React avanzado: rendimiento, React 19, pruebas y arquitectura

Creado por Manuel Herrera Mesén · 28 de noviembre de 2026

Parte 3 de 4 · react · typescript · rendimiento · testing · frontend · serie-react

EN ESTE ARTÍCULO

- Cómo trabaja React por dentro
- Hooks propios
- Refs y el DOM
- TypeScript avanzado para React
- Rendimiento
- React 19 a fondo
- Pruebas
- Arquitectura, calidad y seguridad
- Panorama y camino a senior
- Lo que aprendiste
- Qué sigue
- Bibliografía

> Parte 3 de 4 de la serie «Aprende React construyendo una app real». Viene de la parte 2: React intermedio.

> **Nota:** Proyecto educativo de fans, sin fines de lucro. No está afiliado, patrocinado ni avalado por Nintendo, Game Freak ni The Pokémon Company. Pokémon y sus nombres son marcas de sus respectivos dueños. Datos servidos por PokéAPI.

La mini Pokédex de la parte 2 pide datos, navega, valida formularios y recuerda preferencias. Y un día pasa algo raro: cambias de Pokémon y el campo de notas conserva lo que escribiste para el anterior; un efecto corre aunque nada de lo que usa cambió; la lista tarda en responder a cada tecla. Ninguno de esos problemas se ve leyendo el componente. Para entenderlos hace falta saber qué hace React por dentro, y para no repetirlos, medir y probar.

De eso trata esta parte: el modelo de cómo renderiza React, las herramientas para escribir código que otros puedan reutilizar, medir y probar, y lo que separa un ejercicio de una aplicación profesional.

Cómo leer esta guía. Igual que en las partes anteriores: el texto principal va paso a paso, y los recuadros **Para profundizar** tienen el detalle fino. Esta parte supone que hiciste las partes 1 y 2; cuando un concepto viene de ellas, el texto enlaza la sección donde se explicó. Cuando dice «la prueba del ejemplo», habla de las pruebas automáticas con las que comprobé cada ejemplo: en «Pruebas» aprendes a escribirlas, y ahí se muestran varias.

Todo el código lo ejecuté con React 19.3.1, TypeScript 7.0.2, TanStack Query 5.108.0, Vitest 5.1.0, Testing Library 16.3.3 y MSW 3.0.7: cada ejemplo tiene pruebas automáticas que pasan, y los mensajes de error que ves son los que dieron las herramientas. Las pruebas de punta a punta de Dex 506 las corrí con Playwright 1.65.1 en Chromium.

Los ejemplos siguen en el proyecto de Vite de las partes anteriores. Esta parte no suma bibliotecas a la app: Vitest, Testing Library y MSW se instalan como dependencias de desarrollo en «Pruebas». Para ver un ejemplo, ponlo en el **`Layout`** de la parte 2, debajo de **`<Outlet />`**. Si recibe props, pásale valores de prueba: **`onSearch={console.log}`**, **`onSave={console.log}`** o, para un **`save`** que devuelve una promesa, **`save={() => new Promise((resolve) => setTimeout(resolve, 500))}`**. Los archivos de error, como **`list-typo.tsx`**, existen para mostrar el mensaje del compilador: no hace falta copiarlos.

Al terminar esta parte podrás:

1. explicar cuándo y por qué React vuelve a renderizar un componente;
2. diseñar hooks propios que otros puedan reutilizar;
3. usar refs para tocar el DOM o guardar valores sin renderizar;
4. tipar componentes y hooks con TypeScript avanzado;
5. medir el rendimiento antes de optimizarlo, React Compiler incluido;
6. usar las APIs de React 19;
7. probar tu código en tres niveles y organizar un proyecto real con criterios de calidad y seguridad.

Cómo trabaja React por dentro

En esta sección vas a construir el modelo mental que explica casi todos los «comportamientos raros» de React.

Render y commit

En la parte 1 viste que renderizar es llamar al componente para saber qué mostrar. Ahora hace falta precisarlo, porque una actualización de la pantalla tiene dos pasos:

- La **fase de render**: React llama a tus componentes y calcula cómo debería verse la pantalla. No toca el DOM. Por eso el render tiene que ser puro: React puede llamarlo, descartarlo y volver a llamarlo.
- La **fase de commit**: React aplica en el DOM las diferencias que calculó en el render, comparando el resultado nuevo con el anterior. Después corren los efectos.

Que un componente renderice no significa que el DOM cambie: si el resultado es igual al anterior, el commit no toca nada.

Qué hace que un componente renderice

Un componente vuelve a renderizar por tres motivos:

1. **cambió su estado**, con un **`set`** de **`useState`** o un **`dispatch`** de **`useReducer`**;
2. **renderizó su padre**: por defecto, cuando un componente renderiza, renderizan los componentes que él crea en su **`return`**, aunque sus props sean las mismas;
3. **cambió un Context que lee**, como viste en «Context».

El segundo es el que más sorprende. Para verlo, este **`logRender`** escribe en la consola cada render que le avisa un **`Profiler`**, el componente de React que mide renders y que conociste en la parte 2:

src/render/logRender.ts

```ts
import type { ProfilerOnRenderCallback } from 'react'

export const logRender: ProfilerOnRenderCallback = (id, phase, actualDuration) => {
  console.info(`render de ${id} (${phase}): ${actualDuration.toFixed(1)} ms`)
}
```

src/render/ParentRenders.tsx

```tsx
import { Profiler, useState } from 'react'
import { logRender } from './logRender'

function TrainerBadge() {
  return <p>Entrenadora: Ana</p>
}

export function ParentRenders() {
  const [views, setViews] = useState(0)
  return (
    <>
      <button onClick={() => setViews(views + 1)}>Vistas: {views}</button>
      <Profiler id="TrainerBadge" onRender={logRender}>
        <TrainerBadge />
      </Profiler>
    </>
  )
}
```

`TrainerBadge` no recibe props ni tiene estado, y aun así cada clic en «Vistas» lo vuelve a renderizar: la prueba del ejemplo registra un **`mount`** y después un **`update`** por clic. **`phase`** dice si fue el primer render (**`mount`**) o uno posterior (**`update`**), y **`actualDuration`**, cuánto tardó.

Casi siempre eso está bien: renderizar es barato, y el commit solo toca lo que cambió. En «Rendimiento» vas a ver cuándo deja de estar bien y qué hacer.

> – Para profundizar
>
> el **`Profiler`** mide en desarrollo. En el build de producción está apagado, salvo que uses un build especial de perfilado (React, *Profiler*).

> – Para profundizar
>
> un componente que llega como **`children`** ya lo creó otro componente. Si el que lo recibe renderiza por su propio estado, **`children`** es el mismo elemento de antes, y React no lo vuelve a renderizar. Pasar componentes como **`children`** es una forma de evitar renders sin memoizar nada.

El estado vive en la posición del árbol

Para decidir qué cambió, React compara el árbol nuevo con el anterior posición por posición: si en el mismo lugar hay un componente del mismo tipo, lo conserva, con su estado. Ese proceso es la **reconciliación** que viste en la parte 1. La consecuencia: el estado no pertenece a los datos que muestra el componente, sino a su lugar en el árbol.

Este campo de notas guarda lo que escribes en su propio estado:

src/render/NoteField.tsx

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

export function NoteField({ creature }: { creature: string }) {
  const [note, setNote] = useState('')
  return (
    <label>
      Nota sobre {creature} <input value={note} onChange={(event) =>
setNote(event.target.value)} />
    </label>
  )
}
```

ANTES

src/render/NotesWithoutKey.tsx

```tsx
import { useState } from 'react'
import { NoteField } from './NoteField'

export function NotesWithoutKey() {
  const [creature, setCreature] =
useState('brotin')
```

DESPUÉS

src/render/NotesWithKey.tsx

```tsx
import { useState } from 'react'
import { NoteField } from './NoteField'

export function NotesWithKey() {
  const [creature, setCreature] =
useState('brotin')
```

```
  return (
    <>
      <button onClick={() =>
setCreature(creature === 'brotin' ?
'lagunix' : 'brotin')}>Cambiar</button>
      <NoteField creature={creature} />
    </>
  )
}
```

```
  return (
    <>
      <button onClick={() =>
setCreature(creature === 'brotin' ?
'lagunix' : 'brotin')}>Cambiar</button>
      <NoteField key={creature} creature=
{creature} />
    </>
  )
}
```

En el bloque «antes», escribes una nota sobre Brotín, cambias a Lagunix y la nota sigue ahí: para React es el mismo **`NoteField`**, en el mismo lugar, solo que con otra prop. En el «después», la **`key={creature}`** le dice a React que cada criatura es un campo distinto. Cuando la **`key`** cambia, React descarta el componente anterior, con su estado, y crea uno nuevo: el campo empieza vacío. Es la misma **`key`** de las listas de la parte 1, usada fuera de una lista para reiniciar un componente.

Identidad: dos objetos iguales no son el mismo

Para saber si algo cambió (una dependencia de un efecto, el valor de un estado, las props de un componente memoizado, que verás en «Rendimiento»), React usa **`Object.is`**: compara números, textos y booleanos por su valor, pero objetos, arreglos y funciones por su **identidad**, es decir, si son el mismo objeto en memoria. Dos objetos creados por separado nunca son el mismo, aunque tengan los mismos datos.

Este componente arma un objeto de filtros en cada render y lo usa como dependencia de un efecto:

ANTES

src/render/TypeSearchWithObject.tsx

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

export function TypeSearchWithObject({
onSearch }: { onSearch: (filters: { type:
string }) => void }) {
  const [type, setType] =
useState('planta')
  const [views, setViews] = useState(0)
  const filters = { type }

  useEffect(() => {
    onSearch(filters)
  }, [filters, onSearch])

  return (
    <>
```

DESPUÉS

src/render/TypeSearch.tsx

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

export function TypeSearch({ onSearch }:
{ onSearch: (filters: { type: string })
=> void }) {
  const [type, setType] =
useState('planta')
  const [views, setViews] = useState(0)

  useEffect(() => {
    onSearch({ type })
  }, [type, onSearch])

  return (
    <>
      <button onClick={() =>
setType('fuego')}>Tipo fuego</button>
```

```
      <button onClick={() =>
setType('fuego')}>Tipo fuego</button>
      <button onClick={() =>
setViews(views + 1)}>Vistas: {views}
</button>
    </>
  )
}
```

```
      <button onClick={() =>
setViews(views + 1)}>Vistas: {views}
</button>
    </>
  )
}
```

En el «antes», cada render crea un **`filters`** nuevo, el efecto ve una dependencia distinta y vuelve a llamar a **`onSearch`**, aunque solo cambiaron las vistas. La prueba del ejemplo lo comprueba: un clic en «Vistas» produce una búsqueda de más. El linter ya lo marca:

```
⚠ react-hooks(exhaustive-deps): React hook useEffect depends on `filters`, which changes
every render
    ╭─[src/render/TypeSearchWithObject.tsx:10:7]
  5 │   const [views, setViews] = useState(0)
  6 │   const filters = { type }
    ·         ───┬───
    ·            ╰── `filters` is declared here
  7 │
  8 │   useEffect(() => {
  9 │     onSearch(filters)
 10 │   }, [filters, onSearch])
    ·       ───┬───
    ·          ╰── it will always cause this hook to re-evaluate
 11 │
    ╰────
  help: Try memoizing this variable with `useRef` or `useCallback`.

