:

```

import { useState } from 'react';

function Login() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [tocados, setTocados] = useState({});

  const errorEmail = !email
    ? 'El correo es obligatorio'
    : (!/\S+@\S+\.\S+/.test(email) ? 'Correo no válido' : '');
  const errorPassword = !password ? 'La contraseña es obligatoria' : '';
  const hayErrores = errorEmail !== '' || errorPassword !== '';

  function blur(campo) {
    setTocados({ ...tocados, [campo]: true });
  }

  return (
    <form onSubmit={e => { e.preventDefault(); if (!hayErrores)
alert('Bienvenido'); }}>
      <div>
        <input placeholder="Correo" value={email}
          onChange={e => setEmail(e.target.value)} onBlur={() => blur('email')} />
        {tocados.email && errorEmail && <p>{errorEmail}</p>}
      </div>
      <div>
        <input type="password" placeholder="Contraseña" value={password}
          onChange={e => setPassword(e.target.value)} onBlur={() =>
blur('password')} />
        {tocados.password && errorPassword && <p>{errorPassword}</p>}
      </div>
      <button type="submit" disabled={hayErrores}>Entrar</button>
    </form>
  );
}

```

Explicación: los errores se calculan como valores derivados con operadores ternarios anidados: primero se revisa si está vacío, luego el formato. El mensaje solo aparece si el campo fue tocado (onBlur) y el error existe. El botón se deshabilita con hayErrores. Es el patrón profesional en versión compacta para dos campos.

Ejercicio 3: buscador con useMemo sobre 1000 items generados

Genera 1000 usuarios (id, nombre, edad). Agrega un input de búsqueda por nombre y un filtro de edad mínima con otro input numérico. Usa useMemo para que el filtrado solo se recalcule cuando cambien la búsqueda o la edad mínima. Agrega un contador de renderizados o un botón irrelevante para comprobar que el filtrado no se repite.

Solución:

```

import { useState, useMemo } from 'react';

const nombres = ['Ana', 'Luis', 'María', 'Pedro', 'Sofía', 'Diego', 'Lucía', 'Jorge'];
function generarUsuarios(n) {
  const arr = [];
  for (let i = 1; i <= n; i++) {
    arr.push({
      id: i,
      nombre: nombres[i % nombres.length] + ' ' + i,
      edad: 18 + (i * 7) % 50
    });
  }
  return arr;
}
const usuarios = generarUsuarios(1000);

function BuscadorUsuarios() {
  const [texto, setTexto] = useState('');
  const [edadMin, setEdadMin] = useState(0);
  const [clics, setClics] = useState(0);

  const filtrados = useMemo(() => {
    console.log('Filtrando usuarios...');
    return usuarios.filter(u =>
      u.nombre.toLowerCase().includes(texto.toLowerCase()) && u.edad >= edadMin
    );
  }, [texto, edadMin]);

  return (
    <div>
      <input placeholder="Buscar nombre" value={texto} onChange={e =>
        setTexto(e.target.value)} />
      <input type="number" placeholder="Edad mínima" value={edadMin}
        onChange={e => setEdadMin(Number(e.target.value) || 0)} />
      <button onClick={() => setClics(clics + 1)}>Botón irrelevante
        ({clics})</button>
      <p>Encontrados: {filtrados.length}</p>
      {filtrados.slice(0, 15).map(u => (
        <p key={u.id}>{u.nombre}, {u.edad} años</p>
      ))}
    </div>
  );
}

```

Explicación: el cálculo usa texto y edadMin, así que ambas van en las dependencias. El botón irrelevante cambia clics y provoca renderizados: abre la consola y comprueba que "Filtrando usuarios..." no aparece al presionarlo, solo cuando cambias los filtros. `Number(e.target.value) || 0` convierte el input numérico y evita NaN si lo borran. Mostramos 15 resultados para no saturar.

Ejercicio 4: formulario de pedido con validación de cantidad

Un formulario para pedir un producto: nombre del cliente (obligatorio), cantidad (número entre 1 y 10) y dirección de entrega (mínimo 10 caracteres). Valida todo, muestra errores por campo y solo permite enviar si todo es válido.

Solución:

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

function validarPedido(v) {
  const e = {};
  if (!v.cliente.trim()) e.cliente = 'Tu nombre es obligatorio';
  const cant = Number(v.cantidad);
  if (!v.cantidad) e.cantidad = 'Indica la cantidad';
  else if (!Number.isInteger(cant) || cant < 1 || cant > 10) e.cantidad = 'Cantidad entre 1 y 10';
  if (v.direccion.trim().length < 10) e.direccion = 'Dirección muy corta (mínimo 10 caracteres)';
  return e;
}

function Pedido() {
  const [valores, setValores] = useState({ cliente: '', cantidad: '', direccion: '' });
  const [tocados, setTocados] = useState({});
  const errores = validarPedido(valores);
  const hayErrores = Object.keys(errores).length > 0;

  function cambio(e) {
    setValores({ ...valores, [e.target.name]: e.target.value });
  }

  function blur(e) {
    setTocados({ ...tocados, [e.target.name]: true });
  }

  function mostrar(c) {
    return tocados[c] && errores[c];
  }
}
```

```

    return (
      <form onSubmit={e => { e.preventDefault(); if (!hayErrores) alert('Pedido
enviado'); }}>
        <div>
          <input name="cliente" placeholder="Tu nombre" value={valores.cliente}
onChange={cambio} onBlur={blur} />
          {mostrar('cliente') && <p>{errores.cliente}</p>}
        </div>
        <div>
          <input name="cantidad" type="number" placeholder="Cantidad (1-10)"
value={valores.cantidad} onChange={cambio} onBlur={blur} />
          {mostrar('cantidad') && <p>{errores.cantidad}</p>}
        </div>
        <div>
          <input name="direccion" placeholder="Dirección de entrega"
value={valores.direccion} onChange={cambio} onBlur={blur} />
          {mostrar('direccion') && <p>{errores.direccion}</p>}
        </div>
        <button type="submit" disabled={hayErrores}>Enviar pedido</button>
      </form>
    );
  }
}

```

Explicación: la validación de cantidad muestra un caso numérico: convertimos el string del input a número y revisamos que sea entero dentro del rango con `Number.isInteger`. La dirección usa `trim().length` para exigir contenido real. El resto sigue el patrón estándar: objeto de valores, errores derivados, touched por campo y botón deshabilitado.

Ejercicio 5: perfil editable con errores por campo

Un formulario para editar el perfil: nombre de usuario (mínimo 4 caracteres), email válido, biografía (opcional, pero si se escribe, máximo 200 caracteres) y edad (opcional, pero si se escribe debe ser un número entre 13 y 120). Los campos opcionales solo se validan si tienen contenido.

Solución:

```

import { useState } from 'react';

function validarPerfil(v) {
  const e = {};
  if (!v.usuario.trim()) e.usuario = 'El usuario es obligatorio';
  else if (v.usuario.trim().length < 4) e.usuario = 'Mínimo 4 caracteres';
  if (!v.email) e.email = 'El correo es obligatorio';
}

```

```

    else if (!/\S+@\S+\.\S+/.test(v.email)) e.email = 'Correo no válido';
    if (v.bio && v.bio.length > 200) e.bio = 'Máximo 200 caracteres (' + v.bio.length +
'/200)';
    if (v.edad) {
        const num = Number(v.edad);
        if (!Number.isInteger(num) || num < 13 || num > 120) e.edad = 'Edad entre 13 y
120';
    }
    return e;
}

function Perfil() {
    const [valores, setValores] = useState({ usuario: '', email: '', bio: '', edad: ''
});
    const [tocados, setTocados] = useState({});
    const errores = validarPerfil(valores);
    const hayErrores = Object.keys(errores).length > 0;

    function cambio(e) {
        setValores({ ...valores, [e.target.name]: e.target.value });
    }
    function blur(e) {
        setTocados({ ...tocados, [e.target.name]: true });
    }

    return (
        <form onSubmit={e => { e.preventDefault(); if (!hayErrores) alert('Perfil
guardado'); }}>
            <div>
                <input name="usuario" placeholder="Usuario" value={valores.usuario}
onChange={cambio} onBlur={blur} />
                {tocados.usuario && errores.usuario && <p>{errores.usuario}</p>}
            </div>
            <div>
                <input name="email" placeholder="Correo" value={valores.email}
onChange={cambio} onBlur={blur} />
                {tocados.email && errores.email && <p>{errores.email}</p>}
            </div>
            <div>
                <textarea name="bio" placeholder="Biografía (opcional)" value={valores.bio}
onChange={cambio} onBlur={blur} />
                {tocados.bio && errores.bio && <p>{errores.bio}</p>}
                <p>{valores.bio.length}/200</p>
            </div>
        </div>
    );
}

```



```

        <input name="edad" type="number" placeholder="Edad (opcional)"
value={valores.edad} onChange={cambio} onBlur={blur} />
        {tocados.edad && errores.edad && <p>{errores.edad}</p>}
    </div>
    <button type="submit" disabled={hayErrores}>Guardar perfil</button>
</form>
);
}

```

Explicación: la novedad aquí son los campos opcionales con validación condicional: bio y edad solo se revisan si tienen contenido (if (v.bio ...), if (v.edad ...)). El contador de caracteres {valores.bio.length}/200 le da al usuario feedback visual de cuánto le queda, una práctica común en bios y comentarios. La edad acepta vacío pero exige un entero en rango si se escribe. Este patrón de "opcional pero válido si se llena" aparece en casi todos los formularios de perfil reales.

Ejercicio 6: detalle de usuario con carga simulada y redirección

Crea una página de perfil en /usuario/:id. Simula una carga de datos con setTimeout (como si fuera una API). Mientras carga muestra "Cargando...". Si el id no corresponde a ningún usuario, redirige a /no-encontrado con el componente Navigate. Incluye enlaces de "anterior" y "siguiente" usuario usando Link con URLs dinámicas.

Solución:

```

import { useState, useEffect } from 'react';
import { BrowserRouter, Routes, Route, Link, useParams, Navigate } from
'react-router-dom';

const usuarios = [
  { id: 1, nombre: 'Ana', rol: 'Diseñadora' },
  { id: 2, nombre: 'Luis', rol: 'Programador' },
  { id: 3, nombre: 'María', rol: 'Escritora' },
];

function Perfil() {
  const { id } = useParams();
  const [usuario, setUsuario] = useState(null);
  const [cargando, setCargando] = useState(true);
  const idNum = Number(id);

  useEffect(() => {
    setCargando(true);
    const t = setTimeout(() => {

```

```

        setUsuario(usuarios.find(u => u.id === idNum) || null);
        setCargando(false);
    }, 600);
    return () => clearTimeout(t);
}, [idNum]));

if (cargando) return <p>Cargando...</p>;
if (!usuario) return <Navigate to="/no-encontrado" replace />;

return (
    <div>
        <h2>{usuario.nombre}</h2>
        <p>Rol: {usuario.rol}</p>
        <Link to={'/usuario/' + (idNum - 1)}>Anterior</Link>
        {' | '}
        <Link to={'/usuario/' + (idNum + 1)}>Siguiete</Link>
    </div>
);
}

function NoEncontrado() {
    return (
        <div>
            <h2>Usuario no encontrado</h2>
            <Link to="/usuario/1">Ver primer usuario</Link>
        </div>
    );
}

function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route path="/usuario/:id" element={<Perfil />} />
                <Route path="/no-encontrado" element={<NoEncontrado />} />
            </Routes>
        </BrowserRouter>
    );
}

```

Explicación: el `setTimeout` simula la latencia de una red real. El `cleanup return () => clearTimeout(t)` evita que un `timeout` viejo actualice el estado si el usuario navega rápido entre perfiles. `idNum` va en las dependencias del efecto para que al navegar de `/usuario/1` a `/usuario/2` se recarguen los datos sin desmontar el componente. Si el usuario no existe, `Navigate` con `replace` lo manda a la página 404 sin ensuciar el historial. Los enlaces

anterior/siguiente muestran URLs dinámicas construidas con aritmética sobre el id. Prueba /usuario/2, luego dale a "Siguiente" dos veces para llegar al 404.

Glosario rápido de la unidad

Parámetro de ruta: la parte variable de una URL definida con dos puntos, como :id en /producto/:id. useParams: hook de React Router que devuelve un objeto con los parámetros de la ruta actual, siempre como strings. Slug: un identificador legible en la URL, como "teclado-mecanico" en /producto/teclado-mecanico; se usa en lugar de ids numéricos cuando quieres URLs amigables y mejor posicionamiento en buscadores. Touched: marca que indica si el usuario ya interactuó con un campo; sirve para mostrar errores solo después de tocarlo. Validación síncrona: la que se hace con los datos que ya tienes (formato, longitud, coincidencia); validación asíncrona: la que requiere preguntarle al servidor (usuario disponible, cupón válido). Debounce: técnica que espera a que el usuario deje de escribir antes de ejecutar algo costoso, como una petición al servidor. Memoización: guardar el resultado de un cálculo para reutilizarlo. Dependencias: el arreglo que le dice a useMemo (y a useEffect) cuándo debe volver a trabajar. Profiler: herramienta de React DevTools que mide cuánto tarda cada renderizado. Renderizado: cada vez que React vuelve a ejecutar tu componente y actualiza lo que se muestra. Estado derivado: un valor que se calcula a partir de otro estado en cada renderizado, como los errores que se calculan a partir de los valores del formulario, sin necesidad de guardarlo en su propio useState.

Retos extra (sin solución, para practicar por tu cuenta)

Reto 1: extiende el blog del ejercicio 1 con un buscador. Agrega un input en la lista que filtre los artículos por título usando useMemo. Luego haz que cada resultado siga enlazando a su detalle con Link. Pista: el filtrado debe depender solo del texto de búsqueda.

Reto 2: agrega validación asíncrona al formulario de registro del ejemplo 2. Simula con setTimeout una función que "pregunta al servidor" si el nombre de usuario ya existe (usa un arreglo de nombres ocupados). Muestra "Verificando..." mientras esperas y "Ese usuario ya está ocupado" si existe. Usa debounce para no disparar la verificación en cada letra.

Reto 3: crea una página de categoría con dos parámetros: /categoría/:categoría/producto/:id. La lista de productos se filtra primero por categoría y luego se busca el producto por id. Si la categoría no existe o el producto no pertenece a esa categoría, muestra la página de no encontrado. Este reto te obliga a validar dos parámetros a la vez, que es justo lo que pasa en las tiendas reales.

Si completas los tres retos sin mirar las soluciones de los ejercicios, ya dominas la unidad de verdad, no solo de lectura. Y si alguno se te resiste, vuelve a los ejemplos paso a paso: están ahí justo para eso, para consultarlos las veces que hagan falta hasta que el patrón te salga natural.

Resumen de la unidad

En esta unidad cruzaste la línea entre el prototipo y el producto profesional con tres herramientas: rutas dinámicas, formularios con validación y optimización con useMemo.

Con useParams aprendiste a leer parámetros de la URL: defines una ruta como /producto/:id, capturas el valor con `const { id } = useParams()` y lo usas para buscar y mostrar el contenido correspondiente. Te llevas tres reglas que te van a ahorrar horas de depuración: el parámetro siempre llega como string, así que conviértelo a número con `Number(id)` antes de compararlo con ids numéricos; maneja el caso de id inexistente con un return temprano que muestre "no encontrado" en lugar de romper la app; e incluye el parámetro en las dependencias del useEffect que carga los datos, porque el usuario puede navegar de un id a otro sin desmontar el componente. Con Link construyes las URLs dinámicas que conectan la lista con el detalle, y ese flujo lista -> detalle es la columna vertebral de catálogos, blogs y perfiles.

En formularios avanzados aprendiste el patrón profesional completo. Un objeto en el estado guarda todos los valores y una función genérica los actualiza usando el atributo name de cada input. Una función validar() recibe los valores y devuelve un objeto con un mensaje por cada campo inválido. Los errores se muestran junto a cada campo con renderizado condicional, pero solo después de que el usuario tocó el campo (onBlur y el estado touched) o intentó enviar, para no agredirlo con mensajes prematuros. Calculas los errores como valor derivado en cada renderizado en lugar de duplicarlos en otro estado. Deshabilitas el botón de enviar mientras haya errores, y aun así validas de nuevo en el submit. Dominas las validaciones más comunes: requerido con trim, email con una regex simple, longitud mínima y máxima, rangos numéricos y confirmación de contraseña comparando dos campos. Y recuerdas la regla de oro: la validación del navegador mejora la experiencia, pero la validación de seguridad vive en el servidor.

Con useMemo aprendiste qué es la memoización: guardar el resultado de un cálculo pesado y reutilizarlo mientras sus dependencias no cambien. La sintaxis `const total = useMemo(() => suma(lista), [lista])` te da ese poder en una línea. Viste el caso ideal: filtrar miles de elementos sin repetir el trabajo en cada renderizado. Pero igual de importante, aprendiste cuándo no usarlo: la mayoría de los cálculos son baratos y optimizarlos prematuramente solo complica el código. También conociste de pasada a sus parientes, `React.memo` para componentes y `useCallback` para funciones, con la misma idea de fondo: no repitas trabajo ya hecho.

Los errores comunes de la unidad quedaron claros: comparar el id string con números sin convertir, olvidar el caso no encontrado, validar solo al enviar sin feedback en vivo, mostrar errores antes de tocar los campos, y poner dependencias incompletas en useMemo. Si evitas esos cinco, tu código ya está por encima del promedio.

Lo que sigue es práctica deliberada: cada vez que construyas una lista, pregúntate si necesita una vista de detalle con ruta dinámica; cada formulario que hagas, dale validación campo por campo con mensajes claros; y cada vez que sientas que algo "se traba", mide antes de optimizar y usa useMemo solo donde el cálculo lo justifique. Estas tres habilidades, aplicadas con disciplina, son las que separan una app que se ve bien en una demo de una que la gente usa todos los días sin frustrarse.

Y un último consejo que vale para toda tu carrera: los usuarios nunca te van a felicitar por estas tres cosas cuando funcionan bien. Nadie dice "qué bonita validación de formulario" ni "qué rápida la memoización". Pero sí se van, en silencio, cuando fallan: cierran la pestaña si el formulario los frustra, no comparten el producto si la URL no sirve, y dejan de usar la app si se traba. El trabajo invisible bien hecho es el que sostiene a los productos buenos.

Guarda esta unidad como referencia y vuelve a ella cada vez que construyas algo real. La primera vez que la leíste aprendiste los conceptos; la segunda vez que la consultes, ya con un proyecto entre manos, vas a notar detalles que hoy pasaron de largo. Así se aprende de verdad: una vez para entender, y muchas veces para dominar. Ya tienes las herramientas. Ahora a construir.

UNIDAD 10: Proyectos reales y graduación

Llegaste a la última unidad del libro. Respira hondo y date cuenta de algo: hace nueve unidades no sabías qué era un componente, y hoy dominas props, hooks, rutas y consumo de datos. Eso no es cualquier cosa. Pero aquí viene la verdad que separa a quien estudia de quien trabaja: nadie contrata por cuántas definiciones te sabes de memoria, contratan por lo que puedes construir. Esta unidad es la que convierte todo lo que aprendiste en prueba viva de que sabes programar.

Piensa en un arquitecto que solo ha leído libros de estructuras y nunca ha levantado ni un muro. Por más teoría que tenga, nadie le va a confiar un edificio. En programación pasa exactamente igual. Los tutoriales, los cursos y hasta los exámenes son como los planos: útiles, necesarios, pero el edificio real se construye cuando te sientas frente al teclado y conviertes una idea en una aplicación que funciona, que alguien puede abrir en su teléfono y usar.

Por eso esta unidad no es teoría: es taller. Vamos a construir dos proyectos completos, paso a paso, desde cero. El primero es una lista de tareas (to-do app), la clásica de las clásicas. No la menosprecies: en ella están `useState`, formularios controlados, listas con `map` y `keys`, tachar elementos con `map` y borrar con `filter`. Es tu primera app completa en React, y dominarla de verdad vale más que cien ejemplos sueltos. El segundo proyecto es una tienda en línea con carrito de compras: catálogo de productos, página de detalle por producto, carrito global con `Context`, navegación con `React Router` y datos reales desde una API. Eso ya es nivel portafolio, algo que puedes mostrar en una entrevista sin que se te caiga la cara de vergüenza.

Ahora, déjame hablarte de por qué los proyectos mandan en el empleo. Cuando un reclutador revisa tu perfil, lo primero que abre es tu GitHub. No busca tus certificados (aunque tenerlos no estorba), busca código. Quiere ver que tus proyectos tienen README, que los commits tienen sentido, que el código está organizado por componentes y que la app se puede ejecutar. Y si además la tienes desplegada en internet con un link que puede abrir, ya estás por encima del 80% de los candidatos que solo ponen "conocimiento en React" en su CV sin nada que lo respalde.

Cómo presentarlos en tu portafolio importa tanto como construirlos. Cada proyecto debe tener: un nombre claro, una descripción de una o dos líneas que diga qué problema resuelve, las tecnologías que usaste (React, React Router, Context API, etc.), un link a la demo en vivo y un link al código. En el README del repositorio incluye capturas de pantalla, cómo instalarlo y ejecutarlo, y una sección de "qué aprendí" donde cuentes en dos párrafos los retos que enfrentaste. Eso último es oro puro en entrevistas, porque demuestra que pensaste, no que copiaste.

Para que no te quedes en blanco, aquí tienes el esqueleto de un buen README para la tienda. Adáptalo con tus palabras:

```
# Mi Tienda - E-commerce con React

Tienda en línea con catálogo, detalle de producto y carrito de compras.
Los datos vienen de una API pública y el carrito persiste en localStorage.

## Demo
https://mi-tienda.vercel.app

## Tecnologías
React, React Router, Context API, Vite, CSS

## Qué aprendí
- Estado global con Context para el carrito accesible desde todas las rutas.
- Parámetros dinámicos de ruta (/producto/:id) para el detalle.
- El reto fue sincronizar el contador del navbar con el carrito;
  lo resolví con un valor derivado con useMemo en el provider.

## Cómo correrlo
npm install
npm run dev
```

Corto, claro y con evidencia de que pensaste. El reclutador que lea "el reto fue X y lo resolví con Y" ya sabe que puede hablar contigo de código real. Y el link de demo le permite comprobarlo en diez segundos sin clonar nada. Proyecto sin demo es como restaurante sin fotos de la comida: la gente desconfía.

Los dos proyectos de esta unidad están diseñados para ese propósito. La lista de tareas demuestra que dominas el estado local y la interactividad básica. La tienda demuestra arquitectura: contexto global, rutas, datos externos y lógica de negocio (el carrito). Juntos cuentan una historia: "empecé con lo fundamental y llegué a construir una app con estructura profesional". Esa historia es la que te consigue entrevistas.

Un consejo antes de empezar: no copies el código como robot. Escríbelo tú, línea por línea, y en cada línea pregúntate "¿qué hace esto y por qué está aquí?". Cuando algo no funcione (y algo no va a funcionar, te lo garantizo), no borres todo: lee el error, piensa, y arréglalo. Esa es la habilidad real del programador. El que sabe depurar vale más que el que sabe memorizar.

Al final de esta unidad no solo tendrás dos proyectos: tendrás el hábito de construir, el criterio para planear un proyecto de principio a fin, y una ruta clara de qué aprender después. Vamos a cerrar este libro cómo se debe: construyendo.

Un mapa rápido de lo que vas a usar en cada proyecto, para que veas que nada sale de la nada: la lista de tareas usa `useState` (unidad 4), formularios controlados (unidad 5), `map` y `keys` (unidad 6) y `useEffect` con `localStorage` (unidad 7). La tienda usa todo eso más `Context` (unidad 8) y `React Router` (unidad 9). Cada unidad del libro fue un ladrillo; estos proyectos son la casa. Si en algún punto sientes que algo no te sale, no es que "no sirvas para esto": es que un ladrillo quedó flojo y hay que repasarlo. Regresa a esa unidad, repasa el ejemplo, y vuelve. Así trabajan los profesionales también, solo que más rápido porque ya lo hicieron cien veces.

Y una última cosa antes de ensuciarnos las manos: los dos proyectos están pensados para que los hagas en ese orden. La lista de tareas te toma una tarde y te da tu primera victoria completa: una app que hace algo útil, que persiste datos y que puedes mostrar. La tienda te toma un fin de semana y te da tu pieza de portafolio seria. No te saltes la primera por ir directo a la segunda; la confianza se construye con victorias pequeñas antes de los retos grandes.

Explicación detallada

PROYECTO 1: Lista de tareas, tu primera app completa

Vamos a construirla como lo haría un profesional: primero el diseño de los datos, luego la interfaz, luego la lógica, y al final los detalles como la persistencia. Este orden no es capricho: si no sabes qué forma tienen tus datos, no puedes decidir cómo mostrarlos ni cómo modificarlos.

El diseño de los datos

Toda app gira alrededor de sus datos. Nuestra lista de tareas necesita guardar varias tareas, y cada tarea necesita: un identificador único (para saber cuál es cuál), el texto de la tarea, y si está completada o no. En JavaScript eso se representa perfecto como un arreglo de objetos:

```
[
  { id: 1, texto: "Aprender useState", completada: false },
  { id: 2, texto: "Hacer la lista de tareas", completada: true }
]
```

El ``id`` es clave. ¿Por qué no usar el texto como identificador? Porque dos tareas pueden tener el mismo texto ("comprar leche" dos veces), y porque el texto puede cambiar si permitimos editar. El ``id`` es la huella digital de la tarea: único e inmutable. Para generarlo usaremos ``Date.now()``, que devuelve los milisegundos actuales. Es simple y funciona bien para este proyecto: es prácticamente imposible crear dos tareas en el mismo milisegundo a mano. En

apps reales se usan librerías como `uuid`, pero aquí `Date.now()` es suficiente y no requiere instalar nada.

El estado completo de la app vive en un solo `useState` en el componente principal:

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

Un solo arreglo con todo. Esta es una decisión de diseño importante: en lugar de tener un estado para los textos y otro para los completados, guardamos objetos completos. Así cada operación (agregar, tachar, borrar) es una transformación de ese arreglo. Simple de razonar, simple de depurar.

Agregar tareas con un formulario controlado

Necesitamos un input donde el usuario escriba y un botón (o la tecla Enter) para agregar. Aquí entra el formulario controlado que viste en la unidad de formularios: el valor del input vive en el estado, no en el DOM.

```
const [textoNuevo, setTextoNuevo] = useState("");

function agregarTarea(e) {
  e.preventDefault();
  const texto = textoNuevo.trim();
  if (texto === "") return;
  const nuevaTarea = {
    id: Date.now(),
    texto: texto,
    completada: false
  };
  setTareas([...tareas, nuevaTarea]);
  setTextoNuevo("");
}
```

Fíjate en los detalles, porque los detalles son lo que separa el código de juguete del código serio. Primero, `e.preventDefault()`: sin esto, el formulario recargaría la página al enviarse, y perderíamos todo el estado. Es el error número uno de los principiantes con formularios en React. Segundo, `.trim()` y la validación de texto vacío: sin esto el usuario podría agregar tareas en blanco o con puros espacios, y la app se vería rota. Tercero, no mutamos el arreglo original: creamos uno nuevo con el spread `[...tareas, nuevaTarea]`. React detecta cambios comparando referencias, así que mutar el arreglo directamente (con `push`, por ejemplo) no dispararía la actualización de la interfaz. Este es el principio de inmutabilidad que vimos en la unidad de estado: nunca modifiques el estado directamente, siempre crea una copia nueva.

Después de agregar, limpiamos el input con `setTextoNuevo("")`. Pequeño detalle de experiencia de usuario que hace la app sentirse profesional: el campo queda listo para la siguiente tarea.

El JSX del formulario:

```
<form onSubmit={agregarTarea}>
  <input
    type="text"
    value={textoNuevo}
    onChange={(e) => setTextoNuevo(e.target.value)}
    placeholder="Escribe una tarea..."
  />
  <button type="submit">Agregar</button>
</form>
```

Al usar `onSubmit` en el form en lugar de `onClick` en el botón, la tecla Enter también funciona. Otra de esas cosas pequeñas que los usuarios esperan sin saberlo.

Tachar tareas: map con negación

Tachar una tarea significa invertir su estado `completada`: si estaba en false pasa a true y viceversa. Para eso usamos `map`, que recorre el arreglo y devuelve uno nuevo. En cada tarea nos preguntamos: "¿es esta la que quiero cambiar?" Si sí, devolvemos una copia con `completada` invertida; si no, la devolvemos igual.

```
function tacharTarea(id) {
  setTareas(tareas.map((tarea) =>
    tarea.id === id
      ? { ...tarea, completada: !tarea.completada }
      : tarea
  ));
}
```

Lee esta función despacio porque es el patrón más importante de todo el proyecto. `tareas.map(...)` crea un arreglo nuevo (inmutabilidad, ¿recuerdas?). Para cada tarea, el operador ternario decide: si el `id` coincide, spread de la tarea original más `completada: !tarea.completada` (la negación invierte el booleano); si no coincide, la tarea se devuelve intacta. El resultado es un arreglo nuevo donde solo cambió una tarea, y React re-renderiza.

Visualmente, la tarea tachada se muestra con una clase CSS condicional:

```
<span className={tarea.completada ? "completada" : ""}>
  {tarea.texto}
</span>
```

Y en el CSS:

```
.completada {
  text-decoration: line-through;
  color: #888;
}
```

El `className` condicional es un patrón que vas a usar en todos tus proyectos: la apariencia depende del estado. Aquí, el estado `completada` controla si el texto se tacha o no. Estado -> interfaz, siempre. Esa es la filosofía de React en una línea.

Borrar tareas: filter

Borrar es aún más simple conceptualmente: queremos un arreglo nuevo sin la tarea que tiene ese `id`. Eso es exactamente lo que hace `filter`: se queda con los elementos que cumplen la condición.

```
function borrarTarea(id) {
  setTareas(tareas.filter((tarea) => tarea.id !== id));
}
```

Una línea. `filter` devuelve un arreglo nuevo con todas las tareas cuyo `id` sea diferente al que queremos borrar. La tarea con ese `id` simplemente desaparece del arreglo nuevo. Inmutabilidad otra vez: no tocamos el original.

Nota el trío de operaciones que ya dominas: `map` para transformar un elemento, `filter` para eliminar elementos, spread para agregar elementos. Con estas tres operaciones puedes manipular cualquier lista en React. No es exageración: la mayoría de las apps son listas que se agregan, se transforman y se filtran.

Renderizar la lista con keys

```

<ul>
  {tareas.map((tarea) => (
    <li key={tarea.id}>
      <span
        className={tarea.completada ? "completada" : ""}
        onClick={() => tacharTarea(tarea.id)}
      >
        {tarea.texto}
      </span>
      <button onClick={() => borrarTarea(tarea.id)}>Borrar</button>
    </li>
  ))}
</ul>

```

El `key={tarea.id}` es obligatorio y no es decoración. React usa la key para identificar cada elemento de la lista entre renders. Si usaras el índice del arreglo como key y borraras la primera tarea, React se confundiría y podría mostrar datos mezclados. Con el `id` único, cada `<li>` está atado a su tarea para siempre. Regla de oro: la key debe ser estable y única; el `id` de tus datos es perfecto.

Fíjate también en `onClick={() => tacharTarea(tarea.id)}`: usamos una arrow function para pasar el `id` como argumento. Si escribieras `onClick={tacharTarea(tarea.id)}` la función se ejecutaría inmediatamente al renderizar, no al hacer clic. Este es otro error clásico de principiantes. La arrow function crea una función nueva que "recuerda" el `id` y solo se ejecuta cuando el usuario hace clic.

Persistencia con localStorage y useEffect

Hasta aquí la app funciona, pero tiene un problema: si recargas la página, las tareas desaparecen. El estado vive en memoria y la memoria se borra al recargar. La solución es `localStorage`, el almacén del navegador que sobrevive recargas. Necesitamos dos cosas: leer las tareas guardadas al iniciar la app, y guardarlas cada vez que cambien.

Leer al iniciar: el valor inicial de `useState` puede ser una función (inicializador perezoso), que solo se ejecuta una vez:

```

const [tareas, setTareas] = useState(() => {
  const guardadas = localStorage.getItem("mis-tareas");
  return guardadas ? JSON.parse(guardadas) : [];
});

```

¿Por qué una función y no directamente el valor? Porque `localStorage.getItem` y `JSON.parse` se ejecutarían en cada render si los pones directamente, desperdiciando trabajo. Con la función, React la ejecuta solo la primera vez. `localStorage` solo guarda strings, así que convertimos el arreglo a JSON al guardar y lo parseamos al leer. El operador ternario maneja el caso de la primera visita, cuando aún no hay nada guardado: empezamos con un arreglo vacío.

Guardar al cambiar: aquí entra `useEffect`.

```
useEffect(() => {  
  localStorage.setItem("mis-tareas", JSON.stringify(tareas));  
}, [tareas]);
```

Este efecto se ejecuta después de cada render donde `tareas` haya cambiado. Traducido: "cada vez que la lista de tareas cambie, guárdala en `localStorage`". El arreglo de dependencias `[tareas]` es lo que lo hace eficiente: si otro estado cambia pero las tareas no, el efecto no se ejecuta.

Juntos, estos dos pedazos forman la persistencia completa: al abrir la app lees lo guardado, y cada cambio se guarda solo. Cierra la página, apaga la compu, vuelve mañana: tus tareas siguen ahí. Eso convierte un juguete en una app de verdad.

Un detalle importante: `JSON.parse` puede fallar si los datos guardados están corruptos. En un proyecto profesional envolverías la lectura en `try/catch`. Para este proyecto lo dejamos simple, pero que sepas que existe esa consideración.

Estructura de componentes

Hasta ahora todo vive en `App`, y para este proyecto está bien. Pero vamos a organizarlo un poco para practicar la composición: `App` maneja el estado y la lógica, y delega la presentación a componentes hijos.

- `App.jsx`: estado de tareas, funciones agregar/tachar/borrar, persistencia, y el layout general.
- `FormularioTarea.jsx`: recibe `textoNuevo`, `setTextoNuevo` y `agregarTarea` por props; renderiza el form.
- `ListaTareas.jsx`: recibe `tareas`, `tacharTarea` y `borrarTarea`; renderiza el `<ul>`.
- `Tarea.jsx`: recibe una `tarea` y las funciones; renderiza un `<li>`.

¿Vale la pena dividir un proyecto tan chico? Sí, por dos razones. Una: prácticas el flujo de props hacia abajo y eventos hacia arriba, que es el corazón de React. Dos: cuando el proyecto

crezca (filtros, edición, categorías), cada pieza ya tiene su lugar. La regla práctica: si un bloque de JSX tiene una responsabilidad clara y recibe props, es candidato a componente.

El flujo de datos queda así: el estado vive en `App`. `App` pasa datos y funciones hacia abajo por props. Los hijos nunca modifican el estado directamente: llaman a las funciones que recibieron. Cuando el usuario escribe en el input de `FormularioTarea`, se llama a `setTextoNuevo` que vive en `App`; cuando hace clic en tachar, se llama a `tacharTarea` que vive en `App`. Todo el estado en un solo lugar, toda la lógica en un solo lugar, la presentación repartida. Este patrón se llama "levantar el estado" (lifting state up) y lo vas a usar toda tu vida en React.

PROYECTO 2: Tienda con carrito, nivel portafolio

La lista de tareas demuestra que manejas estado local. La tienda demuestra que manejas arquitectura: estado global compartido entre páginas, navegación, datos externos y lógica de negocio real. Es el proyecto que pones hasta arriba de tu portafolio.

Arquitectura general

Antes de escribir código, el mapa. La tienda tiene tres páginas y un estado global:

- Página `/`: catálogo con la lista de productos.
- Página `/producto/:id`: detalle de un producto (foto grande, descripción, precio, botón de agregar al carrito). El `:id` es un parámetro de ruta: cambia según el producto.
- Página `/carrito`: el carrito con cantidades, opción de quitar productos, vaciar, y el total.

Y el estado global: el carrito debe ser accesible desde todas las páginas. Si el carrito viviera en el estado de la página del catálogo, se perdería al navegar al detalle. Por eso usamos Context: `CartContext` envuelve toda la app y expone el carrito y sus operaciones a cualquier componente que lo necesite.

Archivos del proyecto:

- `context/CartContext.jsx`: el contexto del carrito (estado + operaciones).
- `App.jsx`: el Router con las tres rutas, envuelto en el provider del carrito.
- `components/Navbar.jsx`: barra de navegación con link al carrito y contador de productos.
- `components/ProductCard.jsx`: tarjeta de producto para el catálogo.
- `pages/Catalogo.jsx`: página principal, obtiene los productos y los muestra en grid.

- ``pages/DetalleProducto.jsx``: lee el ``:id`` de la URL con ``useParams``, busca el producto y lo muestra.
- ``pages/Carrito.jsx``: muestra el contenido del carrito, cantidades, total y botones.

Esta organización por carpetas (``components``, ``pages``, ``context``) es la convención estándar en proyectos React reales. Cuando un reclutador abra tu repositorio y vea esta estructura, va a reconocer a alguien que ha visto proyectos profesionales.

El contexto del carrito: `CartContext`

El carrito es un arreglo de objetos, donde cada objeto es un producto más su cantidad:

```
{ id: 5, titulo: "Playera", precio: 299, cantidad: 2 }
```

¿Por qué guardar la cantidad en lugar de repetir el producto dos veces en el arreglo? Porque así el total y la interfaz son más simples: una línea por producto con su cantidad, como en las tiendas reales. Las operaciones que necesitamos: agregar (si ya existe, suma cantidad; si no, lo agrega con cantidad 1), quitar (elimina el producto), cambiar cantidad, vaciar todo, y calcular el total.

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

const CartContext = createContext();

export function CartProvider({ children }) {
  const [carrito, setCarrito] = useState([]);

  function agregarAlCarrito(producto) {
    setCarrito((prev) => {
      const existente = prev.find((item) => item.id === producto.id);
      if (existente) {
        return prev.map((item) =>
          item.id === producto.id
            ? { ...item, cantidad: item.cantidad + 1 }
            : item
        );
      }
      return [...prev, { ...producto, cantidad: 1 }];
    });
  }

  function quitarDelCarrito(id) {
    setCarrito((prev) => prev.filter((item) => item.id !== id));
  }
}
```

```

function cambiarCantidad(id, nuevaCantidad) {
  if (nuevaCantidad <= 0) {
    quitarDelCarrito(id);
    return;
  }
  setCarrito((prev) =>
    prev.map((item) =>
      item.id === id ? { ...item, cantidad: nuevaCantidad } : item
    )
  );
}

function vaciarCarrito() {
  setCarrito([]);
}

const total = useMemo(() => {
  return carrito.reduce((suma, item) => suma + item.precio * item.cantidad, 0);
}, [carrito]);

const totalProductos = useMemo(() => {
  return carrito.reduce((suma, item) => suma + item.cantidad, 0);
}, [carrito]);

return (
  <CartContext.Provider
    value={{
      carrito,
      agregarAlCarrito,
      quitarDelCarrito,
      cambiarCantidad,
      vaciarCarrito,
      total,
      totalProductos
    }}
  >
    {children}
  </CartContext.Provider>
);
}

export function useCarrito() {
  return useContext(CartContext);
}

```


Vamos por partes porque aquí hay varios conceptos juntos. Primero, nota que usamos la forma de función en `setCarrito((prev) => ...)`: en lugar de leer `carrito` directamente, recibimos el valor previo como argumento. Esto es más seguro cuando varias actualizaciones pueden ocurrir seguidas (por ejemplo, el usuario hace clic en "agregar" tres veces rápido): cada actualización trabaja sobre el resultado de la anterior, no sobre un valor viejo capturado. Se llama actualización funcional y es la forma correcta cuando el nuevo estado depende del anterior.

Segundo, `agregarAlCarrito` implementa la lógica de "si ya existe suma, si no agrega": busca con `find`, y según el resultado usa `map` (el patrón de tachar de la lista de tareas, ¿ves cómo se repite?) o `spread` para agregar. Los patrones que aprendiste en el proyecto 1 reaparecen aquí. No es coincidencia: son los mismos fundamentos en un contexto más grande.

Tercero, el `total` con `useMemo` y `reduce`. `reduce` recorre el carrito acumulando la suma de `precio * cantidad`. `useMemo` memoriza el resultado y solo lo recalcula cuando `carrito` cambia. ¿Es necesario `useMemo` aquí? Para un carrito chico, no; el cálculo es barato. Pero es buena práctica mostrar que sabes que existe y cuándo usarlo, y en la página del carrito el total se usa en varios lugares. Además, en entrevistas te pueden preguntar para qué sirve `useMemo`, y este es un ejemplo honesto.

Cuarto, el custom hook `useCarrito`. En lugar de que cada componente importe `useContext` y `CartContext`, exponemos un hook con nombre claro. Cualquier componente hace `const { carrito, agregarAlCarrito, total } = useCarrito();` y listo. Es más legible, y si algún día cambias la implementación del contexto, solo tocas un lugar.

El Router: App.jsx con rutas

```

import { BrowserRouter, Routes, Route } from "react-router-dom";
import { CartProvider } from "../context/CartContext";
import Navbar from "../components/Navbar";
import Catalogo from "../pages/Catalogo";
import DetalleProducto from "../pages/DetalleProducto";
import Carrito from "../pages/Carrito";

function App() {
  return (
    <CartProvider>
      <BrowserRouter>
        <Navbar />
        <Routes>
          <Route path="/" element={<Catalogo />} />
          <Route path="/producto/:id" element={<DetalleProducto />} />
          <Route path="/carrito" element={<Carrito />} />
        </Routes>
      </BrowserRouter>
    </CartProvider>
  );
}

export default App;

```

El orden de los envoltorios importa: `CartProvider` está fuera del `BrowserRouter`, así que el carrito está disponible en todas las rutas, incluyendo el `Navbar` que muestra el contador. Si el provider estuviera dentro de una ruta, solo esa ruta tendría acceso. Regla general: los providers globales van lo más afuera posible.

La ruta `/producto/:id` usa un parámetro dinámico. Cuando el usuario visita `/producto/7`, React Router renderiza `DetalleProducto` y el componente puede leer ese `7` con `useParams`. Es el mismo componente para todos los productos; lo que cambia es el dato que muestra según el parámetro. Una ruta, infinitos productos.

Los datos: arreglo local o fetch a API real

Tienes dos opciones para los productos, y te recomiendo hacerlas en orden: primero con un arreglo local para que la app funcione sin depender de internet, y después cambiar a datos reales de una API pública.

Opción 1, arreglo local (`data/productos.js`):

```
export const productos = [
  {
    id: 1,
    titulo: "Playera React",
    precio: 299,
    descripcion: "Playera de algodón con el logo de React.",
    imagen: "https://via.placeholder.com/300",
    categoria: "ropa"
  },
  {
    id: 2,
    titulo: "Taza de programador",
    precio: 149,
    descripcion: "Taza cerámica de 350ml, ideal para el café de las desveladas.",
    imagen: "https://via.placeholder.com/300",
    categoria: "hogar"
  }
];
```

Opción 2, API real con fetch. La API pública `https://fakestoreapi.com/products` devuelve productos de verdad con títulos, precios, descripciones e imágenes. El catálogo los pide con `useEffect`:

```
const [productos, setProductos] = useState([]);
const [cargando, setCargando] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
  fetch("https://fakestoreapi.com/products")
    .then((res) => res.json())
    .then((datos) => {
      setProductos(datos);
      setCargando(false);
    })
    .catch((err) => {
      setError("No se pudieron cargar los productos");
      setCargando(false);
    });
}, []);
```

Los tres estados (`productos`, `cargando`, `error`) son el trío estándar para consumo de datos que viste en la unidad de fetch. La interfaz muestra un mensaje de "Cargando..." mientras pide, el grid si todo sale bien, y un mensaje de error si algo falla. Los tres casos existen en el mundo real: la red es lenta y a veces falla, y tu app debe manejar los tres con dignidad.

Ojo con un detalle: la API devuelve los campos en inglés (`title`, `price`, `description`, `image`). Tienes dos opciones: adaptar tus componentes a esos nombres, o mapear los datos al recibirlos para usar tus nombres en español. Mapear es más limpio:

```
.then((datos) => {
  const adaptados = datos.map((p) => ({
    id: p.id,
    titulo: p.title,
    precio: p.price,
    descripcion: p.description,
    imagen: p.image,
    categoria: p.category
  }));
  setProductos(adaptados);
})
```

Así el resto de tu app siempre trabaja con tu formato, y si mañana cambias de API solo tocas este mapeo. Es una capa de adaptación chiquita que demuestra criterio de arquitectura.

Página de detalle con useParams

```

import { useParams } from "react-router-dom";
import { useCarrito } from "../context/CartContext";

function DetalleProducto({ productos }) {
  const { id } = useParams();
  const { agregarAlCarrito } = useCarrito();
  const producto = productos.find((p) => p.id === Number(id));

  if (!producto) {
    return <p>Producto no encontrado.</p>;
  }

  return (
    <div>
      <img src={producto.imagen} alt={producto.titulo} />
      <h2>{producto.titulo}</h2>
      <p>{producto.descripcion}</p>
      <p><strong>${producto.precio}</strong></p>
      <button onClick={() => agregarAlCarrito(producto)}>
        Agregar al carrito
      </button>
    </div>
  );
}

```

Tres cosas a notar. Una: `useParams` devuelve strings, así que comparamos con `Number(id)` porque nuestros ids son números. Si no conviertes, el `find` nunca encuentra nada y siempre muestra "Producto no encontrado": un bug silencioso y frustrante. Dos: manejamos el caso de producto no encontrado (URL inventada o producto eliminado). Tres: el botón usa `agregarAlCarrito` del contexto: el detalle no sabe nada del carrito, solo llama a la función. Desacoplado y limpio.

¿Cómo llegan los `productos` al detalle? Si usas arreglo local, lo importas directo en el componente. Si usas fetch, necesitas los productos disponibles en varias páginas: ahí conviene otro contexto (`ProductsContext`) o levantar el fetch a `App` y pasar por props. Para no complicar, con fetch puedes hacer que cada página que necesite productos haga su propia petición, o mejor: crea el fetch una vez en un contexto de productos. En los ejemplos paso a paso verás la versión con arreglo local, que es la base; el ejercicio de fetch te toca a ti en la práctica.

La página del carrito

```

import { useCarrito } from "../context/CartContext";

function Carrito() {
  const {
    carrito,
    quitarDelCarrito,
    cambiarCantidad,
    vaciarCarrito,
    total
  } = useCarrito();

  if (carrito.length === 0) {
    return <p>Tu carrito está vacío. Ve al catálogo y agrega algo.</p>;
  }

  return (
    <div>
      <h2>Tu carrito</h2>
      {carrito.map((item) => (
        <div key={item.id}>
          <img src={item.imagen} alt={item.titulo} width="80" />
          <h3>{item.titulo}</h3>
          <p>${item.precio}</p>
          <button onClick={() => cambiarCantidad(item.id, item.cantidad - 1)}>
            -
          </button>
          <span>{item.cantidad}</span>
          <button onClick={() => cambiarCantidad(item.id, item.cantidad + 1)}>
            +
          </button>
          <p>Subtotal: ${item.precio * item.cantidad}</p>
          <button onClick={() => quitarDelCarrito(item.id)}>
            Quitar
          </button>
        </div>
      ))}
      <h3>Total: ${total}</h3>
      <button onClick={vaciarCarrito}>Vaciar carrito</button>
      <button>Finalizar compra</button>
    </div>
  );
}

```

El carrito vacío merece su propio estado visual: un mensaje amable en lugar de una página en blanco. Los botones de `+` y `-` ajustan la cantidad, y fíjate que `cambiarCantidad` ya maneja el caso de llegar a cero (quita el producto). El subtotal por línea y el total general usan el `total` del contexto. El botón "Finalizar compra" lo dejamos como ejercicio en la práctica (pista: un checkout simulado).

El Navbar con contador

```
import { Link } from "react-router-dom";
import { useCarrito } from "../context/CartContext";

function Navbar() {
  const { totalProductos } = useCarrito();

  return (
    <nav>
      <Link to="/">Mi Tienda</Link>
      <Link to="/carrito">Carrito ({totalProductos})</Link>
    </nav>
  );
}
```

Pequeño pero poderoso: el contador del carrito se actualiza solo en todas las páginas porque viene del contexto. Agregas un producto en el detalle y el número del navbar cambia sin que hagas nada más. Esa es la magia del estado global bien puesto: un solo origen de verdad, muchas vistas sincronizadas.

Pasos de construcción ordenados

Si este proyecto te abruma, es porque lo estás viendo completo. Nadie lo construye de un jalón. El orden profesional es:

1. Monta la estructura de archivos y el Router con páginas vacías que digan "Catálogo", "Detalle", "Carrito". Verifica que la navegación funciona.
2. Crea el arreglo local de productos y el catálogo que los muestra en grid con `ProductCard`. Verifica que se ven.
3. Crea la página de detalle con `useParams`. Verifica que cada producto muestra su info.
4. Crea el `CartContext` con agregar y quitar. Conecta el botón del detalle y el contador del navbar. Verifica que el número sube.
5. Crea la página del carrito con cantidades, subtotales y total. Verifica las matemáticas con 2-3 productos.

6. Agrega los estados de carga y vacío, los estilos, y el fetch a la API real.
7. Prueba todo el flujo como usuario: entra, agrega, navega, cambia cantidades, vacía.

Cada paso termina con algo visible y verificable. Si algo se rompe, sabes que fue en el último paso, no en "todo el proyecto". Este es el método que separa a quien termina proyectos de quien los abandona a la mitad.

De la idea al proyecto terminado

Ya tienes dos proyectos guiados. Pero el objetivo real es que puedas construir el tercero tú solo. El método es el mismo siempre, y vale la pena explicitarlo:

Planear: antes de tocar el teclado, escribe en una hoja o en un archivo qué va a hacer la app. Lista las pantallas o páginas, lista los datos que necesita cada una, y lista las acciones del usuario. Para la tienda ese documento habría sido: "páginas: catálogo, detalle, carrito; datos: productos (id, título, precio, descripción, imagen), carrito (producto + cantidad); acciones: ver, agregar, cambiar cantidad, quitar, vaciar". Si no puedes describirlo en palabras, no estás listo para codificarlo.

Dividir en partes: toma esa lista y ordénala de lo más básico a lo más avanzado, como hicimos en los pasos de construcción. Cada parte debe ser construible y probable en una sentada. "Hacer la tienda" es abrumador; "mostrar un grid de productos con datos locales" es una tarde de trabajo. Los proyectos no se terminan de golpe, se terminan por partes.

Probar en cada paso: después de cada parte, úsala como usuario. Haz clic en todo, recarga la página, mete datos raros (textos larguísimos, precios en cero, carrito vacío). Los bugs encontrados temprano cuestan minutos; los encontrados al final cuestan horas porque ya no sabes qué los causó.

Errores comunes en proyectos (y cómo evitarlos)

Querer hacerlo perfecto al inicio: el principiante quiere que la primera versión ya tenga diseño hermoso, animaciones y diez funcionalidades. Resultado: se abruma y no termina nada. La regla es: primero que funcione, después que funcione bien, al final que se vea bonito. La versión 1 de tu tienda puede ser fea; si el carrito suma bien, ya ganaste. El diseño se itera.

No probar en cada paso: escribir 300 líneas sin probar nada y luego preguntarse por qué nada funciona. Ya sabes el antídoto: pasos pequeños, verificación constante. Si algo falla, el culpable está en las últimas 20 líneas, no en las 300.

Perderse sin plan: empezar a codificar sin saber qué datos necesitas ni qué páginas hay. A la mitad descubres que el carrito debería haber sido global y tienes que reescribir todo. Diez minutos de planeación ahorran diez horas de reescritura. El plan no tiene que ser formal: una lista en un papel sirve.

Copiar sin entender: encontrar el código del carrito en internet, pegarlo, y que "funcione" sin saber por qué. El día de la entrevista te preguntan "¿por qué usaste useMemo para el total?" y te quedas en blanco. Usa referencias, claro que sí (todos lo hacemos), pero escribe el código tú y entiendes cada línea. Si no puedes explicar una línea, no la mereces en tu proyecto.

No terminar: el error más caro. Un proyecto al 90% en tu computadora vale cero en tu portafolio; un proyecto al 100% aunque sea simple vale oro. Termina. Desplégalo. Ponle README. El último 10% (pulir, documentar, desplegar) es lo que convierte código en portafolio.

Depurar como profesional: cuando algo no funciona

En los proyectos vas a pasar más tiempo depurando que escribiendo código nuevo. No es señal de que seas malo; es la naturaleza del trabajo. Lo que distingue al profesional no es que no tenga bugs, es que tiene un método para encontrarlos.

Primero: lee el error. React y Vite muestran los errores en el navegador con el archivo y la línea exacta. "Cannot read properties of undefined" significa que intentaste usar algo que no existe (casi siempre un dato que aún no llega del fetch o un prop mal escrito). "X is not a function" significa que llamaste algo que no es función (típico: olvidar los paréntesis al pasar una función vs. ejecutarla). No cierres el error con fastidio: es una pista gratis.

Segundo: usa console.log sin vergüenza. Antes de adivinar por qué el carrito muestra mal el total, imprime el carrito: `console.log(carrito)`. Ver los datos reales casi siempre revela el problema en segundos. Los profesionales lo hacen todos los días; no es trampa, es instrumento.

Tercero: instala React DevTools (extensión del navegador). Te muestra el árbol de componentes, las props que recibe cada uno y el estado actual. Cuando un componente no muestra lo que esperas, abres DevTools y ves exactamente qué datos tiene. Es como rayos X para tu app.

Cuarto: aísla el problema. Si la tienda completa falla, comenta temporalmente secciones hasta que funcione, y luego reactiva una por una. El último pedazo que reactivas antes de que se rompa es el culpable. Este método de "divide y vencerás" funciona siempre, sin excepción.

Quinto: revisa la lista de sospechosos habituales. En React el 90% de los bugs de principiantes son: olvidar `e.preventDefault()` en formularios (la página se recarga), mutar el estado directamente en lugar de copiarlo (la interfaz no se actualiza), keys duplicadas o índices como key (listas que muestran datos mezclados), comparar string con número en `useParams` (el find nunca encuentra), y dependencias mal puestas en `useEffect` (peticiones infinitas o datos que no se actualizan). Con esta lista en la cabeza resuelves la mayoría de los casos en minutos. Y cuando un bug te tome horas, no te sientas mal: hasta los seniors pasan tardes enteras persiguiendo uno. La diferencia es que ellos ya saben que es parte del trabajo y no se rinden.

Publicar y documentar tus proyectos: del código al portafolio

Un proyecto que solo existe en tu computadora no existe para el mundo. Para que cuente en tu portafolio necesita dos cosas: estar publicado en internet con un link que cualquiera pueda abrir, y tener un README que explique qué es. Vamos con ambas.

