React intermedio: datos, rutas, formularios y estado global

Creado por Manuel Herrera Mesén · 26 de noviembre de 2026

Parte 2 de 4 · react · typescript · tanstack-query · react-router · frontend · serie-react

EN ESTE ARTÍCULO

- Del render a los efectos
- useEffect sin miedo
- Cuándo NO necesitas un efecto
- Hablar con el servidor
- TanStack Query
- Navegación con React Router
- Context
- Formularios
- Estado global
- Lo que aprendiste
- Qué sigue
- Bibliografía

> Parte 2 de 4 de la serie «Aprende React construyendo una app real». Viene de la parte 1: React desde cero.

> **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 1 vive en un arreglo escrito a mano. Una aplicación real no: los datos están en un servidor, tardan en llegar, a veces no llegan, y la persona que usa la app espera poder compartir un enlace a la ficha de un Pokémon, iniciar sesión y que su preferencia de vista siga ahí mañana.

En esta parte llevamos la mini Pokédex a ese mundo. Lo que hace React no cambia; cambia todo lo que rodea al componente, y cada herramienta de esta parte existe porque resolver eso a mano se volvió difícil.

Cómo leer esta guía. Igual que en la parte 1: el texto principal va paso a paso, y los recuadros **Para profundizar** tienen el detalle fino. Cuando un concepto viene de la parte 1, el texto enlaza la sección donde se explicó.

Todo el código lo ejecuté con React 19.3.1, TypeScript 7.0.2, axios 1.21.1, TanStack Query 5.108.0, React Router 8.5.1, React Hook Form 7.92.0, `@hookform/resolvers` 7.0.7, Zod 4.6.7, Zustand 5.1.0, Redux Toolkit 2.14.0 y react-redux 9.3.1: cada ejemplo tiene pruebas automáticas que pasan, y los mensajes que ves son los que dio la herramienta.

Los ejemplos siguen en el proyecto de Vite que creaste en la parte 1. Antes de empezar, instala las bibliotecas de esta parte; cada una se explica en su sección:

```
npm install axios @tanstack/react-query react-router react-hook-form @hookform/resolvers
zod zustand @reduxjs/toolkit react-redux
```

Para ver cada ejemplo, haz como en la parte 1: impórtalo en **`App.tsx`** y ponlo en su **`return`**. En «Navegación con React Router», **`App.tsx`** pasa a ser el router, y ahí te digo dónde seguir probando.

Al terminar esta parte podrás:

1. usar efectos correctamente y reconocer cuándo sobran;
2. pedir datos a una API con axios y con TanStack Query;
3. navegar entre pantallas con React Router;
4. compartir datos entre componentes con Context;
5. construir formularios validados y accesibles;
6. elegir dónde vive cada dato: en el componente, en la URL, en el servidor, en el formulario o en un estado global.

Del render a los efectos

En esta sección vas a ver por qué pedir datos durante el render no funciona.

El render tiene que ser puro

En la parte 1 viste que renderizar es llamar al componente para saber qué mostrar, y que React puede hacerlo muchas veces. Por eso el render tiene que ser una **función pura**: con las mismas props y el mismo estado, devuelve lo mismo, y no cambia nada fuera de sí misma.

Un **efecto secundario** es justamente eso que el render no debe hacer: cambiar algo fuera del componente. Pedir datos a un servidor, cambiar el título de la pestaña o escuchar el tamaño de la ventana son efectos secundarios.

El problema: pedir datos en el render

Este componente pide un Pokémon a PokéAPI directamente en el cuerpo de la función:

src/effects/FetchInRender.tsx

```
import { useState } from 'react'

export function FetchInRender() {
  const [pokemon, setPokemon] = useState<{ name: string } | null>(null)
```

```
  fetch('https://pokeapi.co/api/v2/pokemon/1')
    .then((response) => response.json())
    .then((data: { name: string }) => setPokemon(data))
  return <h2>{pokemon?.name}</h2>
}
```

`useState<{ name: string } | null>(null)` dice qué forma tiene el estado: un objeto con un **`name`**, o **`null`** mientras no llegó nada. Esos **`< >`** son la sintaxis de los genéricos de TypeScript, que vas a ver en la parte 3; por ahora, léelos como «este estado guarda esto». Y **`pokemon?.name`** usa el **encadenamiento opcional** (**`?.`**): si **`pokemon`** es **`null`**, da **`undefined`** en lugar de fallar.

Cada render dispara un pedido. Cuando llega la respuesta, **`setPokemon`** guarda el objeto que devolvió **`response.json()`**, que es un objeto nuevo cada vez, aunque traiga el mismo nombre. React solo se salta el render si el valor nuevo es igual al anterior, y dos objetos distintos nunca lo son: hay otro render, que dispara otro pedido, y así sin parar. En la prueba automática con la que lo comprobé, el servidor responde siempre lo mismo, y aun así hay más de tres pedidos en 300 milisegundos; seguirían si la prueba no lo cortara. Hace falta un lugar para el código que toca el mundo exterior, fuera del render: los efectos.

useEffect sin miedo

En esta sección vas a sincronizar un componente con algo externo a React.

Qué es

`useEffect` es el hook que ejecuta código después de que React aplicó el render en pantalla. Técnicamente, recibe una función (el efecto) y una lista de **dependencias**: los valores que, al cambiar, hacen que el efecto vuelva a correr. Sirve para sincronizar el componente con algo que React no controla: el título de la pestaña, un listener del navegador (una función que el navegador llama cada vez que pasa un evento, como un cambio de tamaño de la ventana), un pedido a un servidor.

src/effects/DocumentTitle.tsx

```
import { useEffect } from 'react'

export function DocumentTitle({ pokemonName }: { pokemonName: string }) {
  useEffect(() => {
    document.title = `${pokemonName} · Mini Pokédex`
  }, [pokemonName])

  return <h2>{pokemonName}</h2>
}
```

Cuando **`pokemonName`** cambia, React renderiza, aplica el cambio en pantalla y después corre el efecto, que actualiza el título. Si **`pokemonName`** no cambia, el efecto no vuelve a correr.