Found 1 warning and 0 errors.
Finished in N ms on 1 file with 116 rules using 2 threads.
```

Las dos herramientas que sugiere el linter las vas a ver más adelante, y para un objeto ninguna es la indicada: la que corresponde memoiza valores y está en «Rendimiento». La salida más simple es la del «después»: depender del valor primitivo, **`type`**, y armar el objeto dentro del efecto.

> *"Si vienes de Vue: el sistema de reactividad de Vue sabe qué componentes leen cada dato y vuelve a renderizar solo esos. En React, renderizar un componente renderiza a los que crea, y lo que evita eso es la composición con **`children`** o la memoización, que escribes tú o hace React Compiler, como verás en «Rendimiento»."*

Hooks propios

En esta sección vas a diseñar hooks que otros puedan reutilizar sin leer cómo están hechos.

Las reglas de los hooks

En la parte 2 escribiste tu primer hook propio, **`usePokemonList`**. Antes de diseñar más, conviene entender las dos reglas que cumplen todos los hooks (**`use`** es la excepción a la primera: se puede llamar dentro de un **`if`** o de un bucle, pero también solo desde componentes o hooks):

1. **Solo se llaman en el nivel superior** del componente o del hook: nunca dentro de un **`if`**, un bucle o una función interna.
2. **Solo se llaman desde componentes o desde otros hooks**, no desde funciones comunes.

La razón es cómo guarda React el estado: no por nombre, sino por orden. El primer **`useState`** del componente es la caja 1, el segundo, la caja 2. Si un hook se llama solo a veces, el orden cambia de un render al otro: React lo detecta y falla con un error («Rendered more hooks than during the previous render»), o, si la cantidad de hooks coincide, le da a cada uno el estado de otro. Este componente rompe la primera regla:

conditional-hook.tsx

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

export function Nickname({ editable }: { editable: boolean }) {
  if (!editable) return <p>Sin apodo</p>
  const [nickname, setNickname] = useState('')
  return <input value={nickname} onChange={(event) => setNickname(event.target.value)}
/>
}
```

```
× react-hooks(rules-of-hooks): React Hook "useState" is called conditionally. React
Hooks must be called in the exact same order in every component render.
   ╭─[conditional-hook.tsx:5:35]
 4 │   if (!editable) return <p>Sin apodo</p>
 5 │   const [nickname, setNickname] = useState('')
   ·                                   ──────┬──────
   ·                                         └── This Hook call is not reachable on
every render path.
 6 │   return <input value={nickname} onChange={(event) =>
setNickname(event.target.value)} />
   ╰────

  help: Move the Hook call before the condition, or call it unconditionally and branch
inside the Hook/effect instead.

Found 0 warnings and 1 error.
Finished in N ms on 1 file with 116 rules using 2 threads.
```

La corrección es llamar al hook siempre, antes de cualquier **`return`**, y decidir después qué mostrar.

useDebounce

Un buscador que pide datos en cada tecla hace cinco pedidos para escribir «lagun». Hacer **debounce** es esperar a que un valor deje de cambiar durante un tiempo antes de usarlo. Es lógica que no depende de qué buscas, así que va en un hook propio:

src/hooks/useDebounce.ts

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

export function useDebounce(value: string, delayMs: number): string {
  const [debounced, setDebounced] = useState(value)

  useEffect(() => {
    const timer = setTimeout(() => setDebounced(value), delayMs)
    return () => clearTimeout(timer)
  }, [value, delayMs])

  return debounced
}
```

Cada vez que **`value`** cambia, el efecto programa un temporizador; si **`value`** vuelve a cambiar antes de que se cumpla, la limpieza lo cancela y se programa otro. Solo cuando pasan **`delayMs`** milisegundos sin cambios, **`debounced`** toma el valor nuevo.

Componer hooks

Un hook propio puede usar otros hooks, incluidos los de una biblioteca. **`usePokemon`** envuelve la query de TanStack Query de la parte 2 en un nombre que dice qué hace:

src/hooks/usePokemon.ts

```ts
import { useQuery } from '@tanstack/react-query'
import { fetchPokemon } from '../api/pokeapi'

export function usePokemon(name: string) {
  return useQuery({
    queryKey: ['pokemon', name],
    queryFn: ({ signal }) => fetchPokemon(name, signal),
    enabled: name !== '',
  })
}
```

TanStack Query le pasa a **`queryFn`** una **`signal`** para cancelar el pedido, la misma idea del **`AbortController`** de la parte 2. **`enabled: name !== ''`** evita pedir un Pokémon sin nombre.

Con los dos hooks, el buscador queda así:

src/hooks/PokemonSearch.tsx

```tsx
import { useState } from 'react'
import { useDebounce } from './useDebounce'
import { usePokemon } from './usePokemon'

export function PokemonSearch() {
  const [text, setText] = useState('')
  const name = useDebounce(text.trim().toLowerCase(), 300)
  const pokemon = usePokemon(name)

  return (
    <>
      <label>
        Buscar <input value={text} onChange={(event) => setText(event.target.value)} />
      </label>
      {pokemon.isFetching && <p>Buscando…</p>}
      {pokemon.isError && <p role="alert">No encontré ese Pokémon.</p>}
      {pokemon.data && <h2>{`#${pokemon.data.id} ${pokemon.data.name}`}</h2>}
    </>
  )
}
```

El componente no sabe cómo se espera ni cómo se pide: lee de arriba abajo. La prueba del ejemplo escribe «Brotin» y cuenta un solo pedido, con el nombre completo. En el navegador, escribe el nombre de un Pokémon de verdad: las criaturas de las pruebas son inventadas.

Un hook mal diseñado

Este hook hace lo mismo que la búsqueda de arriba, pero mezcla todo:

src/hooks/usePokemonSearch.ts

```ts
import { useEffect, useState } from 'react'
import { fetchPokemon, type PokemonDetail } from '../api/pokeapi'

export function usePokemonSearch() {
  const [text, setText] = useState('')
  const [pokemon, setPokemon] = useState<PokemonDetail | null>(null)

  useEffect(() => {
    document.title = `Buscando ${text}`
    if (text === '') return
    fetchPokemon(text)
      .then(setPokemon)
      .catch(() => setPokemon(null))
  }, [text])

  return { text, setText, pokemon }
```

```
}
```

Funciona, pero, comparado con **`PokemonSearch`**:

- **hace tres trabajos**: guarda el texto, cambia el título de la pestaña y pide datos. Quien solo quiera la búsqueda se lleva el título;
- **es dueño del texto**: el componente no puede pasarle un texto que venga de otro lado, como la URL;
- **pide en cada letra y no cancela**: la prueba cuenta seis pedidos para «brotin», y una respuesta vieja puede pisar a una nueva, la carrera de la parte 2.

De la comparación salen las reglas para diseñar un hook: que haga **una sola cosa**; que reciba lo que necesita como **parámetros** en vez de guardarlo por dentro; que devuelva valores con nombre; y que no tenga efectos que quien lo usa no espera.

Refs y el DOM

En esta sección vas a guardar valores que no afectan la pantalla y a tocar el DOM cuando React no alcanza.

El problema: un valor que no es estado

Un cronómetro de entrenamiento necesita guardar el identificador del intervalo que lo hace avanzar, para poder pararlo después. Ese identificador no se muestra: guardarlo en un estado haría un render sin motivo. Y una variable común no sirve: se pierde, porque cada render vuelve a ejecutar la función del componente desde el principio.

useRef

Una **ref** es una caja que React guarda entre renders y que no hace renderizar cuando cambia. Técnicamente, **`useRef`** devuelve un objeto **`{ current }`**: el mismo objeto en todos los renders del componente, cuyo **`current`** puedes leer y cambiar cuando quieras, salvo durante el render.

src/refs/Stopwatch.tsx

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

export function Stopwatch() {
  const [seconds, setSeconds] = useState(0)
  const intervalId = useRef<number | null>(null)

  function start() {
    if (intervalId.current !== null) return
    intervalId.current = window.setInterval(() => setSeconds((current) => current + 1),
1000)
  }
```

```
  function stop() {
    if (intervalId.current === null) return
    window.clearInterval(intervalId.current)
    intervalId.current = null
  }

  useEffect(() => {
    return () => {
      if (intervalId.current !== null) window.clearInterval(intervalId.current)
      intervalId.current = null
    }
  }, [])

  return (
    <>
      <p>Entrenamiento: {seconds} s</p>
      <button onClick={start}>Empezar</button>
      <button onClick={stop}>Parar</button>
    </>
  )
}
```

- **`intervalId.current`** guarda el identificador mientras el cronómetro corre, y **`null`** cuando está parado: así «Empezar» dos veces no crea dos intervalos. La prueba del ejemplo lo comprueba.
- Los segundos sí son estado, porque se muestran.
- El efecto, con la lista de dependencias vacía, solo tiene limpieza: si el componente se desmonta con el cronómetro en marcha, para el intervalo y deja la ref en **`null`**, lista para volver a empezar.

> **Cuidado:** no leas ni cambies **`ref.current`** durante el render. El render tiene que ser puro, y una ref no avisa cuando cambia: la pantalla mostraría un valor viejo. Las refs se usan en manejadores de eventos y en efectos.

Refs al DOM

El otro uso de una ref es llegar a un elemento del DOM. Si pasas la ref en el atributo **`ref`** de un elemento, React guarda ahí el nodo del DOM después del commit. Con eso, este formulario lleva el foco al campo con error, algo que con props y estado no se puede hacer:

src/refs/TrainerForm.tsx

```
import { useId, useRef, useState, type SubmitEvent } from 'react'
import { flushSync } from 'react-dom'

export function TrainerForm({ onSave }: { onSave: (name: string) => void }) {
  const nameRef = useRef<HTMLInputElement>(null)
```

```
  const id = useId()
  const [error, setError] = useState<string | null>(null)

  function handleSubmit(event: SubmitEvent<HTMLFormElement>) {
    event.preventDefault()
    const name = String(new FormData(event.currentTarget).get('name')).trim()
    if (name === '') {
      flushSync(() => setError('Escribe tu nombre'))
      nameRef.current?.focus()
      return
    }
    setError(null)
    onSave(name)
  }

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor={`${id}-name`}>Nombre</label>
      <input
        id={`${id}-name`}
        name="name"
        ref={nameRef}
        aria-invalid={error ? true : undefined}
        aria-describedby={error ? `${id}-error` : undefined}
      />
      {error && <p id={`${id}-error`}>{error}</p>}
      <button>Guardar</button>
    </form>
  )
}
```

`flushSync`, de **`react-dom`**, obliga a React a aplicar el cambio de estado en el DOM antes de seguir. Sin él, el foco llegaría al campo antes de que tenga **`aria-invalid`** y **`aria-describedby`**, y un lector de pantalla lo anunciaría sin el error.