Publicar es más fácil de lo que crees. Con Vite, el proceso es: primero genera la versión de producción con `npm run build`. Eso crea una carpeta `dist/` con archivos estáticos optimizados (el JavaScript minificado, el CSS, el HTML). Esa carpeta es tu app lista para servirse. Luego súbela a un hosting gratuito: Vercel y Netlify son los más populares para proyectos React. El flujo típico en Vercel es: creas cuenta con tu GitHub, le das a "New Project", seleccionas tu repositorio, y Vercel detecta que es Vite y configura todo solo. Cada vez que subas cambios a GitHub, se redespliega automáticamente. En cinco minutos tienes una URL como `mi-tienda.vercel.app` que puedes poner en tu CV.

Un detalle importante con React Router: como las rutas las maneja JavaScript en el navegador y no el servidor, si alguien entra directo a `mi-tienda.vercel.app/carrito` el servidor podría responder "no encontrado". Vercel y Netlify ya manejan esto bien para apps de una sola página en la mayoría de los casos, pero si te topas con ese error, la solución es configurar una regla de reescritura que sirva `index.html` para cualquier ruta. En Netlify se hace con un archivo `_redirects` en la carpeta public con la línea `/* /index.html 200`. Guárdalo como nota para cuando lo necesites.

Ahora el README. Es el archivo `README.md` en la raíz de tu repositorio, y es lo primero que ve quien abre tu proyecto en GitHub. Un buen README de proyecto de portafolio lleva: título y descripción de una o dos líneas (qué es y qué problema resuelve), capturas de pantalla o GIFs de la app funcionando, lista de tecnologías (React, React Router, Context API, Vite), instrucciones para instalar y correr en local (`npm install`, `npm run dev`), estructura del proyecto si es interesante, y una sección de "qué aprendí" donde cuentas los retos. Esa última sección es la que más habla de ti: "implementé el carrito con Context y useMemo para el total;

el reto fue sincronizar el contador del navbar y lo resolví levantando el estado al provider global". Eso en una entrevista vale oro.

Y GitHub mismo es parte del portafolio. Sube tus proyectos con commits que tengan mensajes claros ("agrega página de detalle con useParams", no "cambios"). Un historial limpio dice "esta persona trabaja ordenada". Agrega una descripción al repositorio y los topics (react, javascript, portfolio). Si quieres ir más allá, crea un repositorio con tu nombre de usuario (github.com/tu-usuario/tu-usuario): GitHub muestra su README en tu perfil, y ahí puedes poner tu presentación, tus proyectos destacados con links a las demos, y tus formas de contacto. Es tu tarjeta de presentación como desarrollador.

Con la lista de tareas y la tienda publicadas, documentadas y en GitHub, ya tienes un portafolio que puedes defender en cualquier entrevista junior. No son proyectos de juguete: son apps completas, funcionales y presentadas como profesional.

Ejemplos paso a paso

Aquí está el código completo de ambos proyectos, listo para copiar, probar y extender. Te recomiendo escribirlo a mano en tu editor en lugar de copiar y pegar: la memoria muscular también programa.

Proyecto 1: Lista de tareas completa (App.jsx)

Este archivo integra todo lo de la explicación: formulario controlado, agregar con validación, tachar con map, borrar con filter, persistencia con localStorage y componentes separados. Guárdalo como `App.jsx` en un proyecto Vite nuevo y funciona tal cual.

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

// Componente: formulario para agregar tareas
function FormularioTarea({ textoNuevo, setTextoNuevo, agregarTarea }) {
  return (
    <form onSubmit={agregarTarea}>
      <input
        type="text"
        value={textoNuevo}
        onChange={(e) => setTextoNuevo(e.target.value)}
        placeholder="Escribe una tarea y pulsa Enter..."
      />
      <button type="submit">Agregar</button>
    </form>
  );
};
```

```

}

// Componente: una sola tarea
function Tarea({ tarea, tacharTarea, borrarTarea }) {
  return (
    <li>
      <span
        className={tarea.completada ? "completada" : ""}
        onClick={() => tacharTarea(tarea.id)}
      >
        {tarea.texto}
      </span>
      <button onClick={() => borrarTarea(tarea.id)}>
        Borrar
      </button>
    </li>
  );
}

// Componente: la lista completa
function ListaTareas({ tareas, tacharTarea, borrarTarea }) {
  if (tareas.length === 0) {
    return <p>No hay tareas. Agrega la primera arriba.</p>;
  }
  return (
    <ul>
      {tareas.map((tarea) => (
        <Tarea
          key={tarea.id}
          tarea={tarea}
          tacharTarea={tacharTarea}
          borrarTarea={borrarTarea}
        />
      ))}
    </ul>
  );
}

// Componente principal: aquí vive todo el estado
function App() {
  // Lee de localStorage solo la primera vez
  const [tareas, setTareas] = useState(() => {
    const guardadas = localStorage.getItem("mis-tareas");
    return guardadas ? JSON.parse(guardadas) : [];
  });
}

```

```

const [textoNuevo, setTextoNuevo] = useState("");

// Guarda en localStorage cada vez que cambian las tareas
useEffect(() => {
  localStorage.setItem("mis-tareas", JSON.stringify(tareas));
}, [tareas]);

function agregarTarea(e) {
  e.preventDefault();
  const texto = textoNuevo.trim();
  if (texto === "") return;
  const nuevaTarea = {
    id: Date.now(),
    texto: texto,
    completada: false
  };
  setTareas([...tareas, nuevaTarea]);
  setTextoNuevo("");
}

function tacharTarea(id) {
  setTareas(tareas.map((tarea) =>
    tarea.id === id
    ? { ...tarea, completada: !tarea.completada }
    : tarea
  ));
}

function borrarTarea(id) {
  setTareas(tareas.filter((tarea) => tarea.id !== id));
}

const pendientes = tareas.filter((t) => !t.completada).length;

return (
  <div className="app">
    <h1>Mis tareas</h1>
    <p>
      Tienes {pendientes} tarea(s) pendiente(s) de {tareas.length}.
    </p>
    <FormularioTarea
      textoNuevo={textoNuevo}
      setTextoNuevo={setTextoNuevo}
      agregarTarea={agregarTarea}
    />
  </div>
)

```

```

    <ListaTareas
      tareas={tareas}
      tacharTarea={tacharTarea}
      borrarTarea={borrarTarea}
    />
  </div>
);
}

export default App;

```

Vamos bloque por bloque para que no quede duda suelta.

El componente `FormularioTarea` recibe tres props: el texto actual, la función para cambiarlo y la función para agregar. Es un componente "tonto": no tiene estado propio, solo muestra lo que le pasan y avisa cuando el usuario hace algo. Este tipo de componente es el más fácil de probar y reutilizar, porque su comportamiento depende 100% de sus props.

El componente `Tarea` renderiza un `<li>` con el texto y el botón de borrar. El clic en el texto tacha; fíjate que el `className` condicional es lo único que cambia visualmente. Todo lo demás (el texto, el botón) es igual esté completada o no. Así de directo es el modelo mental: el estado decide la clase, la clase decide el estilo.

`ListaTareas` maneja el caso borde de la lista vacía con un mensaje amable en lugar de un `<ul>` vacío. Parece detalle menor, pero las apps que cuidan sus estados vacíos se sienten profesionales. Además aquí está el `map` con la `key={tarea.id}`: cada `Tarea` recibe su objeto y las funciones para actuar sobre él.

En `App` vive todo el estado y toda la lógica. El inicializador perezoso de `useState` lee `localStorage` una sola vez. El `useEffect` guarda en cada cambio. Las tres funciones (`agregarTarea`, `tacharTarea`, `borrarTarea`) son las que ya analizamos. Y al final, `pendientes` se calcula con `filter` en cada render: es un valor derivado, no necesita su propio estado porque siempre se puede calcular de `tareas`. Regla útil: si un valor se puede calcular del estado, no lo guardes en el estado; calcúlalo.

El CSS mínimo para que se vea decente:

```

.app {
  max-width: 500px;
  margin: 40px auto;
  font-family: sans-serif;
}

form {
  display: flex;
  gap: 8px;
  margin-bottom: 16px;
}

input {
  flex: 1;
  padding: 8px;
}

ul {
  list-style: none;
  padding: 0;
}

li {
  display: flex;
  justify-content: space-between;
  padding: 8px;
  border-bottom: 1px solid #ddd;
}

li span {
  cursor: pointer;
}

.completada {
  text-decoration: line-through;
  color: #888;
}

```

Pruébalo así: crea el proyecto con Vite, reemplaza `App.jsx`, agrega estos estilos en `index.css`, corre `npm run dev`, y juega: agrega tareas, táchelas haciendo clic en el texto, borra algunas, recarga la página y verifica que todo sigue ahí. Si algo no funciona, abre la consola del navegador (F12): React te dice exactamente qué está mal y en qué línea.

Proyecto 2: Tienda con carrito (esqueleto completo)

Ahora el esqueleto completo de la tienda. Son varios archivos; créalos en las carpetas indicadas dentro de `src`.

Archivo `src/data/productos.js`: los datos locales.

```
export const productos = [
  {
    id: 1,
    titulo: "Playera React",
    precio: 299,
    descripcion: "Playera de algodón con el logo de React. Cómoda y de programador.",
    imagen: "https://via.placeholder.com/300x300?text=Playera",
    categoria: "ropa"
  },
  {
    id: 2,
    titulo: "Taza de programador",
    precio: 149,
    descripcion: "Taza de cerámica de 350 ml para el café de las desveladas.",
    imagen: "https://via.placeholder.com/300x300?text=Taza",
    categoria: "hogar"
  },
  {
    id: 3,
    titulo: "Teclado mecánico",
    precio: 1299,
    descripcion: "Teclado mecánico con switches azules. Suena a productividad.",
    imagen: "https://via.placeholder.com/300x300?text=Teclado",
    categoria: "equipo"
  },
  {
    id: 4,
    titulo: "Libro de JavaScript",
    precio: 499,
    descripcion: "Guía completa de JavaScript moderno, de cero a avanzado.",
    imagen: "https://via.placeholder.com/300x300?text=Libro",
    categoria: "libros"
  }
];
```

Archivo `src/context/CartContext.jsx`: el contexto del carrito. Este es el corazón del proyecto, así que va completo y comentado por bloques:


```

import { createContext, useContext, useState, useMemo } from "react";

const CartContext = createContext();

export function CartProvider({ children }) {
  const [carrito, setCarrito] = useState([]);

  // Agrega un producto: si ya existe suma 1 a su cantidad,
  // si no existe lo agrega con cantidad 1
  function agregarAlCarrito(producto) {
    setCarrito((prev) => {
      const existente = prev.find((item) => item.id === producto.id);
      if (existente) {
        return prev.map((item) =>
          item.id === producto.id
            ? { ...item, cantidad: item.cantidad + 1 }
            : item
        );
      }
      return [...prev, { ...producto, cantidad: 1 }];
    });
  }

  // Quita un producto por completo del carrito
  function quitarDelCarrito(id) {
    setCarrito((prev) => prev.filter((item) => item.id !== id));
  }

  // Cambia la cantidad; si llega a cero, quita el producto
  function cambiarCantidad(id, nuevaCantidad) {
    if (nuevaCantidad <= 0) {
      quitarDelCarrito(id);
      return;
    }
    setCarrito((prev) =>
      prev.map((item) =>
        item.id === id ? { ...item, cantidad: nuevaCantidad } : item
      )
    );
  }

  // Vacía el carrito por completo
  function vaciarCarrito() {
    setCarrito([]);
  }
}

```

```

// Total en dinero: suma de precio * cantidad
const total = useMemo(() => {
  return carrito.reduce(
    (suma, item) => suma + item.precio * item.cantidad,
    0
  );
}, [carrito]);

// Total de productos (para el contador del navbar)
const totalProductos = useMemo(() => {
  return carrito.reduce((suma, item) => suma + item.cantidad, 0);
}, [carrito]);

return (
  <CartContext.Provider
    value={{
      carrito,
      agregarAlCarrito,
      quitarDelCarrito,
      cambiarCantidad,
      vaciarCarrito,
      total,
      totalProductos
    }}
  >
    {children}
  </CartContext.Provider>
);
}

// Hook personalizado: así consumen el carrito los componentes
export function useCarrito() {
  return useContext(CartContext);
}

```

Repasa este archivo hasta que cada función te salga sin mirar. `agregarAlCarrito` usa actualización funcional (`setCarrito((prev) => ...)`) para no trabajar con valores viejos si hay clics rápidos seguidos. El `find` + `map` para sumar cantidad es el mismo patrón de tachar de la lista de tareas, aplicado a otro dominio. Y `useCarrito` es el atajo limpio que evita importar `useContext` y el contexto en cada componente.

Archivo `src/components/Navbar.jsx`:

```

import { Link } from "react-router-dom";
import { useCarrito } from "../context/CartContext";

function Navbar() {
  const { totalProductos } = useCarrito();
  return (
    <nav className="navbar">
      <Link to="/" className="logo">Mi Tienda</Link>
      <Link to="/carrito" className="carrito-link">
        Carrito ({totalProductos})
      </Link>
    </nav>
  );
}

export default Navbar;

```

Archivo `src/components/ProductCard.jsx`:

```

import { Link } from "react-router-dom";
import { useCarrito } from "../context/CartContext";

function ProductCard({ producto }) {
  const { agregarAlCarrito } = useCarrito();
  return (
    <div className="card">
      <Link to={`/${producto.id}`}>
        <img src={producto.imagen} alt={producto.titulo} />
      </Link>
      <h3>{producto.titulo}</h3>
      <p className="precio">${producto.precio}</p>
      <button onClick={() => agregarAlCarrito(producto)}>
        Agregar al carrito
      </button>
      <Link to={`/${producto.id}`}>Ver detalle</Link>
    </div>
  );
}

export default ProductCard;

```

Fíjate en el detalle: la tarjeta tiene dos formas de llegar al detalle (clic en la imagen y link de "Ver detalle") y el botón de agregar directo sin salir del catálogo. Eso es pensar como usuario: algunos quieren ver más antes de comprar, otros ya decidieron. La tienda atiende a los dos.

Archivo `src/pages/Catalogo.jsx`:

```
import { productos } from "../data/productos";
import ProductCard from "../components/ProductCard";

function Catalogo() {
  return (
    <div>
      <h1>Catálogo</h1>
      <div className="grid">
        {productos.map((producto) => (
          <ProductCard key={producto.id} producto={producto} />
        ))}
      </div>
    </div>
  );
}

export default Catalogo;
```

Archivo `src/pages/DetalleProducto.jsx`:

```

import { useParams, Link } from "react-router-dom";
import { productos } from "../data/productos";
import { useCarrito } from "../context/CartContext";

function DetalleProducto() {
  const { id } = useParams();
  const { agregarAlCarrito } = useCarrito();
  const producto = productos.find((p) => p.id === Number(id));

  if (!producto) {
    return (
      <div>
        <p>Producto no encontrado.</p>
        <Link to="/">Volver al catálogo</Link>
      </div>
    );
  }

  return (
    <div className="detalle">
      <img src={producto.imagen} alt={producto.titulo} />
      <div>
        <p className="categoria">{producto.categoria}</p>
        <h1>{producto.titulo}</h1>
        <p>{producto.descripcion}</p>
        <p className="precio-grande">${producto.precio}</p>
        <button onClick={() => agregarAlCarrito(producto)}>
          Agregar al carrito
        </button>
        <Link to="/">Volver al catálogo</Link>
      </div>
    </div>
  );
}

export default DetalleProducto;

```

Archivo `src/pages/Carrito.jsx`: el mismo de la explicación, con la lista de items, botones de cantidad, subtotales, total y vaciar.

Archivo `src/App.jsx`: el Router envuelto en `CartProvider`, igual al de la explicación. No olvides instalar react-router-dom con `npm install react-router-dom` antes de correr.

Y un CSS base para que la tienda no se vea como documento de texto plano. Guárdalo en `src/index.css`:

```
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 0;
  background: #f5f5f5;
}

.navbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  background: #1e1e1e;
  padding: 12px 24px;
}

.navbar a {
  color: white;
  text-decoration: none;
  font-weight: bold;
}

.grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(220px, 1fr));
  gap: 16px;
  padding: 24px;
}

.card {
  background: white;
  border-radius: 8px;
  padding: 16px;
  text-align: center;
  box-shadow: 0 2px 6px rgba(0, 0, 0, 0.1);
}

.card img {
  max-width: 100%;
  border-radius: 4px;
}
```

```

}

.precio {
  font-size: 1.2em;
  font-weight: bold;
  color: #2a7a2a;
}

.card button {
  background: #2a7a2a;
  color: white;
  border: none;
  padding: 10px 16px;
  border-radius: 4px;
  cursor: pointer;
  margin: 8px 0;
}

.detalle {
  display: flex;
  gap: 32px;
  padding: 32px;
  background: white;
  margin: 24px;
  border-radius: 8px;
}

.detalle img {
  max-width: 350px;
  border-radius: 8px;
}

.precio-grande {
  font-size: 2em;
  font-weight: bold;
  color: #2a7a2a;
}

```

El `grid` con `repeat(auto-fill, minmax(220px, 1fr))` es la forma moderna de hacer un catálogo responsive sin media queries: el navegador mete tantas columnas como quepan, y cada tarjeta mide mínimo 220px. Si achicas la ventana, las columnas se reorganizan solas. Pruébalo cambiando el tamaño del navegador: tu catálogo se adapta a móvil y escritorio sin código extra.

Para probarlo: `npm run dev`, abre el catálogo, agrega productos desde las tarjetas, entra al detalle de uno y agrégalo también, abre el carrito desde el navbar y juega con las cantidades. Verifica que el contador del navbar siempre coincide con lo que hay en el carrito y que el total es correcto (haz la suma a mano con dos o tres productos; los decimales no deberían aparecer con estos precios, pero si usas la API real con precios como 109.95, considera redondear con `total.toFixed(2)`).

Con estos dos proyectos tienes tu base de portafolio. Pero un proyecto copiado del libro es solo el inicio: los ejercicios de la siguiente sección son los que lo convierten en algo tuyo.

Práctica

Copiar los proyectos te da la base; extenderlos te hace programador. Cada ejercicio agrega una funcionalidad real, del tipo que te pedirían en un trabajo o que un usuario pediría de verdad. Hazlos en orden, porque algunos se apoyan en conceptos de los anteriores. En cada uno primero intenta resolverlo tú solo (dale al menos 30 minutos de pelea honesta), y luego compara con la solución explicada.

Ejercicio 1: Filtro de pendientes y completadas en la lista

Agrega tres botones encima de la lista: "Todas", "Pendientes" y "Completadas". Al hacer clic, la lista muestra solo las tareas de esa categoría. El contador de pendientes debe seguir funcionando.

Solución explicada: necesitas un estado nuevo para el filtro activo, con tres valores posibles. Como el filtro solo afecta lo que se muestra (no los datos), el valor filtrado se calcula en cada render, igual que hicimos con `pendientes`: es un valor derivado, no va en el estado.