La limpieza

Algunos efectos dejan algo en marcha. Este escucha el tamaño de la ventana:

src/effects/WindowWidth.tsx

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

export function WindowWidth() {
  const [width, setWidth] = useState(window.innerWidth)

  useEffect(() => {
    function handleResize() {
      setWidth(window.innerWidth)
    }
    window.addEventListener('resize', handleResize)
    return () => window.removeEventListener('resize', handleResize)
  }, [])

  return <p>Ancho de la ventana: {width}px</p>
}
```

La función que devuelve el efecto es la **limpieza**: React la llama antes de volver a correr el efecto y cuando el componente desaparece de la pantalla. Que el componente aparezca se llama **montaje**, y que desaparezca, **desmontaje**. Sin la limpieza, cada vez que el componente se monta quedaría un listener más escuchando, aunque el componente ya no exista. La lista de dependencias vacía (`[]`) dice que el efecto corre solo al montar.

> **– Para profundizar**
>
> en desarrollo, con el `StrictMode` de la plantilla, React monta cada componente y corre el efecto, la limpieza y otra vez el efecto: una limpieza y un efecto de más (React, *StrictMode*). No es un error: está ahí para que una limpieza que falta se note enseguida. En el build de producción no pasa.

La carrera de pedidos

Pedir datos en un efecto tiene una trampa. Si cambias rápido de un Pokémon a otro, las respuestas pueden llegar en otro orden: la del primero, más lenta, llega después y pisa a la del segundo. Para evitarlo, la limpieza cancela el pedido viejo con un **`AbortController`**, un objeto del navegador que permite cancelar un pedido en curso:

src/effects/PokemonDetails.tsx

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

```tsx
import { fetchPokemon, type PokemonDetail } from '../api/pokeapi'

export function PokemonDetails({ name }: { name: string }) {
  const [pokemon, setPokemon] = useState<PokemonDetail | null>(null)

  useEffect(() => {
    const controller = new AbortController()
    fetchPokemon(name, controller.signal)
      .then(setPokemon)
      .catch((error: unknown) => {
        if (!axios.isCancel(error)) console.error(error)
      })
    return () => controller.abort()
  }, [name])

  if (!pokemon || pokemon.name !== name) return <p>Cargando…</p>
  return <h2>{`#${pokemon.id} ${pokemon.name}`}</h2>
}
```

`fetchPokemon` es una función que pide un Pokémon a PokéAPI; la vas a escribir en «Hablar con el servidor». Recibe la señal del **`AbortController`** (**`controller.signal`**, el objeto que le avisa al pedido que se canceló) y, si cancelas, el pedido se corta. Cada vez que **`name`** cambia, la limpieza cancela el pedido anterior y el efecto lanza uno nuevo. Un pedido cancelado termina con un error a propósito, y por eso el **`catch`** lo ignora con **`axios.isCancel`**. El tipo **`unknown`** de **`error: unknown`** es el de un valor del que TypeScript no sabe nada: hay que comprobar qué es antes de usarlo, como hace **`axios.isCancel`**.

Mientras el Pokémon que llegó no sea el pedido, el componente muestra «Cargando…». Si el pedido falla, se queda ahí y el error solo sale en la consola: en «Hablar con el servidor» vas a mostrarlo en pantalla.

useEffectEvent

A veces un efecto necesita leer un valor sin depender de él. Este registra cada visita a un Pokémon con el tema actual, pero cambiar el tema no es una visita nueva:

src/effects/VisitLogger.tsx

```tsx
import { useEffect, useEffectEvent } from 'react'

export function VisitLogger({ pokemonName, theme }: { pokemonName: string; theme: string
}) {
  const logVisit = useEffectEvent(() => {
    console.info(`Visitaste ${pokemonName} con el tema ${theme}`)
  })

  useEffect(() => {
    logVisit()
  }, [pokemonName])
```

```
  return <p>{pokemonName}</p>
}
```

useEffectEvent separa la parte del efecto que es un evento: **logVisit** siempre ve los valores más recientes de **pokemonName** y **theme**, pero no es una dependencia, así que cambiar el tema no vuelve a correr el efecto. Llegó con React 19.2 (React, *React 19.2*).

Cuándo NO necesitas un efecto

En esta sección vas a reconocer el error más común de quien empieza con efectos.

El problema: calcular con un efecto

El total de niveles de un equipo se puede calcular a partir de sus miembros. Este componente lo guarda en un estado y lo actualiza con un efecto:

ANTES

src/no-effect/TeamLevelWithEffect.tsx

```tsx
import { useEffect, useState } from
'react'
import type { Member } from './TeamLevel'

export function TeamLevelWithEffect({
members }: { members: Member[] }) {
  const [total, setTotal] = useState(0)
  useEffect(() => {
    setTotal(members.reduce((sum, member)
=> sum + member.level, 0))
  }, [members])
  return <p>Nivel total: {total}</p>
}
```

DESPUÉS

src/no-effect/TeamLevel.tsx

```tsx
export type Member = { nickname: string;
level: number }

export function TeamLevel({ members }: {
members: Member[] }) {
  const total = members.reduce((sum,
member) => sum + member.level, 0)
  return <p>Nivel total: {total}</p>
}
```

El primero hace dos renders: uno con el total en **0** y otro con el valor correcto, después de que el efecto lo calcula. Lo comprobé con una prueba automática y un **Profiler**, un componente de React que avisa cada vez que su contenido se renderiza: la primera versión registra dos renders, y la segunda, uno. El linter de la plantilla ya lo marca:

```
⚠ react(set-state-in-effect): Calling setState synchronously within an effect can
trigger cascading renders
   ╭─[src/no-effect/TeamLevelWithEffect.tsx:7:5]
 5 │   const [total, setTotal] = useState(0)
 6 │   useEffect(() => {
   ·   ──────┬──────
```

```
   ·       └── This is the containing effect
 7 |     setTotal(members.reduce((sum, member) => sum + member.level, 0))
   ·     ──────┬──────
   ·           └── Avoid calling setState() directly within an effect
 8 |   }, [members])
   └────

  help: Effects should synchronize React with external systems. Calling setState
synchronously inside an effect starts another render and is usually unnecessary. Derive
the value during render, initialize state directly, or update it from the event that
caused the change. Use an effect only when synchronizing with an external system.
  note: React Compiler skipped optimizing this component or hook. Additional guidance:
https://react.dev/reference/eslint-plugin-react-hooks/lints/set-state-in-effect

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

Un **dato derivado** es uno que se calcula a partir de otros. Se calcula en el render, como en el segundo bloque: es la misma idea que el **`visible`** del buscador de la parte 1. Un efecto es para sincronizarse con algo externo a React, no para transformar datos que ya tienes.