`useId` genera un identificador único y estable para cada instancia del componente. Con un **`id`** fijo como **`"name"`**, dos formularios en la misma página tendrían dos campos con el mismo **`id`**, y el **`htmlFor`** de la segunda etiqueta apuntaría al primer campo. Con **`useId`**, cada formulario tiene los suyos; la prueba del ejemplo monta dos y comprueba que no coinciden. No sirve para las **`key`** de una lista: esas salen de los datos.

ref como prop

A veces la ref tiene que llegar a un elemento que está dentro de otro componente. Desde React 19, un componente función recibe **`ref`** como una prop más (React, *React v19*):

src/refs/SearchInput.tsx

```tsx
import type { Ref } from 'react'

export function SearchInput({ label, ref }: { label: string; ref?: Ref<HTMLInputElement>
}) {
  return (
    <label>
      {label} <input ref={ref} />
    </label>
  )
}
```

src/refs/SearchAgain.tsx

```tsx
import { useRef } from 'react'
import { SearchInput } from './SearchInput'

export function SearchAgain() {
  const inputRef = useRef<HTMLInputElement>(null)
  return (
    <>
      <SearchInput label="Buscar" ref={inputRef} />
      <button onClick={() => inputRef.current?.focus()}>Buscar otra vez</button>
    </>
  )
}
```

`Ref<HTMLInputElement>` es el tipo de una ref que apunta a un **`<input>`**. Antes de React 19 esto pedía envolver el componente en **`forwardRef`**; el equipo de React anunció que lo va a deprecar.

useImperativeHandle

Pasar el **`<input>`** entero le da al padre acceso a todo el elemento. **`useImperativeHandle`** deja que el componente decida qué expone por su ref: aquí, solo **`focus`** y **`clear`**.

src/refs/LimitedSearch.tsx

```tsx
import { useImperativeHandle, useRef, type Ref } from 'react'

export type SearchHandle = { focus: () => void; clear: () => void }

export function LimitedSearch({ ref }: { ref?: Ref<SearchHandle> }) {
  const inputRef = useRef<HTMLInputElement>(null)

  useImperativeHandle(ref, () => ({
    focus: () => inputRef.current?.focus(),
    clear: () => {
      if (inputRef.current) inputRef.current.value = ''
    },
```

```
  }))

  return (
    <label>
      Buscar <input ref={inputRef} />
    </label>
  )
}
```

El padre crea la ref con **`useRef<SearchHandle>(null)`**, se la pasa como **`ref`** y llama, por ejemplo, **`searchRef.current?.clear()`** desde un manejador. Recibe un objeto con esos dos métodos y nada más, y el componente puede cambiar cómo está hecho por dentro sin romper a quien lo usa.

> – **Para profundizar**
>
> el atributo **`ref`** también acepta una función, una **callback ref**. React la llama con el nodo cuando el elemento aparece, y desde React 19 la función puede devolver una limpieza, que React llama cuando el elemento desaparece (React, *React v19*). Sirve, por ejemplo, para empezar a observar un elemento al montarlo y dejar de hacerlo al desmontarlo.

> *"Si vienes de Vue o Angular: la ref al DOM es la template ref de Vue (**`ref="input"`** y **`useTemplateRef`**) y el **`viewChild`** de Angular."*

TypeScript avanzado para React

En esta sección vas a usar los tipos para que los estados imposibles no compilen.

Genéricos

Ya usaste varios: **`useState<{ name: string } | null>`**, **`pokeApi.get<…>`**, **`z.infer<typeof MemberSchema>`**. Un **genérico** es un tipo que recibe otro tipo como parámetro, igual que una función recibe valores. Técnicamente, es un tipo o una función con un **parámetro de tipo** (por convención, **`T`**) que se fija cuando se usa.

Una lista que sirve para cualquier dato:

src/typescript/List.tsx

```
import type { ReactNode } from 'react'

type ListProps<T> = {
  items: T[]
  getKey: (item: T) => string
```

```
  renderItem: (item: T) => ReactNode
}

export function List<T>({ items, getKey, renderItem }: ListProps<T>) {
  return (
    <ul>
      {items.map((item) => (
        <li key={getKey(item)}>{renderItem(item)}</li>
      ))}
    </ul>
  )
}
```

src/typescript/Creatures.tsx

```
import { List } from './List'

type Creature = { id: string; name: string; level: number }

const TEAM: Creature[] = [
  { id: 'c1', name: 'Brotín', level: 12 },
  { id: 'c2', name: 'Lagunix', level: 9 },
]

export function Creatures() {
  return (
    <List
      items={TEAM}
      getKey={(creature) => creature.id}
      renderItem={(creature) => `${creature.name}, nivel ${creature.level}`}
    />
  )
}
```

`List<T>` no sabe qué hay en **`items`**. Cuando la usas, TypeScript **infiere `T`** a partir de lo que le pasas: como **`TEAM`** es un **`Creature[]`**, **`T`** es **`Creature`**, y dentro de **`getKey`** y **`renderItem`** cada **`creature`** es un **`Creature`**. Por eso un error de tipeo no llega a ejecutarse:

list-typo.tsx

```
import { List } from '../src/typescript/List'

type Creature = { id: string; name: string }

export function Creatures({ team }: { team: Creature[] }) {
  return <List items={team} getKey={(creature) => creature.id} renderItem={(creature) =>
creature.nmae} />
}
```

```
list-typo.tsx(6,99): error TS2339: Property 'nmae' does not exist on type 'Creature'.
```

> *"Así se ve en Dex 506: el **useDebounce** de la app es genérico. Sirve para un texto, un número o un objeto, y el tipo que devuelve es el mismo que recibe:"*

apps/web/src/shared/hooks/useDebounce.ts

```ts
/** The value, but only once it has stopped changing for `delayMs`: so we do not request on every key. */
export function useDebounce<T>(value: T, delayMs = 300): T {
  const [debounced, setDebounced] = useState(value)

  useEffect(() => {
    const timer = setTimeout(() => setDebounced(value), delayMs)
    return () => clearTimeout(timer) // if the value changes sooner, the previous wait is canceled
  }, [value, delayMs])

  return debounced
}
```

Uniones discriminadas

Un pedido puede estar cargando, haber fallado o haber terminado bien. Con tres campos sueltos, nada impide combinaciones que no tienen sentido:

ANTES

src/typescript/requestFlags.ts

```ts
type PokemonRequest = {
  isLoading: boolean
  isError: boolean
  data: { name: string } | null
}

export const impossible: PokemonRequest = { isLoading: true, isError: true, data: { name: 'brotin' } }
```

DESPUÉS

src/typescript/RequestView.tsx

```ts
export type RequestState<T> =
  | { status: 'loading' }
  | { status: 'error'; message: string }
  | { status: 'success'; data: T }

export function RequestView({ state }: { state: RequestState<{ name: string }> })
{
  switch (state.status) {
    case 'loading':
      return <p>Cargando…</p>
    case 'error':
      return <p role="alert">{state.message}</p>
    case 'success':
      return <h2>{state.data.name}</h2>
    default: {
```

```
      const unreachable: never = state
      return unreachable
    }
  }
}
```

`impossible` compila: cargando, con error y con datos a la vez. Una **unión discriminada** es una unión de objetos que comparten un campo con un valor literal distinto en cada uno, el **discriminante** (aquí, **`status`**). Al preguntar por **`state.status`**, TypeScript sabe en qué variante estás, el mismo narrowing de la parte 1: dentro de **`case 'error'`** existe **`message`**, y dentro de **`case 'success'`** existe **`data`**. Dentro de la variante **`error`** no hay **`data`** que leer.

El **`default`** hace un **chequeo exhaustivo**. **`never`** es el tipo que no tiene ningún valor: si todas las variantes se atendieron, a **`default`** no llega nada y **`state`** es **`never`**. Si mañana alguien suma una variante y no la atiende, **`state`** ya no es **`never`** y el compilador avisa:

new-status.tsx

```tsx
type RequestState<T> =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'error'; message: string }
  | { status: 'success'; data: T }

export function RequestView({ state }: { state: RequestState<{ name: string }> }) {
  switch (state.status) {
    case 'loading':
      return <p>Cargando…</p>
    case 'error':
      return <p role="alert">{state.message}</p>
    case 'success':
      return <h2>{state.data.name}</h2>
    default: {
      const unreachable: never = state
      return unreachable
    }
  }
}
```

```
new-status.tsx(16,13): error TS2322: Type '{ status: "idle"; }' is not assignable to
type 'never'.
```

Heredar las props de un elemento

Un botón propio debería aceptar todo lo que acepta un **`<button>`**: **`disabled`**, **`onClick`**, **`aria-label`**. **`ComponentProps<'button'>`** es el tipo de las props de un **`<button>`**:

src/typescript/Button.tsx

```tsx
import type { ComponentProps } from 'react'

type ButtonProps = ComponentProps<'button'> & { variant?: 'primary' | 'ghost' }

export function Button({ variant = 'primary', className, ...props }: ButtonProps) {
  return <button type="button" {...props} className={`button button-${variant}
${className ?? ''}`} />
}
```

El **&** suma la prop propia, **variant**, y **...props** junta todo lo demás para pasárselo al **<button>** con el spread de la parte 1. **type="button"** va antes del spread: por defecto, un **<button>** dentro de un formulario lo envía, y este botón no debería hacerlo salvo que quien lo usa pase otro **type**, que el spread pone encima.

Tipos utilitarios, satisfies y as const

TypeScript trae **tipos utilitarios**: genéricos que transforman otro tipo.

src/typescript/types.ts

```tsx
export type Pokemon = { id: number; name: string; types: string[]; height: number;
weight: number }

export type PokemonSummary = Pick<Pokemon, 'id' | 'name'>
export type NewPokemon = Omit<Pokemon, 'id'>
export type PokemonChanges = Partial<NewPokemon>
export type LevelsByName = Record<string, number>
```

Tipo	Qué hace
Pick<T, 'a' | 'b'>	se queda solo con esas propiedades
Omit<T, 'a'>	quita esas propiedades
Partial<T>	vuelve opcionales todas las propiedades
Record<K, V>	un objeto con claves de tipo **K** y valores de tipo **V**

satisfies comprueba que un valor cumple un tipo sin cambiar el tipo que TypeScript infiere de él. **as const** fija los valores literales de un arreglo u objeto, para que no se ensanchen a **string**:

src/typescript/types.ts

```tsx
export const TYPE_COLORS = {
  grass: '#3f9b4a',
  fire: '#d9572b',
  water: '#2f7fd1',
} satisfies Record<string, string>
```

```
export const VIEWS = ['grid', 'list'] as const
export type View = (typeof VIEWS)[number]
```

VIEWS es **readonly ['grid', 'list']**: un arreglo de solo lectura, con esos dos valores exactos. **typeof VIEWS** es su tipo, y **[number]** pide el tipo de cualquiera de sus elementos (el que tendría **VIEWS[0]** o **VIEWS[1]**): la unión **'grid' | 'list'**. La diferencia de **satisfies** con anotar el tipo se ve con un error de tipeo:

colors-typo.ts

```
const TYPE_COLORS = {
  grass: '#3f9b4a',
  fire: '#d9572b',
} satisfies Record<string, string>

const ANNOTATED: Record<string, string> = {
  grass: '#3f9b4a',
  fire: '#d9572b',
}

export const withSatisfies = TYPE_COLORS.grasss
export const withAnnotation = ANNOTATED.grasss
```

```
colors-typo.ts(11,42): error TS2551: Property 'grasss' does not exist on type '{ grass:
string; fire: string; }'. Did you mean 'grass'?
```

Solo hay un error. Con **satisfies**, TypeScript conserva las claves reales (**grass** y **fire**) y marca **grasss**. Con la anotación **Record<string, string>**, cualquier clave es válida, y el error de tipeo pasa sin aviso.