```
const [filtro, setFiltro] = useState("todas"); // "todas" | "pendientes" | "completadas"

const tareasFiltradas = tareas.filter((tarea) => {
  if (filtro === "pendientes") return !tarea.completada;
  if (filtro === "completadas") return tarea.completada;
  return true;
});
```

Y los botones:


```

<div>
  <button onClick={() => setFiltro("todas")}>Todas</button>
  <button onClick={() => setFiltro("pendientes")}>Pendientes</button>
  <button onClick={() => setFiltro("completadas")}>Completadas</button>
</div>

```

Luego `ListaTareas` recibe `tareasFiltradas` en lugar de `tareas`. Nota que borrar y tachar siguen funcionando igual: operan sobre el arreglo completo por `id`, y el filtro solo decide qué se muestra. Esa separación (datos completos vs. vista filtrada) es un patrón que aparece en todas las apps con búsqueda o filtros. Como mejora, marca el botón activo con una clase condicional, igual que hicimos con "completada": `className={filtro === "todas" ? "activo" : ""}`.

Ejercicio 2: Editar tareas existentes

Permite modificar el texto de una tarea ya creada. Al hacer doble clic en el texto (o al pulsar un botón "Editar"), el texto se convierte en un input con el valor actual; al pulsar Enter o salir del input, se guarda el cambio.

Solución explicada: necesitas saber qué tarea se está editando y el texto temporal del input. Dos estados nuevos en `App`: `editandoId` (el id de la tarea en edición, o `null` si ninguna) y `textoEdicion` (el contenido del input).

```

const [editandoId, setEditandoId] = useState(null);
const [textoEdicion, setTextoEdicion] = useState("");

function empezarEdicion(tarea) {
  setEditandoId(tarea.id);
  setTextoEdicion(tarea.texto);
}

function guardarEdicion(id) {
  const texto = textoEdicion.trim();
  if (texto !== "") {
    setTareas(tareas.map((tarea) =>
      tarea.id === id ? { ...tarea, texto: texto } : tarea
    ));
  }
  setEditandoId(null);
  setTextoEdicion("");
}

```

En el componente `Tarea`, si `editandoId === tarea.id` muestra un input; si no, muestra el texto normal:

```
{editandoId === tarea.id ? (  
  <input  
    value={textoEdicion}  
    onChange={(e) => setTextoEdicion(e.target.value)}  
    onBlur={() => guardarEdicion(tarea.id)}  
    onKeyDown={(e) => {  
      if (e.key === "Enter") guardarEdicion(tarea.id);  
      if (e.key === "Escape") setEditandoId(null);  
    }}  
    autoFocus  
  />  
): (  
  <span onClick={() => tacharTarea(tarea.id)}>  
    {tarea.texto}  
  </span>  
)}
```

El patrón de `guardarEdicion` es el mismo `map` con ternario que ya conoces para tachar: solo cambia qué campo se modifica. `onBlur` guarda al salir del input, `Enter` guarda, `Escape` cancela, y `autoFocus` pone el cursor listo. Fíjate cómo un solo patrón (`map` + spread + condición por id) resuelve tachar y editar: dominar pocos patrones a fondo vale más que conocer muchos por encima.

Ejercicio 3: Buscador en la tienda

Agrega una barra de búsqueda en el catálogo que filtre los productos por título mientras escribes. Debe funcionar sin importar mayúsculas o minúsculas.

Solución explicada: un estado para el término de búsqueda y un `filter` derivado, igual que el ejercicio 1 pero con texto libre.

```
const [busqueda, setBusqueda] = useState("");  
  
const productosFiltrados = productos.filter((producto) =>  
  producto.titulo.toLowerCase().includes(busqueda.toLowerCase())  
);
```

El truco está en convertir ambos lados a minúsculas con `toLowerCase()`: así "teclado", "Teclado" y "TECLADO" coinciden. `includes` revisa si el título contiene el texto buscado. Si la

búsqueda está vacía, ``includes("")`` es ``true`` para todo, así que se muestran todos los productos: el caso borde se resuelve solo.

```
<input
  type="text"
  value={busqueda}
  onChange={(e) => setBusqueda(e.target.value)}
  placeholder="Buscar productos..."
/>
{productosFiltrados.length === 0 ? (
  <p>No encontramos productos para "{busqueda}".</p>
) : (
  <div className="grid">
    {productosFiltrados.map((producto) => (
      <ProductCard key={producto.id} producto={producto} />
    ))}
  </div>
)}
```

El mensaje de "no encontramos" cuando no hay resultados es otro estado vacío bien cuidado. Como mejora, combina búsqueda con filtro por categoría: dos filtros derivados encadenados (``filter`` de categoría y luego ``filter`` de búsqueda). Cada filtro es una línea; encadenarlos es trivial una vez que entiendes que ``filter`` devuelve un arreglo nuevo sobre el que puedes filtrar otra vez.

Aquí va esa mejora desarrollada, porque es un patrón que aparece en casi todo e-commerce real. Primero saca las categorías únicas de tus productos:

```
const categorias = ["todas", ...new Set(productos.map((p) => p.categoria))];
const [categoriaActiva, setCategoriaActiva] = useState("todas");
```

``new Set(...)`` elimina duplicados y el spread lo convierte de vuelta en arreglo. Luego encadenas los dos filtros:

```
const productosFiltrados = productos
  .filter((p) => categoriaActiva === "todas" || p.categoria === categoriaActiva)
  .filter((p) => p.titulo.toLowerCase().includes(busqueda.toLowerCase()));
```

Y los botones de categoría:

```

{categorias.map((cat) => (
  <button
    key={cat}
    onClick={() => setCategoriaActiva(cat)}
    className={categoriaActiva === cat ? "activo" : ""}
  >
    {cat}
  </button>
))}

```

Fíjate en lo limpio que queda: cada filtro es una condición independiente, y combinarlos es solo encadenar otro `.filter``. Si mañana agregas filtro por precio ("menos de \$500"), es una línea más en la cadena. Los filtros no se pelean entre sí porque cada uno solo decide "¿pasa este producto mi condición?". Este es el poder de pensar en transformaciones de datos en lugar de enredos de ifs anidados.

Ejercicio 4: Página de checkout simulado

Crea la ruta `/checkout`` con un formulario de datos de envío (nombre, dirección, ciudad, método de pago simulado) y un resumen del pedido. Al "confirmar", muestra una pantalla de éxito con número de orden inventado y vacía el carrito. Si el carrito está vacío, no debe dejar entrar al checkout.

Solución explicada: primero la ruta en `App.jsx``: `<Route path="/checkout" element=<Checkout /> />``. El componente necesita el carrito y el total del contexto, más estados para el formulario.

```

import { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import { useCarrito } from "../context/CartContext";

function Checkout() {
  const { carrito, total, vaciarCarrito } = useCarrito();
  const navigate = useNavigate();
  const [nombre, setNombre] = useState("");
  const [direccion, setDireccion] = useState("");
  const [ciudad, setCiudad] = useState("");
  const [ordenLista, setOrdenLista] = useState(null);

  if (carrito.length === 0 && !ordenLista) {
    return (
      <div>
        <p>No hay productos para comprar.</p>

```

```

        <Link to="/">Volver al catálogo</Link>
    </div>
    );
}

function confirmarOrden(e) {
    e.preventDefault();
    if (nombre.trim() === "" || direccion.trim() === "" || ciudad.trim() === "") {
        return;
    }
    const numeroOrden = "ORD-" + Date.now();
    setOrdenLista({ numero: numeroOrden, total: total });
    vaciarCarrito();
}

if (ordenLista) {
    return (
        <div>
            <h2>Gracias por tu compra, {nombre}</h2>
            <p>Tu número de orden es: <strong>{ordenLista.numero}</strong></p>
            <p>Total pagado: ${ordenLista.total}</p>
            <Link to="/">Seguir comprando</Link>
        </div>
    );
}

return (
    <div>
        <h2>Finalizar compra</h2>
        <p>Total a pagar: ${total}</p>
        <form onSubmit={confirmarOrden}>
            <input
                value={nombre}
                onChange={(e) => setNombre(e.target.value)}
                placeholder="Nombre completo"
            />
            <input
                value={direccion}
                onChange={(e) => setDireccion(e.target.value)}
                placeholder="Dirección"
            />
            <input
                value={ciudad}
                onChange={(e) => setCiudad(e.target.value)}
                placeholder="Ciudad"
            />
        </form>
    </div>
);

```

```

        />
        <button type="submit">Confirmar pedido</button>
    </form>
</div>
);
}

export default Checkout;

```

Los puntos clave: el número de orden se genera con `Date.now()`, igual que los ids de las tareas (mismo truco, nuevo contexto). Guardamos el total en `ordenLista` antes de vaciar el carrito, porque después de `vaciarCarrito()` el total sería cero y la pantalla de éxito mostraría "\$0": el orden de las operaciones importa. La validación evita confirmar con campos vacíos. Y el guardia al inicio (`if (carrito.length === 0 && !ordenLista)`) impide entrar al checkout sin productos, pero permite ver la pantalla de éxito aunque el carrito ya se haya vaciado. Este ejercicio simula un flujo de compra real y es exactamente el tipo de funcionalidad que impresiona en un portafolio.

Ejercicio 5: Modo oscuro en la tienda con context

Agrega un botón en el navbar que alterne entre modo claro y modo oscuro en toda la tienda. El tema debe persistir en localStorage.

Solución explicada: el tema es estado global (afecta a todas las páginas), así que merece su propio contexto, `ThemeContext`, siguiendo el mismo molde del `CartContext`.

```

import { createContext, useContext, useState, useEffect } from "react";

const ThemeContext = createContext();

export function ThemeProvider({ children }) {
  const [oscuro, setOscuro] = useState(() => {
    return localStorage.getItem("tema") === "oscuro";
  });

  useEffect(() => {
    localStorage.setItem("tema", oscuro ? "oscuro" : "claro");
    document.body.className = oscuro ? "tema-oscuro" : "";
  }, [oscuro]);

  function alternarTema() {
    setOscuro(!oscuro);
  }

  return (
    <ThemeContext.Provider value={{ oscuro, alternarTema }}>
      {children}
    </ThemeContext.Provider>
  );
}

export function useTema() {
  return useContext(ThemeContext);
}

```

En `App.jsx` envuelve todo con ``<ThemeProvider>`` (fuera o dentro del ``CartProvider``, no importa mientras esté afuera del contenido). En el navbar:

```

const { oscuro, alternarTema } = useTema();
<button onClick={alternarTema}>
  {oscuro ? "Modo claro" : "Modo oscuro"}
</button>

```

Y en el CSS defines los estilos bajo ``body.tema-oscuro``:

```
body.tema-oscuro {
  background-color: #1a1a1a;
  color: #f0f0f0;
}

body.tema-oscuro .card {
  background-color: #2a2a2a;
  border-color: #444;
}
```

Este ejercicio conecta tres unidades del libro en una sola funcionalidad: Context (unidad de estado global), persistencia con localStorage y useEffect (como en la lista de tareas), y clases condicionales. Cuando puedas combinar conceptos de distintas unidades sin que nadie te diga cómo, ya no estás siguiendo tutoriales: estás programando.

Ejercicio 6: persistencia del carrito en localStorage

Haz que el carrito sobreviva recargas de página, igual que las tareas de la lista. Al abrir la tienda, el carrito debe recuperar lo que tenía; cada cambio debe guardarse automáticamente.

Solución explicada: es el mismo patrón de la lista de tareas aplicado al contexto. En `CartProvider`, el estado inicial lee de localStorage con inicializador perezoso, y un `useEffect` guarda en cada cambio:

```
const [carrito, setCarrito] = useState(() => {
  const guardado = localStorage.getItem("mi-carrito");
  return guardado ? JSON.parse(guardado) : [];
});

useEffect(() => {
  localStorage.setItem("mi-carrito", JSON.stringify(carrito));
}, [carrito]);
```

No olvides importar `useEffect` en el archivo del contexto. Pruébalo: agrega productos, recarga la página y verifica que siguen ahí con sus cantidades y que el total es correcto. Este ejercicio demuestra algo valioso: los patrones se transfieren entre proyectos. Lo que aprendiste persistiendo tareas lo aplicaste persistiendo un carrito sin que nadie te enseñara ese caso específico. Esa transferencia es la señal de que el conocimiento cuajó.

Si completaste los seis ejercicios, tus proyectos ya no son "los del libro": tienen filtros, edición, búsqueda, checkout, modo oscuro y persistencia. Eso es material de portafolio de verdad.

Súbelos a GitHub con un buen README, agrega capturas de pantalla de cada funcionalidad nueva, publica los links de las demos y presume el link en tu LinkedIn y tu CV.

Resumen de la unidad y graduación

Antes de celebrar, hagamos el recorrido completo. Este libro tenía 10 unidades y cada una puso un ladrillo. Aquí están, una por una:

Unidad 1: qué es React y por qué existe. Aprendiste que React es una librería para construir interfaces, que el DOM virtual hace eficiente la actualización de la pantalla, y montaste tu primer proyecto con Vite. El "Hola mundo" que parecía trivial era la puerta de entrada.

Unidad 2: componentes y JSX. Descubriste que toda app de React es un árbol de componentes, que JSX te deja escribir HTML dentro de JavaScript, y que un componente es solo una función que devuelve interfaz. La composición de componentes se volvió tu forma de pensar.

Unidad 3: props. Aprendiste a pasar datos de padres a hijos, a validar lo que recibes y a componer componentes configurables. El flujo unidireccional de datos (los datos bajan, los eventos suben) quedó grabado como ley.

Unidad 4: estado con useState. El momento en que tus apps cobraron vida: contadores, formularios que responden, interfaces que cambian con el clic. Entendiste la inmutabilidad y por qué nunca se muta el estado directamente.

Unidad 5: eventos y formularios. Formularios controlados, validación, preventDefault, y el manejo de lo que el usuario hace. Tus apps dejaron de ser vitrinas y se volvieron herramientas.

Unidad 6: listas y keys. Renderizar arrays con map, la importancia de las keys estables, y los patrones de map/filter que hoy usas dormido. Casi toda app real es una lista con operaciones encima.

Unidad 7: useEffect y ciclo de vida. Efectos secundarios, consumo de datos con fetch, los estados de cargando y error, y la limpieza de efectos. Tus apps empezaron a hablar con el mundo exterior.

Unidad 8: estado global con Context. Cuando el prop drilling se volvió insoportable, el contexto te dio un estado compartido sin pasar props por diez niveles. El custom hook useCarrito nació aquí como idea.

Unidad 9: React Router. Tu app de una página se volvió un sitio de varias páginas: rutas, parámetros dinámicos con `useParams`, links y navegación. La tienda de esta unidad no existiría sin esto.

Unidad 10: proyectos reales. Juntaste todo en dos aplicaciones completas: la lista de tareas (estado local, formularios, persistencia) y la tienda con carrito (contexto, router, datos, lógica de negocio). Del "qué es un componente" al "mira mi tienda funcionando" hay un abismo, y lo cruzaste.

Y ahora lo importante: felicidades. No es frase de relleno. Terminar un libro técnico completo, con código escrito y proyectos funcionando, lo hace una minoría chiquita de la gente que empieza. La mayoría se queda en el capítulo 3. Tú construiste dos apps y las extendiste con cinco ejercicios. Ya no eres alguien que "está aprendiendo React": eres alguien que construye con React. Dilo en voz alta si quieres. Te lo ganaste.

Pero graduarte no significa dejar de aprender; significa que ya sabes aprender solo. La ruta sigue, y aquí está el mapa honesto de lo que sigue y cuándo tomarlo:

Next.js: el framework de React para producción. Te da renderizado del lado del servidor, rutas por archivos y despliegue facilísimo. Tómallo cuando quieras que tus apps carguen más rápido, posicionen en Google y se vean profesionales. Es el siguiente paso natural y el más pedido en ofertas de trabajo junto con React.

Backend con Node: hasta ahora tus apps viven solo en el navegador. Con Node y Express aprenderás a crear tus propias APIs, guardar datos en bases de datos y manejar usuarios con login. Tómallo cuando quieras que tu tienda guarde los pedidos de verdad en lugar de simularlos. Full-stack es la palabra mágica en las vacantes.

TypeScript: JavaScript con tipos. Al principio parece burocracia, pero en proyectos grandes evita una cantidad absurda de bugs y es requisito en la mayoría de empleos serios de React. Tómallo cuando tu JavaScript ya te salga fluido; antes de eso solo te va a estorbar.

React Native: React para apps móviles de verdad, las que se instalan en el teléfono. Reutiliza casi todo lo que sabes: componentes, hooks, estado. Tómallo cuando quieras publicar en App Store y Play Store sin aprender otro lenguaje.

Testing: pruebas automatizadas con herramientas como Vitest y Testing Library. Verifican que tus componentes funcionan aunque cambies el código. Tómallo cuando trabajes en equipo o en proyectos que no te puedes permitir romper; en entrevistas senior siempre preguntan.

Todo esto, gratis, en lucasegaacademy.com. La academia existe para que nunca te atores por dinero: cada tema tiene su curso, sus ejemplos y sus proyectos, igual que este libro.

Últimos consejos para conseguir trabajo, directos y sin humo: tu portafolio manda. Sube la lista de tareas y la tienda a GitHub con READMEs que expliquen qué hacen, qué tecnologías usan y qué aprendiste. Desplégalas gratis (Vercel o Netlify) y pon los links visibles. En LinkedIn escribe en tu titular "Desarrollador React" sin miedo y publica lo que construyes: un post con tu tienda y lo que aprendiste vale más que cien cursos en tu perfil. Y en las entrevistas, cuando te pregunten algo que no sabes, dilo de frente y explica cómo lo averiguarías; la honestidad técnica se nota y se respeta.

Un plan concreto para tus primeros 30 días después de este libro: semana 1, publica los dos proyectos con README y links de demo, y deja tu GitHub presentable. Semana 2, haz los seis ejercicios de extensión y súbelos como mejoras a los mismos repositorios (los commits cuentan historia de progreso). Semana 3, elige un tercer proyecto propio, algo que te interese de verdad (un clon de algo que uses, una app para un negocio de tu familia, lo que sea), y constrúyelo con el método de esta unidad: planear, dividir, probar. Semana 4, arma tu LinkedIn, publica tus tres proyectos y empieza a aplicar a vacantes junior y becas. No esperes a "sentirte listo": nadie se siente listo, y las entrevistas mismas son práctica.

Y cuando apliques, adapta: si la vacante pide TypeScript, menciona que ya tienes la ruta marcada y que estás en ello; si pide Next.js, tu base de React es exactamente lo que Next.js asume. Las empresas contratan juniors por fundamento sólido y capacidad de aprender, no por saberlo todo. Tú ya demostraste las dos cosas terminando este libro.

Gracias por llegar hasta aquí con este libro. Lucas EGA Academy se hizo para gente como tú: la que empieza de cero, se queda hasta la última unidad y construye. Y recuerda algo: el día que te atores en un bug a las 2 de la mañana, que dudes si esto es para ti, vuelve a abrir tu lista de tareas y tu tienda y míralas funcionar. Tú las construiste. Nadie te las regaló. Esa evidencia vale más que cualquier duda pasajera. Nos vemos en el curso de Next.js. A construir se ha dicho.

EXAMEN FINAL

Bienvenido a tu examen final del Curso Completo de React de Lucas EGA Academy. Aquí vas a demostrar todo lo que aprendiste en las 10 unidades del libro: fundamentos y JSX, componentes y props, listas y keys, render condicional, useState, eventos, inputs controlados, formularios, useEffect y dependencias, fetch de APIs, composición y children, estilos, useRef, useContext, custom hooks, React Router y useParams, validación, useMemo y proyectos. El examen tiene 50 preguntas y cada una vale 2 puntos, así que la calificación máxima es 100. Te recomiendo hacerlo sin ver tus apuntes la primera vez, para medir tu nivel real. Cómo calificarte: si sacas de 90 a 100 vas excelente, dominas React y estás listo para construir proyectos reales. Si sacas de 70 a 89 vas bien, solo repasa con calma los temas donde fallaste. Si sacas menos de 70 te conviene repasar el libro completo antes de seguir adelante. En las preguntas abiertas escribe el código como puedas, lo importante es que la lógica sea correcta. Tómate tu tiempo y no te apures. ¡Mucho éxito, tú puedes!

PREGUNTA 1: ¿Qué es JSX?

a) Un lenguaje de programación nuevo creado por Meta para reemplazar a JavaScript. b) Una extensión de sintaxis de JavaScript que te permite escribir código parecido a HTML dentro de tus archivos. c) Una librería para hacer peticiones HTTP a una API. d) El compilador que convierte TypeScript en JavaScript.

PREGUNTA 2: En un componente funcional, ¿cómo recibe las props?

a) Con la palabra reservada this.props dentro de la función. b) Como variables globales declaradas fuera del componente. c) Como el primer parámetro de la función, que normalmente es un objeto. d) React las convierte en variables locales automáticamente sin que hagas nada.

PREGUNTA 3: ¿Cuál es la forma correcta de poner una clase CSS en JSX?

a) `<div class="tarjeta">` b) `<div className="tarjeta">` c) `<div clase="tarjeta">` d) `<div css="tarjeta">`

PREGUNTA 4: Renderizas una lista con map. ¿Por qué cada elemento necesita una prop key única y estable?

a) Porque sin key el código no compila y React marca error. b) Porque React usa la key para identificar cada elemento y actualizar solo lo necesario sin confundir unos con otros. c) Porque la key se muestra en pantalla como identificador del elemento. d) Porque JavaScript exige un identificador único en cada vuelta de map.

PREGUNTA 5: ¿Qué hace este código?