> **Cuidado:** lo que pasa por una acción del usuario tampoco va en un efecto. Si al marcar un favorito hay que guardarlo en el servidor, eso va en el manejador del clic, que es donde sabes que pasó.

Hablar con el servidor

En esta sección vas a pedir datos a una API real y a mostrar sus estados de carga, error y vacío.

Lo justo de HTTP

Una aplicación habla con un servidor por **HTTP**: le manda un pedido y recibe una respuesta. El pedido usa un **método** que dice qué quiere: **`GET`** para leer, **`POST`** para crear, **`PUT`** para reemplazar, **`DELETE`** para borrar. La respuesta trae un **código de estado**: **`200`** es que salió bien, **`404`** que no existe lo que pediste, **`500`** que el servidor falló. Los datos suelen viajar en **JSON**, un formato de texto con una forma parecida a la de los objetos de JavaScript, aunque sin funciones ni fechas.

axios

El navegador trae **`fetch`** para hacer pedidos. Esta serie usa **axios**, una biblioteca que agrega lo que **`fetch`** deja a mano: convierte el JSON sola, trata un **`404`** o un **`500`** como error y permite configurar todos los pedidos en un solo lugar. Ese lugar es una **instancia**:

src/api/pokeapi.ts

```ts
import axios from 'axios'
```

```
export const pokeApi = axios.create({
  baseURL: 'https://pokeapi.co/api/v2',
  timeout: 10_000,
})

pokeApi.interceptors.response.use(
  (response) => response,
  (error: unknown) => {
    if (!axios.isCancel(error)) console.error('PokéAPI no respondió bien:', error)
    return Promise.reject(error)
  },
)
```

baseURL es el comienzo de todas las direcciones; **`timeout: 10_000`** corta un pedido que tarda más de 10 segundos (el _ solo separa los miles, para leer mejor el número); y un **interceptor** es una función que pasa por cada respuesta antes de que llegue a tu código: aquí, registra los errores en la consola, salvo las cancelaciones de «useEffect sin miedo», que no son fallas de PokéAPI. Con la instancia, los pedidos de esta parte quedan así, en el mismo archivo **`pokeapi.ts`**, debajo de la instancia:

src/api/pokeapi.ts

```
export type PokemonPage = { count: number; names: string[] }
export type PokemonDetail = { id: number; name: string; types: string[] }

export const PAGE_SIZE = 10

export async function fetchPokemonPage(page: number): Promise<PokemonPage> {
  const { data } = await pokeApi.get<{ count: number; results: { name: string }[] }>
('/pokemon', {
    params: { limit: PAGE_SIZE, offset: (page - 1) * PAGE_SIZE },
  })
  return { count: data.count, names: data.results.map((result) => result.name) }
}

export async function fetchPokemon(name: string, signal?: AbortSignal):
Promise<PokemonDetail> {
  const { data } = await pokeApi.get<{
    id: number
    name: string
    types: { type: { name: string } }[]
  }>(`/pokemon/${name}`, { signal })
  return { id: data.id, name: data.name, types: data.types.map((entry) =>
entry.type.name) }
}
```

Los dos tipos de arriba, **`PokemonPage`** y **`PokemonDetail`**, son lo que devuelve cada función. En **`pokeApi.get<…>`**, los genéricos dicen qué forma suponemos que tiene la respuesta de PokéAPI: TypeScript

no lo comprueba cuando el programa corre. Cada función toma esa respuesta y la reduce a lo que usa la mini Pokédex.

Un hook propio

Pedir la lista, guardar el resultado y saber si está cargando o falló es lógica que vas a repetir. Un **hook propio** es una función tuya que agrupa esa lógica usando otros hooks. Técnicamente, es una función cuyo nombre empieza con **use** y que llama a hooks de React; sigue las mismas reglas que ellos.

src/fetching/usePokemonList.ts

```
import { useEffect, useState } from 'react'
import { fetchPokemonPage, type PokemonPage } from '../api/pokeapi'

type Outcome = { page: number; data: PokemonPage | null; error: string | null }

export function usePokemonList(page: number) {
  const [outcome, setOutcome] = useState<Outcome | null>(null)

  useEffect(() => {
    let ignore = false
    fetchPokemonPage(page)
      .then((data) => {
        if (!ignore) setOutcome({ page, data, error: null })
      })
      .catch(() => {
        if (!ignore) setOutcome({ page, data: null, error: 'No se pudo cargar la lista.'
})
      })
    return () => {
      ignore = true
    }
  }, [page])

  const isLoading = outcome === null || outcome.page !== page
  return {
    isLoading,
    data: isLoading ? null : outcome.data,
    error: isLoading ? null : outcome.error,
  }
}
```

ignore cumple aquí el papel del **AbortController** de antes, sin cancelar el pedido: cuando **page** cambia, la limpieza lo pone en **true**, y la respuesta vieja, si llega tarde, ya no se guarda.

El hook guarda para qué página llegó la respuesta. Si la página pedida no coincide con la de la respuesta guardada, todavía está cargando: así no hace falta poner el estado en «cargando» dentro del efecto, que

es lo que el linter marcaba en «Cuándo NO necesitas un efecto». El componente usa el hook y muestra los cuatro estados de una lista que viene de un servidor: cargando, error, vacía y con datos.

src/fetching/ManualList.tsx

```tsx
import { usePokemonList } from './usePokemonList'

export function ManualList() {
  const list = usePokemonList(1)

  if (list.isLoading) return <p>Cargando…</p>
  if (list.error) return <p role="alert">{list.error}</p>
  if (!list.data || list.data.names.length === 0) return <p>No hay Pokémon.</p>
  return (
    <ul>
      {list.data.names.map((name) => (
        <li key={name}>{name}</li>
      ))}
    </ul>
  )
}
```

`role="alert"` hace que un lector de pantalla anuncie el mensaje apenas aparece.