Tipos de eventos

Los manejadores reciben eventos con tipo: **SubmitEvent<HTMLFormElement>** para el **onSubmit** de un formulario (lo usaste en la parte 2), **ChangeEvent<HTMLInputElement>** para el **onChange** de un campo y **MouseEvent<HTMLButtonElement>** para un clic. Se importan de **react**, con **import type { MouseEvent } from 'react'**: sin ese import, **MouseEvent** y **SubmitEvent** son los del DOM, que no son genéricos. El genérico dice de qué elemento viene el evento, y por eso **event.currentTarget**, el elemento que tiene el manejador, tiene el tipo correcto. Cuando escribes el manejador en línea (**onChange={(event) => …}**), TypeScript lo infiere solo; cuando lo escribes aparte, como el **handleSubmit** de **TrainerForm**, le pones el tipo tú.

Rendimiento

En esta sección vas a medir antes de optimizar y a conocer las herramientas que evitan trabajo de más.

El problema: optimizar a ciegas

Optimizar sin medir es adivinar, y cada optimización tiene un costo: código más difícil de leer y comparaciones que también consumen tiempo. El orden es: medir, encontrar la parte lenta, optimizar esa parte y volver a medir.

Para medir tienes el **`Profiler`** de «Cómo trabaja React por dentro», la pestaña Profiler de las React DevTools (una extensión del navegador) y, desde React 19.2, las pistas de React en el panel Performance de Chrome DevTools (React, *React 19.2*).

memo

La **memoización** que viste en la parte 2 con **`createSelector`**, aplicada a componentes: **`memo`** envuelve un componente para que no vuelva a renderizar si sus props son iguales a las del render anterior, comparadas una por una con **`Object.is`**.

src/performance/CreatureList.tsx

```tsx
import { memo, Profiler } from 'react'
import { logRender } from '../render/logRender'

type CreatureListProps = { names: string[]; onSelect: (name: string) => void }

function CreatureListBase({ names, onSelect }: CreatureListProps) {
  return (
    <Profiler id="CreatureList" onRender={logRender}>
      <ul>
        {names.map((name) => (
          <li key={name}>
            <button onClick={() => onSelect(name)}>{name}</button>
          </li>
        ))}
      </ul>
    </Profiler>
  )
}

export const CreatureList = memo(CreatureListBase)
```

El **`Profiler`** va dentro de la lista a propósito: así solo avisa cuando la lista renderiza de verdad. Envolviendo a **`CreatureList`** desde afuera, avisaría cada vez que renderiza el padre, aunque **`memo`** se haya saltado la lista.

Por qué memo no alcanza

`memo` compara props, y la identidad de «Cómo trabaja React por dentro» vuelve a aparecer. Las dos páginas usan esta lista de nombres:

src/performance/names.ts

```
export const ALL_NAMES = ['brotin', 'fogarin', 'lagunix', 'chispon', 'rocadon',
'alavento']
```

ANTES

src/performance/PageWithNewProps.tsx

```
import { useState } from 'react'
import { CreatureList } from
'./CreatureList'
import { ALL_NAMES } from './names'

export function PageWithNewProps() {
  const [views, setViews] = useState(0)
  const [selected, setSelected] =
useState<string | null>(null)
  const names = ALL_NAMES.filter((name)
=> name !== selected)

  return (
    <>
      <button onClick={() =>
setViews(views + 1)}>Vistas: {views}
</button>
      <p>Elegido: {selected ?? 'ninguno'}
</p>
      <CreatureList names={names}
onSelect={(name) => setSelected(name)} />
    </>
  )
}
```

DESPUÉS

src/performance/PageWithStableProps.tsx

```
import { useCallback, useMemo, useState }
from 'react'
import { CreatureList } from
'./CreatureList'
import { ALL_NAMES } from './names'

export function PageWithStableProps() {
  const [views, setViews] = useState(0)
  const [selected, setSelected] =
useState<string | null>(null)
  const names = useMemo(() =>
ALL_NAMES.filter((name) => name !==
selected), [selected])
  const handleSelect = useCallback((name:
string) => setSelected(name), [])

  return (
    <>
      <button onClick={() =>
setViews(views + 1)}>Vistas: {views}
</button>
      <p>Elegido: {selected ?? 'ninguno'}
</p>
      <CreatureList names={names}
onSelect={handleSelect} />
    </>
  )
}
```

En el «antes», cada render crea un arreglo **`names`** nuevo con **`filter`** y una función nueva para **`onSelect`**. Las props nunca son iguales, y **`memo`** no evita nada: la prueba del ejemplo registra un render de la lista por cada clic en «Vistas». En el «después»:

- **`useMemo`** guarda el resultado de un cálculo y lo devuelve igual mientras sus dependencias no cambien: **`names`** es el mismo arreglo hasta que cambia **`selected`**;
- **`useCallback`** hace lo mismo con una función: **`handleSelect`** es la misma en todos los renders.

Ahora un clic en «Vistas» no renderiza la lista, y elegir una criatura sí, porque **`names`** cambió.

> **Cuidado:** memoizar cuesta: memoria para guardar los resultados, comparaciones en cada render y código más difícil de leer. Úsalo cuando una medición muestra un render lento, o cuando la identidad importa: props de un componente con **`memo`** o dependencias de un efecto.

React Compiler

React Compiler hace esa memoización por ti, en el build: analiza cada componente y hook, y memoiza los valores y los elementos que lo necesitan, sin **`memo`**, **`useMemo`** ni **`useCallback`** a mano. Su versión 1.0, estable, salió el 7 de octubre de 2025 (Lauren Tan, Joe Savona y Mofei Zhang, *React Compiler v1.0*). Supone que tu código cumple las reglas de React (render puro y hooks en orden): si detecta que un componente no las cumple, lo deja sin optimizar. **Babel** es un traductor de JavaScript que corre durante el build, y el compilador es un plugin de Babel. En Dex 506 lo corre el plugin **`@rolldown/plugin-babel`** dentro de la configuración de Vite; en la parte 4 lo instalas y ves el archivo completo, con sus imports:

apps/web/vite.config.ts

```
export default defineConfig({
  plugins: [
    react(),
    // React Compiler: memoizes components and values at build time, without memo or
useMemo by hand.
    // The decision to keep it on comes from the measurement in part 3 (perf/README.md).
    babel({ presets: [reactCompilerPreset()] }),
    tailwindcss(),
  ],
  server: {
    // In development, /api goes to the Express API. To the browser everything comes
from the same
    // origin (localhost:5173), so the session cookie travels without configuring CORS.
    proxy: {
      '/api': 'http://localhost:3001',
    },
  },
})
```

Los ejemplos de esta parte no lo usan, para que veas lo que pasa sin memoización.

¿Vale la pena? Lo medí en Dex 506 con el caso más caro de la app: escribir en el buscador de la Pokédex con los 151 Pokémon en pantalla, porque cada tecla vuelve a renderizar la página. Lo hice con el build de producción, la CPU cuatro veces más lenta y cinco corridas de diez teclas, midiendo cada tecla desde que se presiona hasta el cuadro siguiente. Hice dos rondas de mediciones:

Variante	Ronda 1: mediana · p75	Ronda 2: mediana · p75
Sin memoizar	9.9 ms · 10.8 ms	9.8 ms · 10.2 ms

Variante	Ronda 1: mediana · p75	Ronda 2: mediana · p75
`memo` a mano en la tarjeta	5.0 ms · 5.5 ms	4.9 ms · 5.7 ms
React Compiler	5.6 ms · 8.9 ms	5.1 ms · 5.5 ms

Apple M3 Pro, Chromium 159, 17 de noviembre de 2026. p75 es el percentil 75: tres de cada cuatro teclas tardaron eso o menos.

El compilador deja la mediana cerca de la mitad, igual que **`memo`** a mano, sin tocar el código de ningún componente. En la primera ronda su percentil 75 fue más alto (8.9 ms), una variación entre corridas que la segunda no repitió. Y hay que decirlo con todas las letras: sin memoizar, 9.8 ms ya entra en un cuadro de 16 ms (lo que dura un cuadro a 60 imágenes por segundo), así que en esta app la diferencia existe pero no se nota. Dex 506 deja el compilador activo por lo que pasa cuando la app crezca: nadie tiene que acordarse de memoizar.

Dividir el código y virtualizar

Dos técnicas más, para problemas que Dex 506 no tiene:

- **Code splitting** es dividir el JavaScript en partes que se descargan cuando hacen falta. Ya lo hiciste en la parte 2 con la carga diferida de la ficha: Vite pone cada **`import()`** dinámico en un archivo aparte.
- **Virtualizar** una lista es renderizar solo las filas que se ven y cambiarlas mientras se desplaza. Con miles de filas, es la diferencia entre trabarse y no; con los 151 de Dex 506 no hace falta, como mostró la medición. Bibliotecas como TanStack Virtual lo resuelven.

useDeferredValue y useTransition

En la parte 2 viste que una transición es un cambio que React puede hacer sin bloquear la pantalla. Ahora la definición completa: una actualización dentro de una **transición** es de baja prioridad, y si llega una urgente (una tecla, un clic) React descarta el render de la transición que tenía a medias, atiende la urgente y después lo empieza de nuevo.

`useDeferredValue` recibe un valor y devuelve una versión que se queda atrás mientras haya trabajo urgente:

src/performance/DeferredSearch.tsx

```
import { memo, useDeferredValue, useState } from 'react'

const Results = memo(function Results({ text, names }: { text: string; names: string[]
}) {
  const visible = names.filter((name) => name.includes(text))
  return (
    <ul>
      {visible.map((name) => (
        <li key={name}>{name}</li>
```

```
      ))}
    </ul>
  )
})

export function DeferredSearch({ names }: { names: string[] }) {
  const [text, setText] = useState('')
  const deferredText = useDeferredValue(text)
  const isStale = text !== deferredText

  return (
    <>
      <label>
        Buscar <input value={text} onChange={(event) => setText(event.target.value)} />
      </label>
      <div aria-busy={isStale} style={{ opacity: isStale ? 0.6 : 1 }}>
        <Results text={deferredText} names={names} />
      </div>
    </>
  )
}
```

El campo usa **`text`** y se actualiza en cada tecla; la lista usa **`deferredText`**, que React actualiza en segundo plano. Mientras son distintos, **`isStale`** atenúa la lista con un **`style`** en línea, un objeto con propiedades de CSS. **`Results`** va con **`memo`** a propósito: sin él, renderizaría con cada tecla igual, porque su padre renderiza.