```
{usuario && <Bienvenida nombre={usuario.nombre} />}
```

a) Siempre muestra el componente Bienvenida. b) Muestra Bienvenida solo si usuario tiene un valor verdadero; si es nulo o indefinido no muestra nada. c) Marca error porque falta el caso contrario con else. d) Muestra el texto "usuario" en la pantalla.

PREGUNTA 6: Tienes `const [contador, setContador] = useState(0)`. ¿Cómo sumas 1 correctamente?

a) `contador = contador + 1`; b) `setContador(contador + 1)`; c) `this.setState({ contador: contador + 1 })`; d) `useState(contador + 1)`;

PREGUNTA 7: Haces `lista.push(nuevoElemento)` directamente sobre un estado que es un arreglo, sin usar el setter. ¿Qué pasa?

a) React detecta el cambio y vuelve a renderizar el componente. b) El componente se rompe y lanza un error en la consola. c) El arreglo sí cambia en memoria, pero React no se entera porque la referencia es la misma y no vuelve a renderizar. d) React crea una copia del arreglo automáticamente antes de mutarlo.

PREGUNTA 8: ¿Cómo pasas una función a `onClick` para que se ejecute solo al hacer clic y no durante el render?

a) `<button onClick={saludar()}>Hola</button>` b) `<button onClick={saludar}>Hola</button>` c) `<button onclick="saludar()">Hola</button>` d) `<button click={saludar}>Hola</button>`

PREGUNTA 9: ¿Qué es un input controlado?

a) Un input que solo acepta números. b) Un input cuyo valor vive en el estado de React y se actualiza con `onChange`. c) Un input deshabilitado que el usuario no puede tocar. d) Un input que se valida solo con atributos de HTML.

PREGUNTA 10: En el `onSubmit` de un formulario en React, ¿para qué sirve `e.preventDefault()`?

a) Para validar que los campos no estén vacíos. b) Para evitar que el navegador recargue la página al enviar el formulario. c) Para limpiar los inputs después de enviar. d) Para convertir los datos del formulario a JSON.

PREGUNTA 11: ¿Cuándo se ejecuta un `useEffect` con arreglo de dependencias vacío `[]`?

a) Después de cada render del componente. b) Solo una vez, después del primer render. c) Antes de cada render del componente. d) Nunca se ejecuta.

PREGUNTA 12: ¿Qué provoca un bucle infinito en este código?

```
useEffect(() => {  
  setContador(contador + 1);  
});
```

a) El efecto no tiene arreglo de dependencias, así que corre en cada render, cambia el estado y eso provoca otro render sin fin. b) Que setContador está mal escrito. c) Que falta importar useEffect. d) Nada, este código funciona bien y solo se ejecuta una vez.

PREGUNTA 13: ¿Dónde es correcto hacer fetch a una API para cargar datos cuando el componente se monta?

a) Directo en el cuerpo del componente, fuera de cualquier hook. b) Dentro de un useEffect, porque la petición es un efecto secundario y así no se repite en cada render. c) Dentro del return, en medio del JSX. d) Solo dentro de un manejador onClick.

PREGUNTA 14: ¿Para qué sirve useRef?

a) Para guardar un valor mutable o una referencia a un elemento del DOM sin provocar un re-render cuando cambia. b) Para crear estados más rápidos que useState. c) Para memorizar componentes completos y evitar que se rendericen. d) Para leer los parámetros dinámicos de la URL.

PREGUNTA 15: ¿Qué problema resuelve useContext?

a) Hacer peticiones HTTP más rápidas. b) Evita el prop drilling: comparte datos con componentes muy anidados sin pasar props por cada nivel intermedio. c) Valida formularios automáticamente. d) Memoriza valores de cálculos costosos.

PREGUNTA 16: ¿Cuál es la regla principal al crear un custom hook?

a) Debe ser una clase que herede de React.Hook. b) Su nombre debe empezar con "use" y dentro puede llamar a otros hooks. c) Debe retornar siempre JSX. d) Solo se puede usar fuera de los componentes.

PREGUNTA 17: Con React Router v6, ¿cómo declaras la ruta de la página de inicio?

a) <Route path="/" component={Inicio} /> b) <Route path="/" element={<Inicio />} /> c) <Router to="/" render={Inicio} /> d) <Link path="/" element={Inicio} />

PREGUNTA 18: ¿Qué hace el hook useParams?

a) Lee los query strings de la URL como ?nombre=ana. b) Devuelve un objeto con los parámetros dinámicos de la ruta, como el id en /usuarios/:id. c) Cambia de página sin recargar el documento. d) Guarda datos en el localStorage del navegador.

PREGUNTA 19: ¿Cuál es una buena práctica para validar un formulario en React?

a) Validar solo en el servidor y nunca en el cliente. b) Usar inputs controlados, validar al enviar (y si quieres al escribir) y mostrar mensajes de error claros. c) Usar alert() por cada campo inválido. d) Dejar que la página se recargue si hay errores.

PREGUNTA 20: ¿Para qué sirve useMemo?

a) Para memorizar el resultado de un cálculo costoso y recalcularlo solo cuando cambian sus dependencias. b) Para guardar datos permanentes en el navegador. c) Para evitar que un componente se desmonte. d) Para crear referencias a elementos del DOM.

PREGUNTA 21: ¿Qué es la prop especial children?

a) Lo que escribes entre la etiqueta de apertura y la de cierre de un componente. b) La lista de componentes hijos guardada en el estado. c) Un hook para acceder a los hijos del DOM. d) Una forma de heredar estilos del componente padre.

PREGUNTA 22: ¿Cómo aplicas estilos en línea en React?

a) <div style="color: red"> b) <div style={{ color: "red", fontSize: "20px" }}> c) <div css="color: red"> d) <div style={color: red}>

PREGUNTA 23: Dos componentes hermanos necesitan compartir el mismo estado. ¿Qué haces?

a) Duplicas el estado con useState en cada uno. b) Elevas el estado al componente padre común y lo pasas a ambos por props. c) Los sincronizas usando localStorage. d) No se puede compartir estado entre componentes hermanos.

PREGUNTA 24: ¿Cuál es la diferencia entre un componente controlado y uno no controlado?

a) El controlado guarda su valor en el estado de React; el no controlado deja que el DOM guarde el valor y se lee con una ref. b) El controlado siempre es más rápido. c) El no controlado no sirve para formularios. d) Son lo mismo, solo cambia el nombre.

PREGUNTA 25: En un useEffect que hace fetch, ¿para qué sirve la función de limpieza que retornas?

a) Es obligatoria en todos los efectos sin excepción. b) Para cancelar peticiones o suscripciones pendientes cuando el componente se desmonta o cambian las dependencias, y evitar actualizar el estado de un componente desmontado. c) Para borrar la caché del navegador. d) Para reiniciar el estado a su valor inicial.

PREGUNTA 26 (V/F): En JSX se usa el atributo `class` para asignar clases CSS, igual que en HTML puro.

PREGUNTA 27 (V/F): Si mutas el estado directamente con `push` o con una asignación, React vuelve a renderizar el componente automáticamente.

PREGUNTA 28 (V/F): Un `useEffect` sin arreglo de dependencias se ejecuta después de cada render del componente.

PREGUNTA 29 (V/F): Las `keys` de una lista solo necesitan ser únicas entre elementos hermanos, no en toda la aplicación.

PREGUNTA 30 (V/F): Cambiar el valor guardado en un `useRef` provoca un re-render del componente.

PREGUNTA 31 (V/F): Los custom hooks deben empezar su nombre con "use" para que React los trate como hooks.

PREGUNTA 32 (V/F): En un formulario controlado no hace falta llamar a `e.preventDefault()` porque React lo hace por ti.

PREGUNTA 33 (V/F): El valor inicial que pasas a `useState` solo se usa en el primer render; en los siguientes renders se ignora.

PREGUNTA 34 (V/F): Puedes llamar hooks dentro de un `if` o de un ciclo `for`, siempre que el componente los necesite.

PREGUNTA 35 (V/F): `useMemo` guarda el resultado de un cálculo y evita repetirlo en cada render si sus dependencias no cambiaron.

PREGUNTA 36 (V/F): `useContext` elimina por completo la necesidad de usar props en una aplicación React.

PREGUNTA 37 (V/F): Escribir `onClick={miFuncion()}` hace que la función se ejecute durante el render y no cuando el usuario hace clic.

PREGUNTA 38 (V/F): El componente `Link` de React Router cambia de página sin recargar todo el documento.

PREGUNTA 39 (V/F): useParams sirve para leer los parámetros dinámicos definidos en la ruta, como el :id de /producto/:id.

PREGUNTA 40 (V/F): Un input no controlado guarda su valor en el estado de React y se actualiza con onChange.

PREGUNTA 41 (ABIERTA): Escribe un componente llamado Contador que muestre un número en pantalla y tenga dos botones: uno para sumar 1 y otro para restar 1. Usa useState. El contador no debe bajar de 0.

PREGUNTA 42 (ABIERTA): El siguiente código tiene 3 errores. Encuéntralos, explica cada uno y escribe el código corregido:

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

  function agregar() {
    tareas.push("Nueva tarea");
  }

  return (
    <div>
      <button onClick={agregar()}>Agregar</button>
      <ul>
        {tareas.map((tarea) => (
          <li>{tarea}</li>
        ))}
      </ul>
    </div>
  );
}
```

PREGUNTA 43 (ABIERTA): Explica con tus palabras por qué es importante la prop key en las listas de React y qué problemas aparecen si usas el índice del arreglo como key cuando la lista puede cambiar de orden o se le pueden agregar o quitar elementos.

PREGUNTA 44 (ABIERTA): Completa el siguiente componente para que obtenga la lista de usuarios de <https://api.ejemplo.com/usuarios> cuando se monta, muestre "Cargando..." mientras pide los datos, muestre un mensaje si algo falla y al final muestre los nombres en una lista:

```
function Usuarios() {  
  const [usuarios, setUsuarios] = useState([]);  
  // completa aquí: estados, efecto y render  
}
```

PREGUNTA 45 (ABIERTA): Escribe un formulario de registro con dos inputs controlados: nombre y correo. Al enviar, valida que el nombre no esté vacío y que el correo contenga un "@". Si algo falla muestra el mensaje de error correspondiente; si todo está bien muestra "Registro exitoso". No olvides evitar la recarga de la página.

PREGUNTA 46 (ABIERTA): Escribe un custom hook llamado useContador que reciba un valor inicial, devuelva el valor actual más dos funciones (incrementar y decrementar), y luego úsalo dentro de un componente que muestre el valor con dos botones.

PREGUNTA 47 (ABIERTA): Crea un TemaContext, un componente proveedor que comparta el valor "oscuro" y un componente Hijo que lea ese valor con useContext y lo muestre en pantalla dentro de un párrafo.

PREGUNTA 48 (ABIERTA): Explica qué es un bucle infinito en useEffect, por qué ocurre, y escribe un ejemplo de código que lo provoque junto con su versión corregida.

PREGUNTA 49 (ABIERTA): Con React Router, escribe las rutas para: la página de inicio (/) que muestra el componente <Inicio />, y la página de detalle (/producto/:id) que muestra <DetalleProducto />. Dentro de DetalleProducto usa useParams para mostrar el id del producto en pantalla.

PREGUNTA 50 (ABIERTA): Escribe un componente que use useRef para enfocar automáticamente un input cuando el componente se monta, de modo que al cargar la página el cursor ya esté dentro del input listo para escribir.

Respuestas y soluciones

RESPUESTA 1: b) Una extensión de sintaxis de JavaScript que te permite escribir código parecido a HTML dentro de tus archivos.

JSX no es un lenguaje nuevo ni una librería aparte: es una extensión de sintaxis de JavaScript que te deja describir la interfaz con un marcado parecido a HTML dentro de tu código. Herramientas como Babel lo transforman en llamadas a React.createElement, que es lo que React realmente entiende. La a) es incorrecta: JSX no reemplaza a JavaScript, solo lo extiende para describir interfaces. La c) se equivoca de tema: las peticiones HTTP se hacen con fetch o axios, nada que ver con JSX. La d) confunde con TypeScript, que sí usa un

compilador para otro lenguaje. La ventaja de JSX es que ves la estructura de tu interfaz, mezclas lógica con marcado usando llaves y escribes componentes más legibles que concatenando strings a mano.

RESPUESTA 2: c) Como el primer parámetro de la función, que normalmente es un objeto.

En React las props viajan de padre a hijo como un solo objeto que el componente funcional recibe en su primer parámetro, por eso lo típico es escribir `function Saludo(props)` o desestructurar directamente `function Saludo({ nombre })`. La opción a) es incorrecta porque `this.props` pertenece a los componentes de clase, no a los funcionales, y en una función `this` ni siquiera apunta al componente. La b) falla porque las props no son globales: cada componente recibe solo las que su padre le pasa. La d) también es falsa porque React no crea variables mágicas por ti; si no recibes el parámetro props no tienes acceso a nada. Entender esto es clave porque todo el flujo de datos en React depende de pasar ese objeto de props correctamente.

RESPUESTA 3: b) `<div className="tarjeta">`

En JSX no puedes usar `class` porque es una palabra reservada de JavaScript (se usa para declarar clases), así que React te pide que uses `className` en su lugar. La opción a) es la trampa clásica de quien viene de HTML puro: si la escribes, React te muestra una advertencia en la consola y el estilo no se aplica como esperas. La c) no existe en React y la d) tampoco; `css` como atributo no es válido en JSX. Este detalle parece menor pero aparece en casi todos los proyectos, y el libro lo repite mucho porque es de los errores más comunes al empezar. Recuerda la regla: en JSX, `class` se convierte en `className` y `for` se convierte en `htmlFor`, por la misma razón de palabras reservadas.

RESPUESTA 4: b) Porque React usa la key para identificar cada elemento y actualizar solo lo necesario sin confundir unos con otros.

Cuando renderizas una lista, React necesita saber qué elemento es cuál entre un render y otro para reutilizar el DOM de forma eficiente. La `key` es ese identificador: con una `key` estable React sabe que un elemento solo cambió de posición o de contenido, y no lo destruye para crear otro. La opción a) es falsa porque el código sí compila sin `key`; solo verás una advertencia en consola. La c) es un invento: la `key` nunca se muestra en pantalla. La d) también es falsa porque `map` no exige ningún identificador; es React quien lo pide. Sin `keys` correctas aparecen errores raros: inputs que pierden su valor, animaciones que se rompen o elementos que muestran datos de otro.

RESPUESTA 5: b) Muestra Bienvenida solo si usuario tiene un valor verdadero; si es nulo o indefinido no muestra nada.

Este es el patrón de render condicional con el operador `&&`, muy usado en React. Si usuario es null, undefined, false o 0, la expresión se evalúa como falsy y React no renderiza nada en ese lugar. Solo cuando usuario existe y es verdadero se evalúa el lado derecho y aparece el componente Bienvenida con su prop nombre. La opción a) es incorrecta porque el componente no siempre aparece: depende del valor de usuario. La c) es falsa porque en React no necesitas un else; no renderizar nada es perfectamente válido. La d) confunde: nunca se muestra el texto "usuario", solo se usa su valor para decidir. Este patrón es ideal para mostrar contenido solo cuando los datos ya llegaron de una API.