Funciona. Pero mira lo que falta: si dos componentes piden la misma página, se piden dos veces; si vuelves a una página, se pide de nuevo; si el servidor cambió, nadie se entera. Resolver eso a mano es mucho código, y es lo que hace la herramienta de la próxima sección.

TanStack Query

En esta sección vas a manejar los datos del servidor con TanStack Query.

El problema: el estado del servidor no es estado de la interfaz

Lo que guardas con **`useState`** es tuyo: solo cambia cuando tu código lo cambia. Los datos del servidor no: los tiene otro, pueden cambiar sin que te enteres y quieres guardarlos un rato para no pedirlos en cada pantalla. Son otra clase de estado, y **TanStack Query** es una biblioteca hecha para él.

Las piezas

Una **query** es un pedido de datos con nombre, que TanStack Query guarda y administra. El nombre es la **query key**, un arreglo que identifica esos datos: **`['pokemon', 'page', 2]`** es la página 2. La respuesta se guarda en una **caché**, una memoria de respuestas, bajo esa key: si otro componente pide la misma key, recibe lo guardado.

Los datos guardados pasan de **frescos** a **viejos** (*stale*) según **`staleTime`**: mientras están frescos, TanStack Query no vuelve a pedirlos. Por defecto se consideran viejos apenas llegan, y una query vieja se vuelve a pedir cuando un componente la usa de nuevo, cuando la ventana recupera el foco o cuando vuelve la conexión (TanStack Query, *Important Defaults*). Ese nuevo pedido se llama **refetch**. Las queries que nadie usa se borran de la caché después de **`gcTime`**, cinco minutos por defecto.

Todo eso vive en un **`QueryClient`**, que se crea una vez y se pone alrededor de la app con **`QueryClientProvider`**:

src/query/queryClient.ts

```ts
import { QueryClient } from '@tanstack/react-query'

export const queryClient = new QueryClient({
  defaultOptions: {
    queries: { staleTime: 60_000 },
  },
})
```

Aquí **`staleTime`** es de un minuto: durante ese minuto, volver a una página no hace un pedido nuevo. En **`main.tsx`**, envuelve **`<App />`** en **`<QueryClientProvider client={queryClient}>`**. Por ahora es el único proveedor; el **`main.tsx`** completo, con el de Redux, está al final de «Estado global».

useQuery

La lista paginada, con **`useQuery`**:

src/query/QueryList.tsx

```tsx
import { keepPreviousData, useQuery } from '@tanstack/react-query'
import { useState } from 'react'
import { fetchPokemonPage, PAGE_SIZE } from '../api/pokeapi'

export function QueryList() {
  const [page, setPage] = useState(1)
  const list = useQuery({
    queryKey: ['pokemon', 'page', page],
    queryFn: () => fetchPokemonPage(page),
    placeholderData: keepPreviousData,
  })

  if (list.isPending) return <p>Cargando…</p>
  if (list.isError) return <p role="alert">No se pudo cargar la lista.</p>
  const pages = Math.ceil(list.data.count / PAGE_SIZE)
  return (
    <>
      <ul aria-busy={list.isPlaceholderData}>
        {list.data.names.map((name) => (
```

```
          <li key={name}>{name}</li>
        ))}
      </ul>
      <button disabled={page === 1} onClick={() => setPage(page - 1)}>
        Anterior
      </button>
      <span>
        Página {page} de {pages}
      </span>
      <button disabled={page === pages} onClick={() => setPage(page + 1)}>
        Siguiente
      </button>
    </>
  )
}
```

Comparado con el hook a mano, no hay efecto ni estado para los datos (el **`useState`** es solo para la página): **`useQuery`** devuelve **`isPending`** (todavía no hay datos), **`isError`** y **`data`**. Cada página es una entrada distinta de la caché, porque su key cambia.

Con **`placeholderData: keepPreviousData`**, al pasar a la página 2 se sigue viendo la página 1 hasta que llega la nueva, en lugar de parpadear con «Cargando…». Mientras tanto, **`isPlaceholderData`** es **`true`**, y el **`aria-busy`** marca la lista como «en actualización» para las tecnologías de asistencia, como los lectores de pantalla.

Mutaciones e invalidación

Leer es una query; cambiar algo en el servidor es una **mutación**. El favorito de un Pokémon se guarda en un servidor de mentira, que vive en el navegador porque PokéAPI solo permite leer:

src/api/fakeServer.ts

```
const wait = (ms: number) => new Promise((resolve) => setTimeout(resolve, ms))
let favorites: string[] = []

export async function getFavorites(): Promise<string[]> {
  await wait(50)
  return [...favorites]
}

export async function setFavorite(name: string, favorite: boolean): Promise<void> {
  await wait(50)
  favorites = favorite ? [...favorites, name] : favorites.filter((current) => current
!== name)
}

export async function login(email: string, password: string): Promise<{ trainerName:
string }> {
```

```
  await wait(50)
  if (email === 'ana@dex506.test' && password === 'dex506-demo') return { trainerName:
'Ana' }
  throw new Error('Correo o contraseña incorrectos')
}
```

src/query/FavoriteToggle.tsx

```
import { useMutation, useQuery, useQueryClient } from '@tanstack/react-query'
import { getFavorites, setFavorite } from '../api/fakeServer'

export function FavoriteToggle({ name }: { name: string }) {
  const queryClient = useQueryClient()
  const favorites = useQuery({ queryKey: ['favorites'], queryFn: getFavorites })
  const toggle = useMutation({
    mutationFn: (favorite: boolean) => setFavorite(name, favorite),
    onSuccess: () => queryClient.invalidateQueries({ queryKey: ['favorites'] }),
  })

  const isFavorite = favorites.data?.includes(name) ?? false
  return (
    <button disabled={favorites.isPending || toggle.isPending} onClick={() =>
toggle.mutate(!isFavorite)}>
      {isFavorite ? '★ Favorito' : '☆ Marcar'}
    </button>
  )
}
```

`useMutation` ejecuta el cambio. Cuando sale bien, la lista de favoritos guardada en la caché quedó vieja: **invalidar** la query (**`invalidateQueries`**) la marca como vieja y, como el botón la está usando, la vuelve a pedir: así la pantalla muestra el favorito nuevo. Mientras carga o guarda, el botón queda deshabilitado. En **`favorites.data?.includes(name) ?? false`**, el **`??`** (**fusión de nulos**) da **`false`** cuando lo de la izquierda es **`null`** o **`undefined`**, es decir, mientras la lista todavía no llegó.