`useTransition` sirve cuando el cambio lo haces tú, con un **`set`**. Este ejemplo cambia entre dos paneles de una ficha, estadísticas y movimientos, y el de movimientos tiene un filtro:

src/performance/panels.tsx

```
import { useState } from 'react'

export function StatsPanel() {
  return <p>PS 46 · Ataque 45 · Defensa 62</p>
}

export function MovesPanel() {
  const [filter, setFilter] = useState('')
  const moves = ['látigo', 'placaje', 'hoja afilada'].filter((move) =>
move.includes(filter))
  return (
    <>
      <label>
        Filtrar movimientos <input value={filter} onChange={(event) =>
setFilter(event.target.value)} />
      </label>
      <ul>
        {moves.map((move) => (
```

```
          <li key={move}>{move}</li>
        ))}
      </ul>
    </>
  )
}
```

src/performance/Tabs.tsx

```tsx
import { useState, useTransition } from 'react'
import { MovesPanel, StatsPanel } from './panels'

type Tab = 'stats' | 'moves'

export function DetailTabs() {
  const [tab, setTab] = useState<Tab>('stats')
  const [isPending, startTransition] = useTransition()

  function select(next: Tab) {
    startTransition(() => setTab(next))
  }

  return (
    <>
      <button aria-pressed={tab === 'stats'} onClick={() => select('stats')}>
        Estadísticas
      </button>
      <button aria-pressed={tab === 'moves'} onClick={() => select('moves')}>
        Movimientos
      </button>
      {isPending && <p>Cambiando de pestaña…</p>}
      {tab === 'stats' ? <StatsPanel /> : <MovesPanel />}
    </>
  )
}
```

`startTransition` marca el cambio de pestaña como transición, e **`isPending`** es **`true`** mientras React la prepara. La regla para elegir: si tienes el **`set`**, **`useTransition`**; si recibes el valor de otro lado (una prop, el resultado de otro hook), **`useDeferredValue`**.

Con ejemplos tan chicos no vas a ver la lista atenuada ni «Cambiando de pestaña…»: aparecen cuando la lista o el panel tardan en renderizar, como una lista de miles de elementos.

> *“Si vienes de Vue o Angular: el **`computed`** de Vue y el de Angular ya memoizan su resultado hasta que cambian sus dependencias. En React, **`useMemo`** es la versión que escribes a mano, y React Compiler, la automática.”*

React 19 a fondo

En esta sección vas a usar las APIs de React 19 que reemplazan lo que antes pedía una biblioteca o un efecto.

use, Suspense y los límites de error

En la parte 2 usaste **use** para leer un Context. También lee una promesa: si la promesa no terminó, el componente **suspende**, y el **Suspense** más cercano muestra su contenido de reemplazo.

src/react19/PokemonName.tsx

```
import { Suspense, use } from 'react'
import type { PokemonDetail } from '../api/pokeapi'
import { ErrorBoundary } from './ErrorBoundary'

function Name({ pokemonPromise }: { pokemonPromise: Promise<PokemonDetail> }) {
  const pokemon = use(pokemonPromise)
  return <h2>{`#${pokemon.id} ${pokemon.name}`}</h2>
}

export function PokemonName({ pokemonPromise }: { pokemonPromise: Promise<PokemonDetail>
}) {
  return (
    <ErrorBoundary fallback={<p role="alert">No se pudo cargar el Pokémon.</p>}>
      <Suspense fallback={<p>Cargando…</p>}>
        <Name pokemonPromise={pokemonPromise} />
      </Suspense>
    </ErrorBoundary>
  )
}
```

src/react19/PickPokemon.tsx

```
import { useState } from 'react'
import { fetchPokemon, type PokemonDetail } from '../api/pokeapi'
import { PokemonName } from './PokemonName'

export function PickPokemon() {
  const [request, setRequest] = useState<{ id: number; promise: Promise<PokemonDetail> }
| null>(null)

  function show() {
    const promise = fetchPokemon('3')
    setRequest((previous) => ({ id: (previous?.id ?? 0) + 1, promise }))
  }

  return (
    <>
```

```
      <button onClick={show}>Ver el Pokémon número 3</button>
      {request && <PokemonName key={request.id} pokemonPromise={request.promise} />}
    </>
  )
}
```

- **Suspense** muestra «Cargando…» mientras la promesa está pendiente y el nombre cuando termina.
- Si la promesa falla, el error sube hasta el **límite de error** (*error boundary*) más cercano, que muestra su reemplazo en lugar de dejar la pantalla en blanco.
- La promesa se crea en el clic, fuera del render. Si se creara en el render, cada render crearía una promesa nueva y el componente suspendería para siempre.
- Cada pedido lleva una **key** distinta: así, un pedido nuevo después de un error monta un límite de error nuevo, sin el error anterior. Es la **key** de «Cómo trabaja React por dentro».

El límite de error es el único caso de la serie en que vas a escribir una **clase**: hoy no hay forma de escribirlo como componente función (React, *Component*). Una clase es la forma de definir componentes anterior a las funciones: **`extends Component`** la convierte en componente, **`state`** es su estado, **`this.props`** son sus props, y **`static`** marca un método que pertenece a la clase y no a cada componente que se crea con ella.

src/react19/ErrorBoundary.tsx

```
import { Component, type ReactNode } from 'react'

type Props = { fallback: ReactNode; children: ReactNode }

export class ErrorBoundary extends Component<Props, { hasError: boolean }> {
  state = { hasError: false }

  static getDerivedStateFromError() {
    return { hasError: true }
  }

  render() {
    return this.state.hasError ? this.props.fallback : this.props.children
  }
}
```

`getDerivedStateFromError` es el método que React llama cuando un componente de adentro falla al renderizar; el estado que devuelve hace que **`render`** muestre el reemplazo. En un proyecto real, también puedes usar la biblioteca **`react-error-boundary`**, que la documentación de React menciona como alternativa. Un límite de error no atrapa errores de los manejadores de eventos ni del código asíncrono que React no ve, como un **`setTimeout`** o un **`.then`** suelto: esos los manejas tú, con **`.catch`** o con **`try`**/**`catch`** alrededor de un **`await`**.

> – Para profundizar
>
> el **`errorElement`** de React Router de la parte 2 es un límite de error que el router pone por ti en las rutas donde lo declaras.

useOptimistic

Cuando marcas un favorito, esperar la respuesta del servidor para mostrar la estrella hace que la app se sienta lenta. Una **actualización optimista** muestra el resultado antes de que el servidor confirme, y lo deshace si falla. **`useOptimistic`** la resuelve:

src/react19/OptimisticFavorite.tsx

```tsx
import { useOptimistic, useState, useTransition } from 'react'

type Props = { initial: boolean; save: (favorite: boolean) => Promise<void> }

export function OptimisticFavorite({ initial, save }: Props) {
  const [favorite, setFavorite] = useState(initial)
  const [shown, setShown] = useOptimistic(favorite)
  const [isPending, startTransition] = useTransition()
  const [failed, setFailed] = useState(false)

  function toggle() {
    setFailed(false)
    startTransition(async () => {
      setShown(!favorite)
      try {
        await save(!favorite)
        startTransition(() => setFavorite(!favorite))
      } catch {
        setFailed(true)
      }
    })
  }

  return (
    <>
      <button aria-pressed={shown} disabled={isPending} onClick={toggle}>
        {shown ? '★ Favorito' : '☆ Marcar'}
      </button>
      {failed && <p role="alert">No se pudo guardar el favorito.</p>}
    </>
  )
}
```

- **`useOptimistic(favorite)`** devuelve **`shown`**, que es igual a **`favorite`** salvo mientras hay una Action en curso, y **`setShown`**, que cambia **`shown`** solo durante esa Action.
- **`setShown`** tiene que llamarse dentro de una transición o una Action de React; por eso todo va en **`startTransition`**, que acepta funciones asíncronas.
- Si **`save`** termina bien, **`setFavorite`** guarda el valor real, y **`shown`** queda igual a él. Va en otro **`startTransition`** porque está después de un **`await`**, como en los ejemplos de la documentación de React.
- Si **`save`** falla, la transición termina sin que **`favorite`** haya cambiado, y React vuelve a mostrar el valor real: la estrella se apaga sola, sin código para deshacerla (React, *useOptimistic*). La prueba del ejemplo lo comprueba con un **`save`** que falla.

ViewTransition

Una **View Transition** es una animación del navegador entre dos estados de la página. Con **`<ViewTransition>`**, estable desde React 19.3 (React, *React 19.3*), React la dispara por ti:

src/react19/CardToDetail.tsx

```
import { startTransition, useState, ViewTransition } from 'react'

export function CardToDetail() {
  const [open, setOpen] = useState(false)
  return (
    <ViewTransition>
      {open ? (
        <article>
          <h2>Brotín</h2>
          <p>Planta. Le gusta el sol de la mañana.</p>
          <button onClick={() => startTransition(() => setOpen(false))}>Volver</button>
        </article>
      ) : (
        <button onClick={() => startTransition(() => setOpen(true))}>Ver a
Brotín</button>
      )}
    </ViewTransition>
  )
}
```

Solo anima las actualizaciones que van dentro de una transición (**`startTransition`**), de un **`Suspense`** o de un **`useDeferredValue`**; un **`set`** común cambia la pantalla sin animar (React, *ViewTransition*). Sin otras opciones, la animación es un fundido. En las pruebas automáticas, que corren en un DOM simulado sin la API de transiciones del navegador, el cambio ocurre igual, sin animación.

> *"Así se ve en Dex 506: la imagen de la ficha lleva el mismo **`name`** que la de su tarjeta en la Pokédex. Cuando una sale y la otra entra en la misma transición, el navegador anima la imagen de un lugar*

al otro:"

apps/web/src/features/pokemon/PokemonDetailPage.tsx

```
{/* The same name as in the Pokédex card: the browser animates the image from one
place to the other. */}
<ViewTransition name={`pokemon-${data.id}`}>
  <PokemonSprite src={data.artworkUrl ?? data.spriteUrl} alt={data.name} size={160} />
</ViewTransition>
```

Activity

Vuelve a las pestañas de «Rendimiento»: **`DetailTabs`** muestra un panel u otro con un ternario, así que cambiar de pestaña desmonta el otro panel y el filtro pierde lo escrito. **`<Activity>`**, que llegó con React 19.2 (React, *React 19.2*), oculta sin desmontar:

src/react19/ActivityTabs.tsx

```
import { Activity, useState } from 'react'
import { MovesPanel, StatsPanel } from '../performance/panels'

export function ActivityTabs() {
  const [tab, setTab] = useState<'stats' | 'moves'>('stats')
  return (
    <>
      <button aria-pressed={tab === 'stats'} onClick={() => setTab('stats')}>
        Estadísticas
      </button>
      <button aria-pressed={tab === 'moves'} onClick={() => setTab('moves')}>
        Movimientos
      </button>
      <Activity mode={tab === 'stats' ? 'visible' : 'hidden'}>
        <StatsPanel />
      </Activity>
      <Activity mode={tab === 'moves' ? 'visible' : 'hidden'}>
        <MovesPanel />
      </Activity>
    </>
  )
}
```

Con **`mode="hidden"`**, React oculta el contenido con **`display: none`** y conserva su estado y su DOM; también desmonta sus efectos, y los vuelve a montar al mostrarlo (React, *Activity*). La prueba del ejemplo escribe «hoja» en el filtro, cambia de pestaña, vuelve, y el texto sigue ahí; con **`DetailTabs`**, la misma prueba encuentra el campo vacío.