RESPUESTA 6: b) `setContador(contador + 1)`;

En React nunca modificas el estado directamente: usas la función setter que te devuelve `useState`, y ella se encarga de guardar el nuevo valor y pedir un re-render. La opción a) es el error clásico: asignar directamente a la variable no avisa a React y la interfaz no se actualiza. La c) pertenece a los componentes de clase con `this.setState`, que no aplica en componentes funcionales. La d) no tiene sentido: `useState` solo se llama una vez para declarar el estado, no para actualizarlo. Además conviene recordar que si el nuevo valor depende del anterior y hay varias actualizaciones seguidas, lo más seguro es usar la forma funcional `setContador(c => c + 1)`, así siempre trabajas con el valor más reciente.

RESPUESTA 7: c) El arreglo sí cambia en memoria, pero React no se entera porque la referencia es la misma y no vuelve a renderizar.

React decide si debe actualizar comparando referencias, no revisando el contenido elemento por elemento. Cuando haces push sobre el mismo arreglo, la referencia no cambia, así que React cree que nada pasó y no vuelve a renderizar: tus datos cambiaron pero la pantalla se quedó igual. La opción a) describe lo que uno esperaría, pero no es así cómo funciona. La b) es falsa porque mutar no lanza ningún error, solo silencio. La d) también es falsa: React no hace copias por ti. La forma correcta es crear un arreglo nuevo, por ejemplo `setLista([...lista, nuevoElemento])`. Este es uno de los errores más comunes y más difíciles de detectar, porque el `console.log` sí muestra el dato nuevo aunque la pantalla no cambie.

RESPUESTA 8: b) `<button onClick={saludar}>Hola</button>`

En React debes pasar la referencia de la función, sin paréntesis, para que se ejecute solo cuando ocurra el clic. La opción a) es la trampa más común del curso: con paréntesis la función se ejecuta de inmediato durante el render, y además su valor de retorno (normalmente undefined) queda como manejador, así que el botón ya no hace nada al hacer clic. La c) usa la

sintaxis de HTML puro con string, que en React no funciona porque los eventos se pasan como props de JavaScript. La d) inventa una prop click que React no reconoce; el nombre correcto del evento es onClick con camelCase. Si necesitas pasar argumentos, usa una función flecha: onClick={() => saludar("Ana")}.

RESPUESTA 9: b) Un input cuyo valor vive en el estado de React y se actualiza con onChange.

Un input controlado tiene su atributo value conectado a una variable de estado y un onChange que actualiza ese estado con cada tecla que el usuario escribe. Así React siempre sabe lo que hay en el campo y tú puedes validarlo, limpiarlo o transformarlo cuando quieras. La opción a) es falsa porque un controlado acepta cualquier texto; el tipo de dato lo decides tú. La c) describe un input deshabilitado, que es otra cosa. La d) habla de validación HTML nativa, que funciona con o sin React. La contraparte es el input no controlado, donde el DOM guarda el valor y lo lees con una ref. El libro recomienda los controlados para formularios porque te dan control total sobre los datos.

RESPUESTA 10: b) Para evitar que el navegador recargue la página al enviar el formulario.

Por defecto, cuando envías un formulario HTML el navegador recarga la página y manda los datos al servidor, lo cual rompe la experiencia de una aplicación React de una sola página: perderías todo el estado. Con e.preventDefault() cancelas ese comportamiento nativo y te quedas con el control para hacer lo que quieras con los datos, como validarlos o mandarlos con fetch. La opción a) es incorrecta porque preventDefault no valida nada; la validación la programas tú aparte. La c) es falsa porque limpiar los inputs se hace actualizando el estado. La d) también es falsa: convertir a JSON es un paso que tú haces manualmente. Sin esta línea, cada submit reinicia tu app.

RESPUESTA 11: b) Solo una vez, después del primer render.

Un arreglo de dependencias vacío le dice a React que el efecto no depende de ningún valor, así que solo necesita ejecutarse una vez cuando el componente se monta, parecido al viejo componentDidMount de las clases. Es el lugar ideal para cargar datos iniciales de una API o suscribirte a algo una sola vez. La opción a) describe lo que pasa cuando omites el arreglo por completo: el efecto corre después de cada render. La c) es falsa porque los efectos siempre corren después del render, nunca antes; React primero pinta y luego ejecuta efectos. La d) también es falsa: con [] el efecto sí se ejecuta, una sola vez. Ojo con un detalle: en modo estricto de desarrollo React puede ejecutar el efecto dos veces para detectar problemas, pero en producción corre una sola vez.

RESPUESTA 12: a) El efecto no tiene arreglo de dependencias, así que corre en cada render, cambia el estado y eso provoca otro render sin fin.

Sin arreglo de dependencias el efecto se ejecuta después de cada render. Dentro del efecto llamas a `setContador`, lo que actualiza el estado, provoca otro render y el ciclo se repite sin fin. El navegador se congela o React te lanza una advertencia de renders infinitos. Las opciones b) y c) son distractores: el setter está bien escrito y `useEffect` se asume importado. La d) es la trampa: este código jamás funciona bien. La solución es agregar el arreglo de dependencias correcto: si solo quieres que corra al montar usa `[]`, y si el efecto depende de `contador` inclúyelo en el arreglo. Este es el error más famoso de `useEffect` y el libro le dedica una sección completa.

RESPUESTA 13: b) Dentro de un `useEffect`, porque la petición es un efecto secundario y así no se repite en cada render.

Pedir datos a una API es un efecto secundario: algo que va más allá de calcular lo que se muestra en pantalla. Si pones el `fetch` directo en el cuerpo del componente, se ejecutaría en cada render, y como al llegar los datos actualizas el estado, provocarías otro render y otra petición en un ciclo sin fin. El `useEffect` con arreglo vacío `[]` garantiza que la petición se haga una sola vez al montar el componente. La opción a) causa ese ciclo infinito de peticiones. La c) es imposible: en el `return` solo describes la interfaz, no ejecutas lógica asíncrona. La d) es muy limitada porque los datos no se cargarían solos al abrir la página. Además, dentro del efecto puedes manejar carga, error y limpieza de forma ordenada.

RESPUESTA 14: a) Para guardar un valor mutable o una referencia a un elemento del DOM sin provocar un re-render cuando cambia.

`useRef` te devuelve un objeto con una propiedad `current` que puedes leer y modificar libremente sin que React vuelva a renderizar, a diferencia de `useState`. Sus dos usos principales son: guardar una referencia directa a un elemento del DOM (por ejemplo para enfocar un input con `inputRef.current.focus()`) y guardar valores mutables que deben sobrevivir entre renders, como el id de un temporizador. La opción b) es falsa: `useRef` no crea estado visible y es más limitado para la interfaz. La c) describe a `React.memo` o `useMemo`, no a `useRef`. La d) describe a `useParams` de `React Router`. Recuerda: si el valor debe mostrarse en pantalla y cambiar la interfaz, usa `useState`; si solo lo necesitas internamente, `useRef` es tu amigo.

RESPUESTA 15: b) Evita el prop drilling: comparte datos con componentes muy anidados sin pasar props por cada nivel intermedio.

Cuando un dato lo necesita un componente que está cinco niveles abajo, pasarlo por props obliga a todos los componentes intermedios a recibirlo y reenviarlo aunque no lo usen; eso es el prop drilling y vuelve el código frágil. Con createContext y useContext creas un canal directo: el proveedor pone el valor arriba y cualquier componente abajo lo lee con useContext sin intermediarios. La opción a) es falsa porque el contexto no acelera peticiones. La c) también: la validación se programa aparte. La d) describe a useMemo. Ojo: el contexto no elimina las props por completo, solo evita pasar datos por niveles que no los necesitan; para datos locales de un componente las props siguen siendo lo correcto.

RESPUESTA 16: b) Su nombre debe empezar con "use" y dentro puede llamar a otros hooks.

La convención de que el nombre empiece con "use" no es un capricho: así React y las herramientas (como el linter) saben que esa función es un hook y le aplican las reglas de los hooks, además de permitirte llamar useState o useEffect dentro de ella. Un custom hook es simplemente una función que encapsula lógica con estado reutilizable, como useContador o useFetch, y puedes usarla en varios componentes. La opción a) es falsa porque los hooks son funciones, no clases. La c) es falsa: un hook normalmente retorna valores o funciones, no JSX (eso lo hacen los componentes). La d) también es falsa: los hooks solo pueden llamarse dentro de componentes o dentro de otros hooks, nunca sueltos. Esta convención es lo que hace posible compartir lógica sin repetir código.

RESPUESTA 17: b) <Route path="/" element={<Inicio />} />

En React Router v6 la forma correcta es usar el componente Route dentro de Routes con la prop element, a la que le pasas el componente ya instanciado como JSX: element={<Inicio />}. La opción a) usa la sintaxis de la versión 5 (component={Inicio}), que en v6 ya no funciona y es un error muy común al seguir tutoriales viejos. La c) inventa una API: Router no recibe to ni render de esa forma. La d) confunde Link, que sirve para navegar con un enlace clicable, no para declarar rutas. Recuerda la estructura completa: <BrowserRouter> envuelve a <Routes>, que contiene tus <Route>. Si vienes de tutoriales antiguos, este cambió de component a element es lo primero que debes actualizar.

RESPUESTA 18: b) Devuelve un objeto con los parámetros dinámicos de la ruta, como el id en /usuarios/:id.

Cuando defines una ruta con segmentos dinámicos usando dos puntos, como /producto/:id, useParams te devuelve un objeto con esos valores tomados de la URL actual. Si visitas /producto/42, useParams te da { id: "42" } y con eso puedes buscar el producto en tu API. La opción a) describe los query strings (?nombre=ana), que se leen con useSearchParams, no con useParams. La c) describe lo que hace Link o useNavigate. La d) habla de localStorage,

que no tiene nada que ver con el router. Un detalle importante: los valores siempre llegan como strings, así que si necesitas un número debes convertirlo tú con `Number()` o `parseInt`.

RESPUESTA 19: b) Usar inputs controlados, validar al enviar (y si quieres al escribir) y mostrar mensajes de error claros.

La validación en el cliente mejora la experiencia porque el usuario se entera al momento si algo está mal, sin esperar al servidor. Con inputs controlados siempre tienes los valores en el estado, así que al hacer submit puedes revisar reglas (campos vacíos, formato de correo, longitud mínima) y guardar los mensajes de error en otro estado para mostrarlos junto a cada campo. La opción a) es mala práctica: validar solo en el servidor hace la app lenta y torpe. La c) con `alert()` es invasiva y anticuada. La d) es inaceptable en una SPA porque recargar pierde todo el estado. Recuerda que la validación del cliente no sustituye la del servidor: el servidor siempre debe validar de nuevo por seguridad.

RESPUESTA 20: a) Para memorizar el resultado de un cálculo costoso y recalcularlo solo cuando cambian sus dependencias.

Si tienes una operación pesada (filtrar miles de elementos, un cálculo matemático complejo) y el componente se vuelve a renderizar por cualquier motivo, sin `useMemo` ese cálculo se repetiría aunque sus datos no hayan cambiado. Con `useMemo` le dices a React: guarda este resultado y solo recalculalo cuando cambien los valores del arreglo de dependencias. La opción b) describe `localStorage` o `sessionStorage`. La c) no existe: ningún hook evita el desmontaje. La d) describe a `useRef`. Un error común es usar `useMemo` para todo; no vale la pena en cálculos baratos porque la memorización también tiene un costo. Úsalo cuando midas un problema real de rendimiento, no por adelantado.

RESPUESTA 21: a) Lo que escribes entre la etiqueta de apertura y la de cierre de un componente.

La prop `children` es especial porque React la llena automáticamente con todo lo que pongas entre las etiquetas del componente, por ejemplo el texto y el botón dentro de `<Tarjeta>...</Tarjeta>`. Esto permite crear componentes contenedores reutilizables (tarjetas, modales, layouts) que envuelven cualquier contenido sin saber de antemano qué será. La opción b) es falsa: los hijos no se guardan en el estado a menos que tú lo programes. La c) inventa un hook que no existe. La d) confunde con los estilos, que se pasan por `className` o `style`. Este patrón se llama composición y es la forma preferida en React para reutilizar: en lugar de heredar componentes como en la programación orientada a objetos, los compones envolviendo contenido con `children`.

RESPUESTA 22: b) `<div style={{ color: "red", fontSize: "20px" }}>`

En React la prop style espera un objeto de JavaScript, no un string como en HTML. Por eso lleva dobles llaves: las de afuera son las llaves de JSX para insertar JavaScript, y las de adentro son las del objeto. Además las propiedades CSS se escriben en camelCase (fontSize en vez de font-size) porque los guiones no son válidos en nombres de propiedades de objetos. La opción a) usa la sintaxis de HTML puro y React la ignora o marca error. La c) inventa una prop css que no existe. La d) falla porque color: red sin comillas ni objeto no es JavaScript válido. Los estilos en línea sirven para estilos dinámicos, pero para diseños grandes el libro recomienda clases CSS o módulos, que son más fáciles de mantener.

RESPUESTA 23: b) Elevas el estado al componente padre común y lo pasas a ambos por props.

En React los datos fluyen hacia abajo, así que dos hermanos no pueden pasarse estado directamente entre ellos. La solución es subir el estado al primer ancestro que tengan en común: el padre guarda el valor con useState y se lo pasa a cada hijo por props, junto con la función para actualizarlo. Así ambos siempre ven el mismo dato y se mantienen sincronizados. La opción a) duplica el estado y crea dos fuentes de verdad que se desincronizan al primer cambió. La c) con localStorage es un parche lento y fuera del flujo de React. La d) es falsa: claro que se puede, solo hay que elevar el estado. Este patrón se llama "lifting state up" y es fundamental para que los componentes se comuniquen.

RESPUESTA 24: a) El controlado guarda su valor en el estado de React; el no controlado deja que el DOM guarde el valor y se lee con una ref.

En el componente controlado tú eres el dueño del dato: value viene del estado y onChange lo actualiza con cada tecla, así que siempre sabes lo que hay en el campo. En el no controlado no pones value: el DOM guarda lo que el usuario escribe y lo lees con una ref cuando lo necesitas. La opción b) es falsa: el controlado hace más trabajo por render, no es más rápido. La c) es falsa: los no controlados funcionan bien en formularios simples. La d) ignora una diferencia real e importante. El libro recomienda controlados cuando necesitas validar, formatear o reaccionar a cada cambió, y no controlados para formularios muy simples donde solo lees los datos al enviar.

RESPUESTA 25: b) Para cancelar peticiones o suscripciones pendientes cuando el componente se desmonta o cambian las dependencias, y evitar actualizar el estado de un componente desmontado.

La función que retornas en un useEffect es la limpieza: React la ejecuta antes de volver a correr el efecto o cuando el componente se desmonta. Si hiciste un fetch y el usuario navegó a otra página antes de que respondiera, sin limpieza el .then intentaría hacer setState en un componente que ya no existe, lo que causa advertencias y posibles fugas de memoria. Con un

AbortController puedes cancelar la petición, o con una bandera ignorar la respuesta tardía. La opción a) es falsa: la limpieza es opcional y solo la usas cuando el efecto deja algo pendiente. La c) habla de la caché del navegador, que no tiene que ver. La d) confunde con reiniciar estado, que se hace con el setter.

RESPUESTA 26: FALSO. En JSX no existe el atributo `class` para estilos porque `class` es una palabra reservada de JavaScript que se usa para declarar clases. React te obliga a usar `className` en su lugar, y si escribes `class` verás una advertencia en la consola y el estilo no se aplicará como esperas. Este es el error número uno de quien llega de HTML puro y por eso el libro lo repite tanto. La regla es simple: todo lo que en HTML es `class`, en JSX es `className`. Por la misma razón el atributo `for` de las etiquetas `label` se convierte en `htmlFor`. No es un capricho de React: como JSX se compila a JavaScript real, no puede usar palabras que el lenguaje ya tiene apartadas. Acostúmbrate desde el inicio y te ahorrarás muchos dolores de cabeza.

RESPUESTA 27: FALSO. Mutar el estado directamente jamás provoca un re-render. React detecta los cambios comparando la referencia anterior con la nueva, y si haces `push` sobre el mismo arreglo o asignas una propiedad del mismo objeto, la referencia no cambia y React asume que nada pasó. El dato sí cambió en memoria, pero la pantalla se queda igual, lo cual es confuso porque un `console.log` te muestra el valor nuevo. La forma correcta es crear siempre un valor nuevo: un arreglo nuevo con `[...lista, elemento]` o un objeto nuevo con `{...obj, campo: valor}`, y pasarlo al setter. Este es uno de los errores más traicioneros porque no lanza ningún error visible; simplemente la interfaz no se actualiza y no sabes por qué.

RESPUESTA 28: VERDADERO. Cuando omites el segundo argumento de `useEffect`, el efecto se ejecuta después de cada render del componente, sin excepción. Esto incluye el primer render y todos los que provoquen los cambios de estado o de props. Por eso es tan peligroso poner un `setState` dentro de un efecto sin dependencias: cada ejecución actualiza el estado, eso provoca otro render y el efecto vuelve a correr, creando un bucle infinito. Si quieres que el efecto corra solo al montar, debes pasar el arreglo vacío `[]`. Si quieres que corra solo cuando cambien ciertos valores, pon esos valores en el arreglo. La ausencia del arreglo no significa "una vez", significa "siempre", y esa confusión causa la mayoría de los bucles infinitos.

RESPUESTA 29: VERDADERO. React solo usa la `key` para distinguir elementos entre hermanos dentro de la misma lista, así que basta con que sea única en ese nivel. Puedes repetir la misma `key` en dos listas distintas sin ningún problema porque cada lista se reconcilia por separado. Lo que sí debe cumplirse es que la `key` sea estable: el mismo elemento debe conservar la misma `key` entre renders, por eso se recomienda

usar un id de la base de datos y no el índice del arreglo cuando la lista puede reordenarse. Si usas índices y la lista cambia de orden, React confunde los elementos y aparecen errores visuales como inputs que muestran el valor de otra fila. Única entre hermanos y estable en el tiempo: esas son las dos reglas.