> – Para profundizar
>
> TanStack Query también reintenta tres veces una query que falla antes de mostrar el error (TanStack Query, *Important Defaults*). En las pruebas conviene apagarlo con **`retry: false`**, para que un error se vea enseguida.

Navegación con React Router

En esta sección vas a darle a la mini Pokédex direcciones propias, una por pantalla.

El problema: una página que no se puede compartir

La mini Pokédex es una **SPA** (*Single Page Application*): una sola página HTML que JavaScript va cambiando. Si la ficha de un Pokémon no tiene su propia dirección, no la puedes compartir, el botón «atrás» no vuelve a la lista y recargar te devuelve al principio.

Rutas

Una **ruta** es la relación entre una dirección y lo que se muestra: **`/pokedex`** muestra la lista y **`/pokedex/lagunix`** la ficha de Lagunix. **React Router** es la biblioteca que lleva esas rutas en una SPA. Se puede usar de varias formas, que su documentación llama modos; esta serie usa el **modo data**: el router se crea con **`createBrowserRouter`**, fuera de los componentes, y eso habilita funciones como la carga diferida que vas a ver más abajo. Las rutas se escriben como un arreglo de objetos:

src/router/routes.tsx

```
import { Link, NavLink, Outlet, type RouteObject } from 'react-router'
import { PokedexPage } from './PokedexPage'

function Layout() {
  return (
    <>
      <nav>
        <NavLink to="/pokedex">Pokédex</NavLink>
      </nav>
      <main>
        <Outlet />
      </main>
    </>
  )
}

function NotFound() {
  return (
    <p>
      Esta página no existe. <Link to="/pokedex">Volver a la Pokédex</Link>
    </p>
  )
}

export const routes: RouteObject[] = [
  {
    path: '/',
    element: <Layout />,
    errorElement: <p role="alert">Algo salió mal.</p>,
    children: [
      { path: 'pokedex', element: <PokedexPage /> },
      {
        path: 'pokedex/:name',
```

```
        lazy: async () => {
          const { DetailPage } = await import('./DetailPage')
          return { Component: DetailPage }
        },
      },
      { path: '*', element: <NotFound /> },
    ],
  },
]
```

- La ruta **/**, que dibuja **Layout**, es la **ruta padre** de las demás y hace de layout: la parte que comparten varias pantallas (aquí, la navegación). Sus rutas hijas se dibujan donde está **Outlet**.
- Las rutas de **children** son **rutas anidadas**: **pokedex** se suma a la **/** del padre.
- **NavLink** es un enlace que sabe si su ruta está activa, para marcarla en la navegación. **Link** es el enlace común: cambia de pantalla sin recargar la página.
- **path: '*'** atrapa cualquier dirección que no coincida: la página 404.
- **errorElement** se muestra si algo falla al renderizar o al cargar una ruta (también si falla el **import** de **lazy**), en lugar de una pantalla en blanco. Como está en la ruta **/**, reemplaza también al layout.

Con **createBrowserRouter** las rutas usan la barra de direcciones del navegador, y **RouterProvider** las pone en marcha:

src/App.tsx

```
import { createBrowserRouter } from 'react-router'
import { RouterProvider } from 'react-router/dom'
import { routes } from './router/routes'

const router = createBrowserRouter(routes)

export default function App() {
  return <RouterProvider router={router} />
}
```

RouterProvider se importa de **react-router/dom**, la parte de React Router conectada a **react-dom**, el paquete de React para el navegador.

Con el router, **App.tsx** deja de mostrar lo que tenía. Para probar los ejemplos de las secciones siguientes, ponlos en **Layout**, debajo de **<Outlet />**. Y abre **/pokedex**: en **/** el **Outlet** queda vacío, porque ninguna ruta hija coincide.

Parámetros y search params

En **pokedex/:name**, **:name** es un **parámetro**: una parte variable de la dirección. La ficha lo lee con **useParams**. En este ejemplo los nombres son de criaturas inventadas, y la ficha solo muestra el parámetro:

en la parte 4, la ficha de Dex 506 lo usa para pedir el Pokémon a PokéAPI.

src/router/DetailPage.tsx

```
import { Link, useParams } from 'react-router'

export function DetailPage() {
  const { name } = useParams()
  return (
    <>
      <h1>{name}</h1>
      <Link to="/pokedex">Volver</Link>
    </>
  )
}
```

La lista guarda el texto del buscador en la dirección, como **/pokedex?q=lag**. Lo que va después del **?** son los **search params**, y se leen y se cambian con **useSearchParams**:

src/router/PokedexPage.tsx

```
import { Link, useSearchParams } from 'react-router'

const NAMES = ['brotin', 'fogarin', 'lagunix', 'chispon']

export function PokedexPage() {
  const [params, setParams] = useSearchParams()
  const text = params.get('q') ?? ''
  const visible = NAMES.filter((name) => name.includes(text.toLowerCase()))

  return (
    <>
      <label>
        Buscar{' '}
        <input
          value={text}
          onChange={(event) =>
            setParams(event.target.value ? { q: event.target.value } : {}, { replace:
true })
          }
        />
      </label>
      <ul>
        {visible.map((name) => (
          <li key={name}>
            <Link to={`/pokedex/${name}`}>{name}</Link>
          </li>
        ))}
      </ul>
```

```
    </>
  )
}
```

Así el filtro se puede compartir y sobrevive a recargar la página. **`{ replace: true }`** hace que cada letra reemplace la entrada actual del historial en lugar de sumar una nueva: sin él, «atrás» deshace la búsqueda letra por letra. La URL es un lugar más donde puede vivir el estado.

Carga diferida

La ficha no se descarga hasta que alguien entra en ella: la ruta tiene una función **`lazy`** que la importa recién entonces, con **`import('./DetailPage')`**. Ese es un **import dinámico**: carga el archivo cuando se ejecuta la línea y devuelve una promesa con lo que exporta. Esa carga diferida mantiene liviana la descarga inicial; en la parte 3 vas a ver dónde encaja entre las técnicas de rendimiento.