Metadatos del documento

En la parte 2 cambiaste el título de la pestaña con un efecto. En React 19, **`<title>`** y **`<meta>`** se pueden escribir dentro de cualquier componente, y React los pone en el **`<head>`** del documento (React, *title*; React, *meta*):

src/react19/PokemonHead.tsx

```tsx
export function PokemonHead({ name }: { name: string }) {
  return (
    <>
      <title>{`${name} · Mini Pokédex`}</title>
      <meta name="description" content={`La ficha de ${name}`} />
      <h1>{name}</h1>
    </>
  )
}
```

`<title>` acepta un solo texto: por eso va con una plantilla de texto y no con **`{name} · Mini Pokédex`**, que serían dos hijos. Y conviene que haya un solo **`<title>`** en pantalla a la vez.

Pruebas

En esta sección vas a probar tu código en tres niveles, con las mismas herramientas que comprobaron cada ejemplo de esta serie.

La pirámide de pruebas

Una prueba automática es código que usa tu código y comprueba que hace lo que debe; corre en segundos, en cada cambio, y avisa antes de que lo note quien usa la app.

Las pruebas se ordenan en tres niveles, de la base a la punta:

- Una **prueba unitaria** prueba una pieza sola, como un componente o una función, sin red ni servidor. Es rápida y señala exactamente qué falló.
- Una **prueba de integración** prueba varias piezas juntas: un componente con sus funciones de datos, axios y una red simulada.
- Una **prueba de punta a punta** (**e2e**, *end to end*) abre la app real en un navegador real y la usa como una persona. Es la que más confianza da y la más lenta.

Se llama **pirámide** porque conviene tener muchas de la base y pocas de la punta: las unitarias son baratas, y las e2e, caras de correr y de mantener.

Las herramientas

- **Vitest** corre las pruebas. Usa el mismo sistema de configuración que Vite, así que entiende TypeScript y JSX sin pasos extra. Si existe un **`vitest.config.ts`**, lo usa en lugar del **`vite.config.ts`**: por eso repite el plugin **`react()`**.
- **jsdom** simula el DOM del navegador dentro de Node, para que las pruebas rendericen componentes.
- **Testing Library** renderiza componentes y los consulta como lo haría una persona: por su rol, su etiqueta o su texto. **`user-event`**, de la misma familia, simula teclas y clics.

Se instalan como dependencias de desarrollo:

```
npm install -D vitest jsdom @testing-library/react @testing-library/user-event @testing-library/jest-dom msw
```

La configuración de Vitest va en su propio archivo, en la raíz del proyecto:

vitest.config.ts

```
import react from '@vitejs/plugin-react'
import { defineConfig } from 'vitest/config'

export default defineConfig({
  plugins: [react()],
  test: {
    environment: 'jsdom',
    setupFiles: ['./src/test/setup.ts'],
    testTimeout: 15_000,
  },
})
```

`environment: 'jsdom'` activa el DOM simulado, **`testTimeout`** le da a cada prueba hasta 15 segundos (la primera corrida es más lenta) y **`setupFiles`** apunta a un archivo que corre antes de las pruebas. Importa el **`server`** de MSW, que armas en «MSW: la red simulada», más abajo; hasta entonces, deja afuera las líneas que lo usan:

src/test/setup.ts

```
import '@testing-library/jest-dom/vitest'
import { cleanup } from '@testing-library/react'
import { afterAll, afterEach, beforeAll } from 'vitest'
import { server } from './server'

beforeAll(() => server.listen({ onUnhandledFrame: 'error' }))
afterEach(() => {
  cleanup()
  server.resetHandlers()
})
afterAll(() => server.close())
```

`@testing-library/jest-dom/vitest` suma comprobaciones como **`toBeInTheDocument`** o **`toHaveFocus`**; **`beforeAll`**, **`afterEach`** y **`afterAll`** corren algo antes de todas las pruebas del archivo, después de cada una y al final; **`cleanup`** desmonta lo que dejó cada prueba.

Para correrlas, **`npx vitest`** las ejecuta y se queda mirando los cambios, y **`npx vitest run`** las ejecuta una vez, como hace la **integración continua** (*CI*), el servidor que corre las pruebas en cada cambio que subes. Vitest encuentra por sí solo los archivos **`*.test.*`** y **`*.spec.*`** (**`.ts`**, **`.tsx`**, **`.js`**…).

Una prueba unitaria

La tarjeta de un Pokémon:

src/testing/PokemonCard.tsx

```
type Props = { name: string; number: number; isFavorite: boolean; onToggleFavorite: ()
=> void }

export function PokemonCard({ name, number, isFavorite, onToggleFavorite }: Props) {
  return (
    <article aria-label={name}>
      <p>#{String(number).padStart(3, '0')}</p>
      <h2>{name}</h2>
      <button aria-pressed={isFavorite} onClick={onToggleFavorite}>
        {isFavorite ? 'Quitar de favoritos' : 'Marcar como favorito'}
      </button>
    </article>
  )
}
```

src/testing/PokemonCard.test.tsx

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { expect, it, vi } from 'vitest'
import { PokemonCard } from './PokemonCard'
```

```
it('muestra el número con tres cifras y avisa al marcar el favorito', async () => {
  const user = userEvent.setup()
  const onToggleFavorite = vi.fn()
  render(<PokemonCard name="Brotín" number={1} isFavorite={false} onToggleFavorite=
{onToggleFavorite} />)

  const card = screen.getByRole('article', { name: 'Brotín' })
  expect(card).toHaveTextContent('#001')

  await user.click(screen.getByRole('button', { name: 'Marcar como favorito' }))
  expect(onToggleFavorite).toHaveBeenCalledTimes(1)
})
```

- **`it`** declara una prueba con una frase que dice qué comprueba.
- **`userEvent.setup()`** prepara un usuario simulado, que es la forma recomendada de usar **`user-event`**; **`user.click`** y **`user.type`** hacen clic y escriben como una persona.
- **`render`** monta el componente en el DOM simulado, y **`vi.fn()`** crea una función falsa que recuerda cómo la llamaron. **`screen`** es la pantalla entera, desde donde se hacen las consultas.
- **`screen.getByRole('button', { name: 'Marcar como favorito' })`** busca un botón por su **nombre accesible**, el que leería un lector de pantalla. Testing Library recomienda buscar primero por rol: si un elemento no se encuentra por su rol, puede que tu interfaz no sea accesible (Testing Library, *About Queries*).
- Las consultas **`getBy…`** buscan el elemento ahora y fallan si no está; las **`findBy…`** esperan a que aparezca, por eso devuelven una promesa y van con **`await`**. **`getByLabelText`** busca un campo por el texto de su etiqueta.
- **`expect(…)`** comprueba un resultado: si no se cumple, la prueba falla con un mensaje.

Una prueba de un formulario

El formulario de «Refs y el DOM», probado como lo usaría una persona:

src/testing/TrainerForm.test.tsx

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { expect, it, vi } from 'vitest'
import { TrainerForm } from '../refs/TrainerForm'

it('no guarda un nombre vacío, explica por qué y lleva el foco al campo', async () => {
  const user = userEvent.setup()
  const onSave = vi.fn()
  render(<TrainerForm onSave={onSave} />)

  await user.click(screen.getByRole('button', { name: 'Guardar' }))
```

```
  const name = screen.getByRole('textbox', { name: 'Nombre' })
  expect(name).toHaveAccessibleDescription('Escribe tu nombre')
  expect(name).toHaveFocus()
  expect(onSave).not.toHaveBeenCalled()

  await user.type(name, 'Ana')
  await user.click(screen.getByRole('button', { name: 'Guardar' }))
  expect(onSave).toHaveBeenCalledWith('Ana')
})
```

La prueba no sabe que hay una ref ni un **`useId`**: comprueba lo que se ve (el mensaje asociado al campo y el foco) y lo que pasa (que **`onSave`** no se llama hasta que el nombre es válido). Por eso sigue pasando si cambias cómo está hecho el formulario por dentro.

MSW: la red simulada

MSW (*Mock Service Worker*) intercepta los pedidos de red de tu código y responde lo que tú definas, sin cambiar el código de la app: axios hace su pedido como siempre, y MSW lo atiende. Los **handlers** dicen qué responder a cada dirección:

src/test/handlers.ts

```
import { http, HttpResponse } from 'msw'

const CORS = { 'Access-Control-Allow-Origin': '*' }
export const NAMES = ['brotin', 'fogarin', 'lagunix', 'chispon']

export const handlers = [
  http.get('https://pokeapi.co/api/v2/pokemon/:name', ({ params }) => {
    const key = String(params.name)
    const name = /^\d+$/.test(key) ? NAMES[Number(key) - 1] : key
    if (!name || !NAMES.includes(name)) return new HttpResponse(null, { status: 404,
headers: CORS })
    return HttpResponse.json(
      { id: NAMES.indexOf(name) + 1, name, types: [{ type: { name: 'grass' } }] },
      { headers: CORS },
    )
  }),
]
```

src/test/server.ts

```
import { setupServer } from 'msw/node'
import { handlers } from './handlers'

export const server = setupServer(...handlers)
```

Un **origen** es la combinación de protocolo, dominio y puerto de una dirección: **`https://pokeapi.co`** y la página de tu app son orígenes distintos. El navegador (y jsdom) solo deja que tu código lea la respuesta de otro origen si ese servidor lo autoriza con el encabezado **`Access-Control-Allow-Origin`** (un **encabezado** es un dato extra que viaja con el pedido o la respuesta): eso es **CORS** (*Cross-Origin Resource Sharing*). Por eso el handler de PokéAPI lo incluye, y los de **`/api`** que verás más abajo, que van al mismo origen de la página, no lo necesitan. **`/^\d+$/`** es una expresión regular que reconoce un texto hecho solo de dígitos: así el handler acepta un número o un nombre, como la PokéAPI real.

`setupServer` arma el servidor simulado para Node. En el **`setup.ts`**, **`onUnhandledFrame: 'error'`** hace fallar cualquier pedido que no tenga handler: así ninguna prueba llama a una API real por error.