RESPUESTA 30: FALSO. Cambiar el valor de un useRef no provoca un re-render bajo ninguna circunstancia. El objeto que devuelve useRef es mutable a propósito: puedes escribir `ref.current = nuevoValor` cuantas veces quieras y React no se entera ni vuelve a pintar. Esa es precisamente su utilidad: guardar datos que deben sobrevivir entre renders (como el id de un intervalo o un contador de renders) sin disparar actualizaciones de la interfaz. Si necesitas que el cambio se refleje en pantalla, debes usar useState. Mucha gente confunde esto y espera ver el nuevo valor del ref en la interfaz de inmediato, pero el componente solo mostrará el valor actualizado hasta que ocurra un re-render por otro motivo. Recuerda: ref es para datos internos, state es para datos visibles.

RESPUESTA 31: VERDADERO. Todo custom hook debe empezar con "use" (por ejemplo `useContador` o `useFetch`) para que React y sus herramientas lo reconozcan como un hook. No es solo una convención de estilo: el linter de React usa ese prefijo para aplicarle las reglas de los hooks, como la de no llamarlo dentro de condicionales, y para detectar errores. Además, dentro de un custom hook puedes llamar a otros hooks como `useState` o `useEffect`, y eso solo está permitido porque el nombre con "use" lo identifica como hook. Si nombras tu función sin ese prefijo y llamas hooks dentro, React no podrá garantizar el orden de las llamadas y todo puede fallar de formas extrañas. Así que ya sabes: si tu función usa hooks por dentro, su nombre empieza con "use" sí o sí.

RESPUESTA 32: FALSO. React no cancela el envío del formulario por ti en ningún caso. Si no llamas a `e.preventDefault()` en el manejador `onSubmit`, el navegador hará lo que siempre ha hecho con los formularios: recargar la página y mandar los datos al servidor, destruyendo todo el estado de tu aplicación de una sola página. Que los inputs sean controlados no cambia este comportamiento, porque el envío del formulario es un evento nativo del navegador, no algo que React intercepte automáticamente. La línea `e.preventDefault()` es obligatoria en prácticamente todo formulario React y debe ser lo primero que hagas en el manejador. Sin ella, da igual qué tan bien validaste: al dar submit la página se reinicia y el usuario pierde todo lo que escribió.

RESPUESTA 33: VERDADERO. El argumento que pasas a `useState` solo se usa para construir el estado inicial en el primer render; en todos los renders siguientes React lo ignora y conserva el valor actual del estado. Esto tiene una consecuencia práctica importante: si calculas el valor inicial a partir de una prop y esa prop cambia después, el

estado no se actualiza solo. Mucha gente espera que `useState(conteolnicial)` se sincronice con la prop, pero no es así. Si necesitas que el estado reaccione a cambios de props, debes hacerlo explícitamente con un `useEffect` o manejar el valor como derivado. Otro detalle: si el valor inicial requiere un cálculo costoso, pásalo como función `useState(() => calculoPesado())` para que solo se ejecute una vez y no en cada render.

RESPUESTA 34: FALSO. Los hooks jamás deben llamarse dentro de un `if`, un ciclo `for` o una función anidada; siempre van en el nivel superior del componente o del custom hook. La razón es que React identifica cada hook por el orden en que se llama en cada render, no por su nombre. Si un hook queda dentro de un condicional, en algunos renders se llamaría y en otros no, el orden se rompería y React mezclaría los estados de unos hooks con otros, provocando errores muy difíciles de rastrear. Esta es una de las dos reglas de los hooks (la otra es llamarlos solo desde componentes o custom hooks). Si necesitas lógica condicional, pon la condición dentro del hook o dentro del efecto, pero la llamada al hook siempre va arriba, sin condiciones.

RESPUESTA 35: VERDADERO. Esa es exactamente la definición de `useMemo`: recibe una función que calcula un valor y un arreglo de dependencias, guarda el resultado y solo vuelve a ejecutar el cálculo cuando alguna dependencia cambia. Si el componente se re-renderiza por otro motivo (por ejemplo cambió un estado no relacionado), `useMemo` devuelve el valor guardado sin repetir el trabajo. Esto solo vale la pena en cálculos realmente costosos, como filtrar u ordenar listas grandes, porque la memorización también tiene un costó. Un error común es envolver todo con `useMemo` "por si acaso"; el libro recomienda usarlo cuando hayas medido un problema real de rendimiento. No confundir con `useCallback`, que memoriza funciones, ni con `React.memo`, que memoriza componentes.

RESPUESTA 36: FALSO. `useContext` no elimina las props ni está pensado para eso. El contexto sirve para datos globales o muy compartidos (tema visual, usuario autenticado, idioma) que de otro modo tendrías que pasar por muchos niveles de componentes. Pero para la comunicación normal entre un padre y su hijo directo, las props siguen siendo la herramienta correcta: son explícitas, fáciles de rastrear y hacen evidente qué datos necesita cada componente. Abusar del contexto para todo vuelve el código difícil de seguir, porque ya no ves de dónde viene cada dato, y además puede causar re-renders innecesarios en todos los consumidores cuando el valor cambia. La regla práctica es: props por defecto para el flujo padre-hijo, contexto solo para datos verdaderamente globales.

RESPUESTA 37: VERDADERO. Los paréntesis ejecutan la función en ese mismo momento, así que `onClick={miFuncion()}` llama a `miFuncion` durante el render y asigna su valor de retorno como manejador del clic. Como la mayoría de las funciones no retornan nada útil, el botón queda sin acción real al hacer clic, y peor aún: si la función actualiza el estado, provocas renders en cascada o un bucle infinito. Lo correcto es pasar la referencia sin paréntesis: `onClick={miFuncion}`. Si necesitas pasar argumentos, envuélvela en una función flecha: `onClick={() => miFuncion(id)}`. Este es el error más común de la unidad de eventos y aparece en casi todos los exámenes porque es fácil de cometer y difícil de notar a simple vista: el código "se ve bien" pero se comporta mal.

RESPUESTA 38: VERDADERO. El componente `Link` de `React Router` navega usando el historial del navegador sin recargar el documento completo, que es justamente la gracia de una aplicación de una sola página. Si usaras una etiqueta `<a>` normal, el navegador haría una petición completa al servidor, recargaría todo y perderías el estado de tu app. `Link` en cambio intercepta el clic, actualiza la URL y le dice al Router que muestre el componente de la nueva ruta, todo en memoria y al instante. Por eso en cualquier proyecto con `React Router` debes reemplazar tus `<a href>` internos por `<Link to>`. Las etiquetas `<a>` solo se justifican para enlaces externos que salen de tu aplicación. Esta diferencia es clave para entender por qué una SPA se siente tan rápida.

RESPUESTA 39: VERDADERO. `useParams` lee exactamente los segmentos dinámicos que definiste en la ruta con dos puntos. Si tu ruta es `/producto/:id` y el usuario visita `/producto/42`, el hook te devuelve el objeto `{ id: "42" }`. Con ese valor normalmente buscas el detalle en tu API o en tu lista de datos dentro de un `useEffect`. Es importante recordar dos cosas: primero, los valores siempre llegan como strings, así que convierte a número si lo necesitas; segundo, el nombre de la propiedad debe coincidir exactamente con el que pusiste en la ruta (`:id` se lee como `params.id`). No lo confundas con los query strings (`?color=rojo`), que se leen con `useSearchParams`, ni con la navegación programática, que se hace con `useNavigate`.

RESPUESTA 40: FALSO. Es al revés: el input no controlado es el que no usa el estado de React. En un no controlado no pones la prop value; el navegador (el DOM) guarda lo que el usuario escribe y tú lo lees cuando lo necesitas, normalmente con una ref, usando defaultValue solo como valor inicial. El que guarda su valor en el estado y se actualiza con onChange es el input controlado. Esta confusión es común porque los nombres se parecen, pero la diferencia es clara: controlado significa que React controla el valor en todo momento; no controlado significa que React se desentiende y el DOM manda. Para formularios donde necesitas validar cada tecla usa controlados; para formularios simples donde solo lees al enviar, los no controlados con ref son menos código.

RESPUESTA 41: Una solución correcta es la siguiente:

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

function Contador() {
  const [cuenta, setCuenta] = useState(0);

  function sumar() {
    setCuenta(cuenta + 1);
  }

  function restar() {
    if (cuenta > 0) {
      setCuenta(cuenta - 1);
    }
  }

  return (
    <div>
      <p>Cuenta: {cuenta}</p>
      <button onClick={sumar}>Sumar</button>
      <button onClick={restar}>Restar</button>
    </div>
  );
}

export default Contador;
```

Este ejercicio junta tres ideas del libro: declarar estado con useState, actualizarlo solo con el setter y conectar eventos con onClick pasando la referencia de la función sin paréntesis. El detalle del límite en cero se resuelve con un simple if antes de restar; otra forma válida es usar

Math.max(0, cuenta - 1). Fíjate que sumar y restar son funciones normales dentro del componente, y que el valor se muestra entre llaves en el JSX. Si en tu solución mutaste la variable directamente o ejecutaste la función en el onClick con paréntesis, repasa las unidades de useState y eventos, porque esos dos errores son los que más puntos cuestan en proyectos reales.

RESPUESTA 42: Los 3 errores son: 1) `onClick={agregar()}` ejecuta la función durante el render por los paréntesis; debe ser `onClick={agregar}`. 2) `tareas.push("Nueva tarea")` muta el estado directamente; React no detecta el cambio porque la referencia del arreglo es la misma y no vuelve a renderizar. Debe usarse el setter con un arreglo nuevo. 3) Falta la prop `key` en los `<li>` de la lista; sin `key` React no identifica cada elemento y muestra una advertencia. Código corregido:

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

  function agregar() {
    setTareas([...tareas, "Nueva tarea"]);
  }

  return (
    <div>
      <button onClick={agregar}>Agregar</button>
      <ul>
        {tareas.map((tarea) => (
          <li key={tarea}>{tarea}</li>
        ))}
      </ul>
    </div>
  );
}
```

Si encontraste los tres, dominas los errores más comunes del curso: eventos que se disparan solos, estado mutado que no actualiza la pantalla y listas sin `key`. Cada uno aparece en casi todos los proyectos, así que vale la pena tenerlos siempre presentes al escribir componentes.

RESPUESTA 43: La `key` es el identificador que React usa para distinguir cada elemento de una lista entre un render y otro. Cuando la lista cambia, React compara las `keys` para saber qué se agregó, qué se quitó, qué se movió y qué solo cambió de contenido; así reutiliza el DOM en vez de recrearlo todo, más rápido y conservando el estado de cada fila. El problema de usar el índice del arreglo como `key` aparece cuando la lista cambia de orden o se agregan o quitan elementos: el índice de cada elemento cambia, así que

React cree que son elementos distintos y los confunde. El síntoma típico es que los inputs de una fila muestran el valor de otra, se pierden los datos escritos o las animaciones se rompen. Por eso la key debe ser única entre hermanos y estable en el tiempo, idealmente un id de la base de datos.

RESPUESTA 44: Una solución correcta es la siguiente:

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

function Usuarios() {
  const [usuarios, setUsuarios] = useState([]);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("https://api.ejemplo.com/usuarios")
      .then((res) => res.json())
      .then((datos) => {
        setUsuarios(datos);
        setCargando(false);
      })
      .catch(() => {
        setError("No se pudieron cargar los usuarios");
        setCargando(false);
      });
  }, []);

  if (cargando) return <p>Cargando...</p>;
  if (error) return <p>{error}</p>;

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

Este es el patrón completo de fetch en React: tres estados (datos, cargando y error), la petición dentro de un useEffect con arreglo vacío para que corra una sola vez al montar, y render condicional para cada fase. Si pusiste el fetch fuera del efecto, la petición se repetiría en cada render. Si olvidaste el estado de error, la app se quedaría en "Cargando..." para

siempre cuando falle la red. Y no olvides la key en la lista: es parte del patrón y el libro la exige siempre. Como mejora extra, en proyectos reales conviene revisar `res.ok` antes de convertir a JSON y usar una función de limpieza con `AbortController` para cancelar la petición si el componente se desmonta.

RESPUESTA 45: Una solución correcta es la siguiente:

```

import { useState } from "react";

function Registro() {
  const [nombre, setNombre] = useState("");
  const [correo, setCorreo] = useState("");
  const [error, setError] = useState("");
  const [exito, setExito] = useState(false);

  function manejarEnvio(e) {
    e.preventDefault();
    if (nombre.trim() === "") {
      setError("El nombre no puede estar vacío");
      return;
    }
    if (!correo.includes("@")) {
      setError("El correo debe contener un @");
      return;
    }
    setError("");
    setExito(true);
  }

  return (
    <form onSubmit={manejarEnvio}>
      <input
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
        placeholder="Nombre"
      />
      <input
        value={correo}
        onChange={(e) => setCorreo(e.target.value)}
        placeholder="Correo"
      />
      <button type="submit">Registrar</button>
      {error && <p>{error}</p>}
      {exito && <p>Registro exitoso</p>}
    </form>
  );
}

```

Aquí se juntan los inputs controlados (value más onChange), el preventDefault para que no se recargue la página y la validación al enviar con mensajes de error en el estado. El trim() evita que un nombre de puros espacios pase la validación. Si tu solución valida solo con HTML o

deja que la página se recargue, repasa la unidad de formularios: en React la validación vive en tu código, no en el navegador. También es buena práctica validar mientras el usuario escribe para darle retroalimentación inmediata, y siempre volver a validar en el servidor porque la validación del cliente se puede brincar. Fíjate que cada input tiene su propio estado y su propio onChange: ese es el patrón de inputs controlados que el libro recomienda para formularios.

RESPUESTA 46: Una solución correcta es la siguiente:

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

function useContador(valorInicial) {
  const [valor, setValor] = useState(valorInicial);

  function incrementar() {
    setValor(valor + 1);
  }

  function decrementar() {
    setValor(valor - 1);
  }

  return { valor, incrementar, decrementar };
}

function MiContador() {
  const { valor, incrementar, decrementar } = useContador(10);

  return (
    <div>
      <p>Valor: {valor}</p>
      <button onClick={incrementar}>Más</button>
      <button onClick={decrementar}>Menos</button>
    </div>
  );
}
```

Un custom hook es una función cuyo nombre empieza con "use" y que por dentro usa otros hooks; su trabajo es encapsular lógica reutilizable. Aquí useContador esconde el useState y las dos funciones, y cualquier componente puede usarlo pasando su propio valor inicial. Fíjate que el hook retorna un objeto con lo que el componente necesita y que cada componente que lo usa tiene su propio estado independiente. Si tu hook no empieza con "use" o lo llamaste dentro de un condicional, repasa las reglas de los hooks: el nombre con prefijo es lo que le dice a React y al linter que esa función es un hook.

RESPUESTA 47: Una solución correcta es la siguiente:

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

const TemaContext = createContext("claro");

function ProveedorTema({ children }) {
  return (
    <TemaContext.Provider value="oscuro">
      {children}
    </TemaContext.Provider>
  );
}

function Hijo() {
  const tema = useContext(TemaContext);
  return <p>El tema actual es: {tema}</p>;
}

function App() {
  return (
    <ProveedorTema>
      <Hijo />
    </ProveedorTema>
  );
}
```

El flujo del contexto tiene tres pasos: crear el contexto con `createContext`, envolver a los componentes con el `Provider` pasándole el `value`, y leerlo abajo con `useContext`. Así el componente `Hijo` recibe "oscuro" sin que ningún componente intermedio tenga que pasar props (adiós al prop drilling). Fíjate que el proveedor usa la prop especial `children` para envolver lo que sea. Si en tu solución pasaste el tema por props a través de varios niveles, funciona, pero pierdes el punto del ejercicio: el contexto existe justo para evitar ese reenvío manual. Úsalo para datos globales como tema, usuario o idioma, no para todo.

RESPUESTA 48: Un bucle infinito en `useEffect` ocurre cuando el efecto provoca, directa o indirectamente, que él mismo vuelva a ejecutarse sin parar. El caso clásico es un efecto sin arreglo de dependencias que actualiza el estado: corre en cada render, el `setState` provoca otro render y el efecto vuelve a correr. Ejemplo que falla:

```
useEffect(() => {
  setContador(contador + 1);
});
```

Versión corregida (corre una sola vez al montar):

```
useEffect(() => {
  setContador(contador + 1);
}, []);
```

También puede pasar con dependencias mal puestas: si el efecto depende de un objeto que se recrea en cada render, el efecto corre siempre. La lección es que el arreglo de dependencias debe reflejar honestamente qué valores usa el efecto: vacío para correr una vez, con valores para correr cuando cambien, y nunca omitirlo si el efecto actualiza el estado. Cuando tu app se congela o ves miles de renders en la consola, el primer sospechoso siempre es un `useEffect`.

RESPUESTA 49: Una solución correcta es la siguiente:

```
import { BrowserRouter, Routes, Route, useParams } from "react-router-dom";

function DetalleProducto() {
  const { id } = useParams();
  return <p>Estás viendo el producto con id: {id}</p>;
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Inicio />} />
        <Route path="/producto/:id" element={<DetalleProducto />} />
      </Routes>
    </BrowserRouter>
  );
}
```

En React Router v6 las rutas se declaran con `Route` dentro de `Routes`, usando la prop `element` con el componente ya instanciado (`<Inicio />`), no con la sintaxis vieja `component={Inicio}` de la v5. El segmento dinámico se marca con dos puntos (`:id`) y `useParams` lo lee devolviendo un objeto con ese nombre; aquí se desestructura directamente `const { id } = useParams()`. Recuerda que el valor llega como string y que todo debe estar envuelto en `BrowserRouter`

para que los hooks del router funcionen. Si tu ruta de detalle no tenía los dos puntos o usaste useParams fuera del Router, repasa la unidad de React Router.

RESPUESTA 50: Una solución correcta es la siguiente:

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

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

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

  return (
    <div>
      <label>Buscar:</label>
      <input ref={inputRef} placeholder="Escribe aquí..." />
    </div>
  );
}
```

Este es el uso más clásico de useRef: guardar una referencia directa a un elemento del DOM. Conectas la ref al input con la prop ref={inputRef} y en un useEffect con arreglo vacío (que corre una vez al montar) llamas a inputRef.current.focus(). El efecto es necesario porque la ref solo existe después de que React pinta el elemento; si intentas el focus durante el render, current todavía es null. Fíjate que cambiar la ref no provoca re-renders, y aquí no los necesitamos. Si en tu solución usaste document.getElementById, funciona pero rompe la filosofía de React: con refs evitas salirte del modelo de componentes y tu código queda limpio y predecible.

Colibrí Editorial EGA

Curso Completo de React

Curso completo de React: componentes, JSX, props, estado, efectos, formularios, contexto, rutas y proyectos reales, con prácticas resueltas y examen final.

lucasegaacademy.com

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México