> **– Para profundizar**
>
> con **`lazy`**, React Router espera a que llegue el código de la ficha antes de mostrarla. Para las partes de una pantalla que tardan en estar listas, React tiene **Suspense**, un componente que muestra un contenido de reemplazo mientras su contenido de adentro no está listo. Lo vas a usar a fondo en la parte 3.

Context

En esta sección vas a compartir un dato con muchos componentes sin pasarlo de mano en mano.

El problema: pasar una prop por niveles que no la usan

El tema claro u oscuro se decide en la página y lo usa un botón que está tres componentes más abajo: la página contiene una lista, la lista una tarjeta y la tarjeta el botón. Con props, la lista y la tarjeta lo reciben solo para pasarlo:

ANTES

src/context/PropDrilling.tsx

```
type Theme = 'light' | 'dark'

function ThemedButton({ theme }: { theme:
Theme }) {
  return <button className={theme ===
'dark' ? 'dark' :
```

DESPUÉS

src/context/WithContext.tsx

```
import { use } from 'react'
import { ThemeContext, type Theme } from
'./themeContext'

function ThemedButton() {
  const theme = use(ThemeContext)
```

```
'light'}>Marcar</button>
}

function Card({ theme }: { theme: Theme
}) {
  return <ThemedButton theme={theme} />
}

function List({ theme }: { theme: Theme
}) {
  return <Card theme={theme} />
}

export function PageWithProps({ theme }:
{ theme: Theme }) {
  return <List theme={theme} />
}
```

```
  return <button className={theme ===
'dark' ? 'dark' :
'light'}>Marcar</button>
}

function Card() {
  return <ThemedButton />
}

function List() {
  return <Card />
}

export function PageWithContext({ theme
}: { theme: Theme }) {
  return (
    <ThemeContext value={theme}>
      <List />
    </ThemeContext>
  )
}
```

Pasar una prop por niveles que no la usan se llama **prop drilling**. Con dos niveles es tolerable; con diez, cada componente intermedio carga props ajenas.

Context

Un **Context** es un canal para que un componente le pase un valor a todos los que están adentro suyo, a cualquier profundidad, sin props. Se crea con **`createContext`**, con un valor por defecto:

src/context/themeContext.ts

```
import { createContext } from 'react'

export type Theme = 'light' | 'dark'
export const ThemeContext = createContext<Theme>('light')
```

El componente que decide el valor lo **provee** envolviendo a sus hijos en **`<ThemeContext value={theme}>`**; cualquier componente de adentro lo lee con **`use(ThemeContext)`**. En la versión con Context, **`List`** y **`Card`** ya no saben nada del tema.

`use` es una función de React que lee un Context (y también una promesa, como vas a ver en la parte 3). Tiene reglas parecidas a las de un hook: solo se llama durante el render, en un componente o en un hook propio. La diferencia es que sí se puede llamar dentro de un **`if`** o de un bucle. También existe **`useContext`**, que lee un Context con todas las reglas de un hook.

Un caso típico es la sesión del usuario: casi toda la app necesita saber quién entró.

src/context/auth.tsx

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

type Auth = { trainerName: string | null; signIn: (name: string) => void; signOut: () =>
void }

const AuthContext = createContext<Auth | null>(null)

export function AuthProvider({ children }: { children: ReactNode }) {
  const [trainerName, setTrainerName] = useState<string | null>(null)
  const value = { trainerName, signIn: setTrainerName, signOut: () =>
setTrainerName(null) }
  return <AuthContext value={value}>{children}</AuthContext>
}

export function useAuth(): Auth {
  const auth = use(AuthContext)
  if (!auth) throw new Error('useAuth tiene que usarse dentro de AuthProvider')
  return auth
}
```

`AuthProvider` guarda el nombre en su estado y lo provee junto con las funciones para entrar y salir. **`useAuth`** es un hook propio que lee el Context y, si alguien lo usa fuera del proveedor, falla con un mensaje claro en vez de devolver **`null`**. En esta parte no lo conectamos a ningún componente: en la parte 4, Dex 506 envuelve la app en **`AuthProvider`**, y su login llama a **`signIn`** cuando el servidor acepta la sesión.

> **Cuidado:** cuando cambia el valor de un Context, se vuelven a renderizar todos los componentes que lo leen. Para datos que cambian seguido y que leen muchos componentes, un Context puede causar renders de más; para eso existen las bibliotecas de estado global de la última sección.

Formularios

En esta sección vas a construir formularios con validación y mensajes de error accesibles.

Dos formas de manejar un campo

En un campo **controlado**, el valor vive en el estado, como el buscador de la parte 1. En uno **no controlado**, lo guarda el navegador, y tu código lo lee recién al enviar. Los dos sirven; la diferencia es quién es el dueño del valor.

Un login con una Action de React

En React 19, el atributo **`action`** de un **`<form>`** acepta una función. Una **Action de React** es una función, normalmente asíncrona, que React ejecuta dentro de una **transición**: un cambio que React puede hacer

sin bloquear la pantalla, y que deja saber cuándo terminó.

src/forms/LoginForm.tsx

```tsx
import { useActionState } from 'react'
import { useFormStatus } from 'react-dom'
import { login } from '../api/fakeServer'

function SubmitButton() {
  const { pending } = useFormStatus()
  return <button disabled={pending}>{pending ? 'Entrando…' : 'Entrar'}</button>
}

type LoginResult = { trainerName: string | null; error: string | null; email: string }

export function LoginForm() {
  const [result, submitAction] = useActionState(
    async (_previous: LoginResult, formData: FormData): Promise<LoginResult> => {
      const email = String(formData.get('email'))
      try {
        const { trainerName } = await login(email, String(formData.get('password')))
        return { trainerName, error: null, email }
      } catch {
        return { trainerName: null, error: 'Correo o contraseña incorrectos.', email }
      }
    },
    { trainerName: null, error: null, email: '' },
  )