Una prueba de integración: un formulario que agrega un miembro al equipo, y un servidor que lo rechaza con un **422**, el código HTTP de «entendí el pedido, pero los datos no son válidos»:

src/testing/teamApi.ts

```ts
import axios from 'axios'

export type ApiError = { error: { code: string; message: string } }

export async function addMember(nickname: string): Promise<void> {
  await axios.post('/api/team', { nickname })
}

export function errorMessage(error: unknown): string {
  if (axios.isAxiosError<ApiError>(error) && error.response?.data?.error) {
    return error.response.data.error.message
  }
  return 'Algo salió mal. Vuelve a intentarlo.'
}
```

src/testing/AddMember.tsx

```tsx
import { useState, type SubmitEvent } from 'react'
import { addMember, errorMessage } from './teamApi'

type Status = { kind: 'idle' } | { kind: 'saved' } | { kind: 'error'; message: string }

export function AddMember() {
  const [status, setStatus] = useState<Status>({ kind: 'idle' })

  async function handleSubmit(event: SubmitEvent<HTMLFormElement>) {
    event.preventDefault()
    try {
      await addMember(String(new FormData(event.currentTarget).get('nickname')))
      setStatus({ kind: 'saved' })
    } catch (error) {
      setStatus({ kind: 'error', message: errorMessage(error) })
```

```
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Apodo <input name="nickname" />
      </label>
      <button>Agregar al equipo</button>
      {status.kind === 'saved' && <p role="status">Agregado.</p>}
      {status.kind === 'error' && <p role="alert">{status.message}</p>}
    </form>
  )
}
```

src/testing/AddMember.test.tsx

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { http, HttpResponse } from 'msw'
import { expect, it } from 'vitest'
import { server } from '../test/server'
import { AddMember } from './AddMember'

it('muestra el mensaje del servidor cuando rechaza el apodo con un 422', async () => {
  server.use(
    http.post('/api/team', () =>
      HttpResponse.json(
        { error: { code: 'INVALID_NICKNAME', message: 'El apodo solo puede tener letras
y números.' } },
        { status: 422 },
      ),
    ),
  )
  const user = userEvent.setup()
  render(<AddMember />)

  await user.type(screen.getByLabelText('Apodo'), 'Brote!')
  await user.click(screen.getByRole('button', { name: 'Agregar al equipo' }))

  expect(await screen.findByRole('alert')).toHaveTextContent('El apodo solo puede tener
letras y números.')
})
```

`server.use` agrega un handler solo para esta prueba, y el **`resetHandlers`** del **`setup.ts`** lo quita al terminar. La prueba recorre el camino completo: el componente, **`addMember`**, axios, la red simulada, **`errorMessage`** y el mensaje en pantalla.

Pruebas de punta a punta con Playwright

Playwright abre un navegador real, levanta la app y la maneja con las mismas consultas por rol. Esta prueba de Dex 506 recorre la app de punta a punta: busca, abre una ficha, marca un favorito, arma un equipo y gana un gimnasio.

e2e/journey.spec.ts

```
test('de la Pokédex a la medalla: buscar, abrir la ficha, armar el equipo y ganar un gimnasio', async ({
  page,
}) => {
  await signIn(page)

  await page.getByRole('link', { name: 'Pokédex' }).click()
  await page.getByRole('searchbox').fill('lag')
  await expect(page.getByText('1 Pokémon')).toBeVisible()
  await expect(page).toHaveURL(/\?q=lag/)

  await page.getByRole('link', { name: 'lagunix' }).click()
  await expect(page.getByRole('heading', { name: 'lagunix', level: 1 })).toBeVisible()
  const favorite = page.getByRole('button', { name: 'Favorito', exact: true })
  await favorite.click()
  await expect(favorite).toHaveAttribute('aria-pressed', 'true')

  await addMember(page, 1, 'Brote', 10, 'hoja-afilada')
  await addMember(page, 4, 'Chispa', 10, 'chispazo')
  await addMember(page, 12, 'Escara', 10, 'picadura')

  await page.goto('/gyms/caribe')
  for (const name of ['Brote', 'Chispa', 'Escara'])
    await page.getByRole('checkbox', { name: new RegExp(name) }).check()
  await page.getByRole('button', { name: 'Retar al gimnasio' }).click()
  await expect(page.getByRole('heading', { name: '¡Victoria!' })).toBeVisible()

  await page.goto('/gyms')
  await expect(page.getByText('Ganaste la Medalla Arrecife')).toBeVisible()
})
```

`signIn` y **`addMember`** son funciones del mismo archivo que inician sesión y agregan un miembro. En la parte 4 las escribes, instalas Playwright y armas su configuración, que levanta la API y la web antes de las pruebas. Las seis pruebas e2e de Dex 506 pasan en Chromium, con la API levantada con criaturas inventadas (está hecha con **Express**, un framework de Node para servidores web, y la construyes en la parte 4) y la web en el servidor de desarrollo de Vite.

Errores frecuentes

> *"El error frecuente: una prueba con **`use`** y una promesa se queda para siempre en «Cargando...», aunque la promesa termine. En un navegador el mismo componente funciona. Envuelve en **`await act(async () => …)`** la acción que hace suspender al componente; **`act`** es la función de React que agrupa las actualizaciones de una prueba y las procesa antes de seguir:"*

src/react19/PickPokemon.test.tsx

```
import { act, render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { expect, it } from 'vitest'
import { PickPokemon } from './PickPokemon'

it('use lee la promesa y Suspense muestra la espera', async () => {
  const user = userEvent.setup()
  render(<PickPokemon />)
  await act(async () => {
    await user.click(screen.getByRole('button', { name: 'Ver el Pokémon número 3' }))
    expect(screen.getByText('Cargando…')).toBeInTheDocument()
  })
  expect(await screen.findByRole('heading', { name: '#3
lagunix' })).toBeInTheDocument()
})
```

> *"El error frecuente: un handler de MSW que responde algo distinto en cada pedido, cuando la API real responde siempre lo mismo. La prueba pasa por el handler, no por tu código, y esconde errores que con la API real sí aparecen. Que los handlers respondan como la API de verdad."*

> *"El error frecuente: con Playwright, **`getByRole('button', { name: 'Escara' })`** también encuentra el botón «Agregar a escarabin» de Dex 506: por defecto, **`name`** compara por una parte del texto y sin distinguir mayúsculas. Usa **`exact: true`** cuando el nombre de un elemento contiene el de otro."*

Arquitectura, calidad y seguridad

En esta sección vas a ver lo que separa un ejercicio de una aplicación profesional, con Dex 506 como ejemplo.

Organizar por funcionalidad

Las carpetas de la web de Dex 506:

```
apps/web/src
apps/web/src/app
apps/web/src/app/store
apps/web/src/features
apps/web/src/features/auth
```

```
apps/web/src/features/gyms
apps/web/src/features/pokedex
apps/web/src/features/pokemon
apps/web/src/features/team
apps/web/src/i18n
apps/web/src/shared
apps/web/src/shared/api
apps/web/src/shared/hooks
apps/web/src/shared/theme
apps/web/src/shared/ui
apps/web/src/test
```

- **`app`** arma la aplicación: las rutas, los proveedores y el store de Redux.
- **`features`** tiene una carpeta por **funcionalidad**: el login, la Pokédex, la ficha, el equipo y los gimnasios. Cada una guarda sus componentes, sus hooks, sus queries y sus pruebas juntos.
- **`shared`** tiene lo que usan varias funcionalidades: el cliente de axios, los hooks genéricos, el tema y los componentes de interfaz.
- **`i18n`** tiene los textos de la interfaz en cada idioma.

Organizar por funcionalidad, y no por tipo de archivo (todos los componentes en una carpeta, todos los hooks en otra), hace que un cambio en el equipo toque una carpeta y no cinco.

Capas y contratos

Dentro de cada funcionalidad, el código se reparte en tres capas:

- la **interfaz**: los componentes, que solo muestran y avisan;
- el **estado**: Context, Zustand o Redux, según la tabla «No todo estado es igual» de la parte 2;
- los **datos**: las queries de TanStack Query y el cliente de axios.

La API de Express y la web comparten un paquete, **`packages/contracts`**, con los esquemas de Zod de cada dato. Un **contrato** es esa definición compartida de qué se envía y qué se recibe: si la API cambia la forma de un dato, la web deja de compilar en vez de fallar en producción. Y como entre dos despliegues la API y la web pueden quedar en versiones distintas, la web además valida con **`parse`** cada respuesta que recibe. Así es el contrato de un error:

packages/contracts/src/schemas.ts

```ts
export const ApiErrorSchema = z.object({
  error: z.object({
    code: z.string(),
    message: z.string(),
    /** One message per form field, when it is a validation error. */
    fields: z.record(z.string(), z.string()).optional(),
  }),
```

```
})
export type ApiError = z.infer<typeof ApiErrorSchema>
```

Todos los errores de la API tienen esa forma, y por eso la web los muestra siempre igual: el **`errorMessage`** de «Pruebas» lee justamente **`error.message`**.

> **– Para profundizar**
>
> Dex 506 usa cuatro mecanismos de estado (TanStack Query, Context, Zustand y Redux Toolkit) para que conozcas los cuatro. En una app de este tamaño, Redux sobra: Zustand alcanza para la alineación de los gimnasios. Redux se justifica en una app grande, con muchas personas trabajando sobre el mismo estado, donde que cada cambio pase por una acción con nombre y quede registrado vale su costo.

Accesibilidad

Una app accesible se puede usar con teclado, con un lector de pantalla y con poca visión. Tres reglas cubren la mayoría de los casos:

- **Semántica:** usar el elemento que corresponde: **`<button>`** para acciones, **`<a>`** para navegar, **`<label>`** para cada campo. Un **`<div>`** con **`onClick`** no se puede enfocar con el teclado, y un lector de pantalla no lo anuncia como botón.
- **Teclado:** todo lo que se hace con el mouse se tiene que poder hacer con el teclado, y el foco tiene que verse.
- **Contraste:** el texto común necesita un contraste de al menos 4.5:1 con su fondo (WCAG, las pautas de accesibilidad web del W3C, nivel AA).

Las pruebas por rol ayudan con la primera, aunque no la garantizan: un **`<div role="button">`** también aparece como botón. Además, Dex 506 pasa una auditoría automática de accesibilidad en sus dos temas; esa auditoría encontró un texto coral con contraste de 4.21:1 en el tema oscuro, que se corrigió.

Seguridad: XSS

Un ataque de **XSS** (*cross-site scripting*) consigue que tu página ejecute un script ajeno, por ejemplo escondido en la biografía de un usuario. React te protege por defecto: lo que pones entre llaves se muestra como texto, nunca como HTML. Dentro de JSX, la forma más directa de saltarse esa protección tiene un nombre que avisa (fuera de JSX, escribir **`innerHTML`** a mano con una ref tiene el mismo riesgo):

ANTES

src/architecture/Bio.tsx

DESPUÉS

src/architecture/Bio.tsx

```
export function UnsafeBio({ bio }: { bio:
string }) {
  return <p dangerouslySetInnerHTML={{
__html: bio }} />
}
```

```
export function Bio({ bio }: { bio:
string }) {
  return <p>{bio}</p>
}
```

Con una biografía que contiene **<img src="x" onerror="alert(1)">**, el «antes» crea de verdad ese **<img>**, y en un navegador el **onerror** se ejecutaría. El «después» muestra el texto tal cual. Las pruebas del ejemplo comprueban los dos casos. Si alguna vez necesitas mostrar HTML de un usuario, límpialo antes con una biblioteca hecha para eso, como DOMPurify.

Seguridad: dónde vive la sesión

Un **token** es un texto secreto que identifica la sesión. Una forma común de manejarla es guardar el token en **sessionStorage**, que es como el **localStorage** de la parte 2 pero se borra al cerrar la pestaña, y mandarlo en cada pedido en el encabezado **Authorization: Bearer …**, la forma estándar de enviarlo:

ANTES

src/architecture/session.ts

```
export function saveSession(token:
string) {
  sessionStorage.setItem('token', token)
}

export function authHeaders() {
  return { Authorization: `Bearer
${sessionStorage.getItem('token')}` }
}
```

DESPUÉS

apps/api/src/routes/auth.ts

```
router.post('/login', async (req, res) =>
{
  const { email, password } =
LoginInputSchema.parse(req.body)
  const user =
store.read().users.find((candidate) =>
candidate.email === email.toLowerCase())
  // The hash is checked even for an
unknown email: the response time does not
reveal which
  // accounts exist.
  const valid = await
verifyPassword(password,
user?.passwordHash ?? FAKE_HASH)
  if (!user || !valid) {
    throw new HttpError(401,
'INVALID_CREDENTIALS', 'Correo o
contraseña incorrectos')
  }
  const token =
randomBytes(32).toString('hex')
  await store.update((data) => {
    // Expired sessions are cleaned up on
each login, so the file does not grow
forever.
```

```
    for (const [old, session] of
Object.entries(data.sessions)) {
      if (session.expiresAt <=
Date.now()) delete data.sessions[old]
    }
    data.sessions[token] = { userId:
user.id, expiresAt: Date.now() +
SESSION_MS }
  })
  // httpOnly: the page's JavaScript
cannot read it, so an injected script
cannot steal it.
  res.cookie(SESSION_COOKIE, token, {
    httpOnly: true,
    sameSite: 'lax',
    secure: options.secureCookies,
    path: '/',
    maxAge: SESSION_MS,
  })
  res.json({ id: user.id, email:
user.email, trainerName: user.trainerName
} satisfies User)
})
```

El problema del «antes»: cualquier script que corra en tu página puede leer **`sessionStorage`**. Un solo XSS, y el atacante se lleva la sesión. El «después» es código de la API de Express de Dex 506, que escribes en la parte 4; lo que importa es **`res.cookie(…)`**: la API guarda la sesión en una **cookie `httpOnly`**, que el navegador manda sola en cada pedido y que el JavaScript de la página no puede leer. La web no guarda nada. Un XSS todavía podría hacer pedidos en nombre del usuario mientras la página está abierta, pero ya no puede llevarse la sesión.

- **`sameSite: 'lax'`** hace que el navegador no mande la cookie en los pedidos que salen de otro sitio, salvo cuando alguien navega a tu sitio con un **`GET`**, por ejemplo desde un enlace: un formulario **`POST`**, un **`fetch`** o una imagen desde otro sitio salen sin ella. Eso frena el **CSRF** (*cross-site request forgery*), el ataque en que otra página hace pedidos a tu API con la sesión de quien la visita, siempre que un pedido **`GET`** nunca cambie datos.
- **`secure`** hace que la cookie solo viaje por HTTPS.
- Para que la cookie viaje sin configurar CORS, la web y la API se sirven desde el mismo origen: en desarrollo, el proxy de Vite, un intermediario que reenvía los pedidos, manda **`/api`** a Express, como se ve en la configuración de «Rendimiento».

Variables de entorno

Vite expone al código del navegador las variables de entorno que empiezan con **`VITE_`**:

src/architecture/config.ts

```
export const config = {
  mapsKey: import.meta.env.VITE_MAPS_KEY ?? '',
}
```

`import.meta.env` es el objeto donde Vite pone esas variables. Para que TypeScript conozca la variable, se declara en **`src/vite-env.d.ts`**, un archivo que creas en **`src`**:

src/vite-env.d.ts

```
/// <reference types="vite/client" />

interface ImportMetaEnv {
  readonly VITE_MAPS_KEY?: string
}
```

El valor sale de un archivo **`.env`** en la raíz del proyecto (una línea **`VITE_MAPS_KEY=…`**) o de la línea de comandos. «Exponer» significa copiar ese valor en el JavaScript que descarga cualquier visitante, siempre que el código la use: en el ejemplo, **`App.tsx`** lee **`config.mapsKey`** para decidir si muestra el mapa (si nadie la usara, Vite la dejaría fuera del build). Para comprobarlo, arma el build con la variable y busca el texto en los archivos generados:

```
VITE_MAPS_KEY=clave-de-ejemplo npx vite build
grep -rho clave-de-ejemplo dist/assets
```

La salida:

```
clave-de-ejemplo
```

Ahí está. Una variable **`VITE_`** no es secreta: una clave de API privada va en el servidor, nunca en una variable **`VITE_`** (Vite, *Env Variables and Modes*).

Internacionalización

La **internacionalización** (**i18n**) es preparar la app para varios idiomas. Una forma simple, la de Dex 506, es un diccionario por idioma, con funciones para los textos que llevan datos, y un Context que da el diccionario del idioma activo:

src/architecture/i18n.tsx

```
import { createContext, use, useMemo, useState, type ReactNode } from 'react'

const dictionaries = {
  es: { greeting: (name: string) => `Hola, ${name}`, signOut: 'Cerrar sesión' },
```

```
  en: { greeting: (name: string) => `Hi, ${name}`, signOut: 'Sign out' },
}

type Language = keyof typeof dictionaries
type I18n = { t: (typeof dictionaries)['es']; language: Language; setLanguage:
(language: Language) => void }

const I18nContext = createContext<I18n | null>(null)

export function I18nProvider({ children }: { children: ReactNode }) {
  const [language, setLanguage] = useState<Language>('es')
  const value = useMemo(() => ({ t: dictionaries[language], language, setLanguage }),
[language])
  return <I18nContext value={value}>{children}</I18nContext>
}

export function useI18n(): I18n {
  const i18n = use(I18nContext)
  if (!i18n) throw new Error('useI18n tiene que usarse dentro de I18nProvider')
  return i18n
}
```

src/architecture/Greeting.tsx

```
import { useI18n } from './i18n'

export function Greeting({ trainerName }: { trainerName: string }) {
  const { t, language, setLanguage } = useI18n()
  return (
    <>
      <p>{t.greeting(trainerName)}</p>
      <button onClick={() => setLanguage(language === 'es' ? 'en' : 'es')}>{language ===
'es' ? 'English' : 'Español'}</button>
      <button>{t.signOut}</button>
    </>
  )
}
```

`keyof typeof dictionaries` es la unión de las claves del objeto, **`'es' | 'en'`**. El **`value`** del proveedor va con **`useMemo`**: así es el mismo objeto mientras no cambie el idioma, y no hace renderizar a quienes leen el Context. Para usarlo, **`Greeting`** tiene que estar dentro de **`<I18nProvider>`**. Los componentes no tienen textos escritos: leen **`t.greeting(name)`** o **`t.signOut`** del idioma activo. La excepción son los nombres de los idiomas, que van siempre en su propio idioma («English», «Español»). Dex 506 hace lo mismo con un diccionario por idioma en **`apps/web/src/i18n`**. Para una app con muchos idiomas, plurales y formatos de fechas y números, hay bibliotecas como react-i18next o FormatJS.

Panorama y camino a senior

En esta sección vas a ver dónde encaja una SPA como Dex 506 dentro del ecosistema de React.

Renderizar en el servidor

Dex 506 es una SPA: el servidor manda un HTML casi vacío, y el navegador descarga el JavaScript y arma la pantalla. Hay otras formas:

- **SSR** (*server-side rendering*): el servidor ejecuta los componentes y manda el HTML ya armado. La página se ve antes y los buscadores la leen completa; después, React la vuelve interactiva en el navegador.
- Los **Server Components** son componentes que se renderizan antes de llegar al navegador, en el build o en el servidor en cada pedido, y cuyo código nunca se descarga. No pueden usar estado ni efectos; para la interactividad se combinan con componentes del cliente, marcados con `'use client'` (React, *Server Components*).
- Las **Server Functions** son funciones asíncronas que se ejecutan en el servidor y que un componente del cliente llama como si fueran locales, marcadas con `'use server'`. Hasta septiembre de 2024 se llamaban todas Server Actions; hoy ese nombre queda para las que se usan como Action (React, *Server Functions*).

Todo eso necesita un **framework**: una herramienta que, además de React, se encarga del servidor, las rutas y el build. La documentación de React recomienda empezar una app nueva con uno, como Next.js o React Router en su modo framework (React, *Creating a React App*). React Router tiene tres modos: el declarativo, el de datos que usaste en la parte 2 y el de framework, que suma SSR, rutas con tipos y code splitting automático (React Router, *Picking a Mode*). Este mismo sitio está hecho con Next.js.

> **Cuidado:** un Server Component o una Server Function es código que corre en tu servidor, con sus propios riesgos. En diciembre de 2025, React publicó un aviso sobre una vulnerabilidad crítica en los Server Components (React, *Critical Security Vulnerability in React Server Components*). Un framework trae responsabilidades de servidor que una SPA no tiene.

Cuándo una SPA y cuándo un framework

Una SPA como Dex 506 conviene cuando la app vive detrás de un login, los buscadores no necesitan leer su contenido y quieres servirla como archivos estáticos, sin servidor de renderizado. Un framework conviene cuando la primera carga y el posicionamiento en buscadores importan, como en una tienda o un blog, o cuando quieres que parte del código nunca llegue al navegador.

Qué estudiar después

- **Un framework**: Next.js o React Router en modo framework, para ver SSR y Server Components en la práctica.
- **Accesibilidad a fondo**: las pautas WCAG y probar tu app con un lector de pantalla.

- **Rendimiento web**: las métricas de carga del navegador y cómo medirlas en producción.
- **React Native con Expo**, si quieres llevar lo que sabes a apps móviles.

Lo que aprendiste

React renderiza en dos fases, render y commit, y un componente renderiza cuando cambia su estado, cuando renderiza su padre o cuando cambia un Context que lee. El estado vive en la posición del árbol, y la **`key`** lo reinicia; React compara objetos y funciones por identidad. Un hook propio bien diseñado hace una sola cosa y recibe lo que necesita. Las refs guardan valores sin renderizar y llegan al DOM. Los genéricos y las uniones discriminadas hacen que los estados imposibles no compilen. Para el rendimiento, primero se mide: **`memo`** evita renders, **`useMemo`** y **`useCallback`** conservan la identidad que **`memo`** necesita, y React Compiler los aplica solo. React 19 trae **`use`**, **`useOptimistic`**, **`<ViewTransition>`**, **`<Activity>`** y los metadatos en componentes. Y una app profesional se prueba en tres niveles, se organiza por funcionalidad y cuida la accesibilidad y la seguridad desde el principio.

Qué sigue

En la parte 4: construye una app React + TypeScript de cero a build de producción juntas todo: construyes Dex 506 desde una carpeta vacía hasta el build de producción, con su API de Node, paso a paso y con el código de cada paso para descargar.

Bibliografía

- React, Profiler. Consultado el 18 de noviembre de 2026.
- React, Component, sobre los límites de error. Consultado el 18 de noviembre de 2026.
- The React Team, React v19, 5 de diciembre de 2024. Consultado el 18 de noviembre de 2026.
- The React Team, React 19.2, 1 de octubre de 2025. Consultado el 18 de noviembre de 2026.
- The React Team, React 19.3, 9 de septiembre de 2026. Consultado el 18 de noviembre de 2026.
- Lauren Tan, Joe Savona y Mofei Zhang, React Compiler v1.0, 7 de octubre de 2025. Consultado el 18 de noviembre de 2026.
- React, useOptimistic, ViewTransition, Activity, title y meta. Consultados el 18 de noviembre de 2026.
- React, Server Components, Server Functions y Creating a React App. Consultados el 18 de noviembre de 2026.
- React, Critical Security Vulnerability in React Server Components, 3 de diciembre de 2025. Consultado el 18 de noviembre de 2026.
- React Router, Picking a Mode. Consultado el 18 de noviembre de 2026.
- Testing Library, About Queries. Consultado el 18 de noviembre de 2026.

- Vite, Env Variables and Modes. Consultado el 18 de noviembre de 2026.