  if (result.trainerName) return <p>Hola, {result.trainerName}</p>
  return (
    <form action={submitAction}>
      <label>
        Correo <input name="email" type="email" defaultValue={result.email} />
      </label>
      <label>
        Contraseña <input name="password" type="password" />
      </label>
      {result.error && <p role="alert">{result.error}</p>}
      <SubmitButton />
    </form>
  )
}
```

- **`useActionState`** guarda el resultado de la Action: el primer valor es lo que devolvió la última vez, y el segundo, la función que se le pasa al atributo **`action`** del formulario. Su segundo argumento, **`{ trainerName: null, error: null, email: '' }`**, es el resultado inicial.

- La Action recibe primero el resultado anterior (**`_previous`**; el guion bajo avisa que esta Action no lo usa) y después los datos del formulario en un **`FormData`**, un objeto del navegador con el valor de cada campo según su **`name`**. Los campos son no controlados.
- Al terminar la Action, React vacía los campos no controlados del formulario, también cuando el login falla. Por eso la Action devuelve el correo, y el campo lo usa como **`defaultValue`**, su valor inicial: así no hay que escribirlo de nuevo.
- **`useFormStatus`** le dice a un componente de adentro del formulario si este se está enviando, sin pasarle props: con eso, el botón muestra «Entrando…».

`login` es la función del servidor de mentira de la sección de TanStack Query.

Un formulario de equipo, a mano y con React Hook Form

Un formulario para agregar un miembro al equipo, con apodo y nivel, validado a mano. Quien lo usa le pasa **`onSave`**, la función que recibe el miembro cuando los datos son válidos:

src/forms/MemberFormManual.tsx

```
import { useState, type SubmitEvent } from 'react'

export function MemberFormManual({ onSave }: { onSave: (member: { nickname: string;
level: number }) => void }) {
  const [nickname, setNickname] = useState('')
  const [level, setLevel] = useState('5')
  const [errors, setErrors] = useState<{ nickname?: string; level?: string }>({})

  function handleSubmit(event: SubmitEvent<HTMLFormElement>) {
    event.preventDefault()
    const next: { nickname?: string; level?: string } = {}
    const trimmed = nickname.trim()
    if (trimmed === '' || trimmed.length > 12) next.nickname = 'Entre 1 y 12 caracteres'
    const levelNumber = Number(level)
    if (!Number.isInteger(levelNumber) || levelNumber < 1 || levelNumber > 100)
next.level = 'Un número del 1 al 100'
    setErrors(next)
    if (Object.keys(next).length === 0) onSave({ nickname: trimmed, level: levelNumber
})
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Apodo <input value={nickname} onChange={(event) =>
setNickname(event.target.value)} />
      </label>
      {errors.nickname && <p>{errors.nickname}</p>}
      <label>
        Nivel <input value={level} onChange={(event) => setLevel(event.target.value)} />
```

```
      </label>
      {errors.level && <p>{errors.level}</p>}
      <button>Guardar</button>
    </form>
  )
}
```

`event.preventDefault()` evita lo que el navegador hace por defecto al enviar un formulario: mandarlo y recargar la página. **`SubmitEvent<HTMLFormElement>`** es el tipo de ese evento, que se importa de **`react`**.

Funciona, pero cada campo suma un estado, cada regla un **`if`**, y los mensajes no quedan asociados a sus campos para un lector de pantalla. **React Hook Form** es una biblioteca de formularios que registra cada campo y hace la validación por ti, y **Zod** es una biblioteca para describir la forma de los datos en un **esquema**, un objeto que sabe validar:

src/forms/memberSchema.ts

```
import { z } from 'zod'

export const MemberSchema = z.object({
  nickname: z.string().trim().min(1, 'Escribe un apodo').max(12, 'Como mucho 12
caracteres'),
  level: z.number({ error: 'Escribe un número' }).int().min(1, 'Mínimo 1').max(100,
'Máximo 100'),
})

export type Member = z.infer<typeof MemberSchema>
```

`z.infer<typeof MemberSchema>` saca de ese esquema el tipo de TypeScript, para no escribirlo dos veces. Ahí **`typeof MemberSchema`** pide el tipo de la variable **`MemberSchema`**, y los **`< >`** son genéricos, de la parte 3; por ahora, lee la línea entera como una receta: «el tipo de los datos que describe este esquema».

src/forms/MemberFormRhf.tsx

```
import { zodResolver } from '@hookform/resolvers/zod'
import { useForm } from 'react-hook-form'
import { MemberSchema, type Member } from './memberSchema'

type SaveResult = { field: 'nickname' | 'level'; message: string } | null

export function MemberFormRhf({ onSave }: { onSave: (member: Member) =>
Promise<SaveResult> }) {
  const {
    register,
    handleSubmit,
    setError,
    formState: { errors, isSubmitting },
```

```
  } = useForm<Member>({ resolver: zodResolver(MemberSchema), defaultValues: { nickname:
'', level: 5 } })

  async function submit(member: Member) {
    const serverError = await onSave(member)
    if (serverError) setError(serverError.field, { message: serverError.message })
  }

  return (
    <form onSubmit={handleSubmit(submit)} noValidate>
      <label htmlFor="nickname">Apodo</label>
      <input
        id="nickname"
        {...register('nickname')}
        aria-invalid={errors.nickname ? true : undefined}
        aria-describedby={errors.nickname ? 'nickname-error' : undefined}
      />
      {errors.nickname && <p id="nickname-error">{errors.nickname.message}</p>}
      <label htmlFor="level">Nivel</label>
      <input
        id="level"
        type="number"
        {...register('level', { valueAsNumber: true })}
        aria-invalid={errors.level ? true : undefined}
        aria-describedby={errors.level ? 'level-error' : undefined}
      />
      {errors.level && <p id="level-error">{errors.level.message}</p>}
      <button disabled={isSubmitting}>Guardar</button>
    </form>
  )
}
```

- **useForm** crea el formulario; con **zodResolver**, la validación la hace el esquema de Zod.
- **register** conecta cada campo: devuelve las props que el **<input>** necesita (**name**, **onChange**, **onBlur** y **ref**, la forma de llegar al elemento del DOM, que vas a ver en la parte 3), y **{...register('nickname')}** se las pasa todas con spread. **valueAsNumber** convierte el nivel en número.
- Aquí **onSave** es asíncrona: devuelve **null** si guardó, o el campo y el mensaje si el servidor rechazó el dato. Para probar el formulario, pásale **onSave={async (member) => { console.log(member); return null }}**.
- **handleSubmit** valida antes de llamar a **submit**: si algo no pasa, **errors** trae el mensaje de cada campo.
- **setError** pinta en su campo un error que vino del servidor, como si lo hubiera encontrado el formulario.
- **htmlFor** es el **for** de HTML (como **className** en lugar de **class**): une la etiqueta con el campo que tiene ese **id**, aunque la etiqueta no lo envuelva.

- **`noValidate`** apaga la validación propia del navegador (un campo **`type="number"`** trae la suya), para que los mensajes sean siempre los del esquema.
- **`aria-invalid`** marca el campo como inválido, y **`aria-describedby`** asocia el mensaje con el campo, para que un lector de pantalla lo lea junto con él. Esa asociación la haces tú con estos atributos: React Hook Form te da los mensajes, no los conecta solo. El mensaje va fuera del **`<label>`**: si estuviera adentro, pasaría a ser parte del nombre accesible del campo.

Cuál elegir

Para un formulario simple, de pocos campos, como un login, una Action alcanza. Para uno con validación compleja, campos que aparecen y desaparecen o varios pasos, React Hook Form con Zod. En la parte 4 vas a usar los dos.

Estado global

En esta sección vas a elegir dónde vive cada dato y a usar Zustand y Redux Toolkit.

No todo estado es igual

Tipo de estado	Ejemplo	Dónde vive
Local de un componente	si un menú está abierto	**`useState`**
De la URL	el filtro de la Pokédex	search params
Del servidor	la lista de Pokémon	TanStack Query
De un formulario	lo que se está escribiendo	React Hook Form, o el propio formulario si los campos son no controlados
Global de la interfaz	la vista preferida, la alineación de un reto	Zustand o Redux

En Dex 506, un **reto** es el desafío a un gimnasio, y la **alineación** son los tres miembros del equipo que eliges para ese reto.

Elegir bien ahorra código: si un dato es del servidor, TanStack Query ya lo guarda, y copiarlo a un estado global solo crea dos versiones que pueden no coincidir. El estado global es para lo que es de la interfaz y lo usan componentes lejanos entre sí.

Zustand

Un **store** es un lugar fuera de los componentes que guarda estado y avisa cuando cambia. Con **Zustand**, una biblioteca de estado global pequeña, un store es un hook:

src/global/preferences.ts

```ts
import { create } from 'zustand'
import { persist } from 'zustand/middleware'

type Preferences = {
  view: 'grid' | 'list'
  setView: (view: 'grid' | 'list') => void
}

export const usePreferences = create<Preferences>()(
  persist(
    (set) => ({
      view: 'grid',
      setView: (view) => set({ view }),
    }),
    { name: 'mini-pokedex.preferences' },
  ),
)
```

src/global/ViewToggle.tsx

```tsx
import { usePreferences } from './preferences'

export function ViewToggle() {
  const view = usePreferences((state) => state.view)
  const setView = usePreferences((state) => state.setView)
  return (
    <button onClick={() => setView(view === 'grid' ? 'list' : 'grid')}>
      Vista: {view === 'grid' ? 'cuadrícula' : 'lista'}
    </button>
  )
}
```

`create<Preferences>()(…)` lleva dos pares de paréntesis a propósito: el primero fija el tipo del store y el segundo recibe su definición. Es la forma que pide Zustand con TypeScript.

Cada componente lee del store solo lo que usa, con un **selector**: una función que recibe el estado y devuelve una parte (**`(state) => state.view`**). El componente vuelve a renderizar solo si esa parte cambia. **`persist`** es un **middleware** de Zustand, una función que envuelve al store para sumarle algo: aquí, lo guarda en el **`localStorage`**, un almacenamiento del navegador que guarda texto por sitio web y sobrevive a recargar la página. Por eso la preferencia sigue ahí después de recargar.

Redux Toolkit

Redux es una de las bibliotecas de estado global más usadas con React, y **Redux Toolkit** es su forma actual de escribirla. Su idea es la del reducer de la parte 1: el estado solo cambia por acciones, y una función pura calcula el estado siguiente.

src/global/lineupSlice.ts

```
import { configureStore, createSelector, createSlice, type PayloadAction } from
'@reduxjs/toolkit'

const initialState: { members: string[] } = { members: [] }

const lineupSlice = createSlice({
  name: 'lineup',
  initialState,
  reducers: {
    toggleMember(state, action: PayloadAction<string>) {
      if (state.members.includes(action.payload)) {
        state.members = state.members.filter((name) => name !== action.payload)
      } else if (state.members.length < 3) {
        state.members.push(action.payload)
      }
    },
  },
})

export const { toggleMember } = lineupSlice.actions

export function makeStore() {
  return configureStore({ reducer: { lineup: lineupSlice.reducer } })
}

export type RootState = ReturnType<ReturnType<typeof makeStore>['getState']>

export const selectLineupSummary = createSelector(
  [(state: RootState) => state.lineup.members],
  (members) => `${members.length} de 3 elegidos`,
)
```

- Una **slice** junta una parte del estado, sus reducers y las acciones que los disparan: **`createSlice`** genera **`toggleMember`**, la función que crea esa acción, a partir del reducer del mismo nombre. Una **acción de Redux** es un objeto como **`{ type: 'lineup/toggleMember', payload: 'brotin' }`**: la misma idea que la acción del reducer de la parte 1, y sin relación con la Action de React de «Formularios». **`PayloadAction<string>`** es el tipo de una acción cuyo **`payload`** es un **`string`**.
- El reducer parece modificar el estado (**`state.members.push`**), pero no lo hace: Redux Toolkit le pasa un borrador y produce un estado nuevo a partir de los cambios.
- **`configureStore`** crea el store. **`makeStore`** lo envuelve en una función para que cada prueba automática (las vas a ver en la parte 3) tenga un store nuevo; la app lo crea una sola vez.
- **`RootState`** es el tipo del estado completo. Léelo como «lo que devuelve **`getState`** del store que crea **`makeStore`**».

- **`createSelector`** crea un selector que guarda su resultado y devuelve el mismo mientras sus entradas no cambien. Recordar el resultado de un cálculo para no repetirlo se llama **memoización**; en la parte 3 la vas a ver a fondo.

src/global/Lineup.tsx

```
import { useDispatch, useSelector } from 'react-redux'
import { selectLineupSummary, toggleMember, type RootState } from './lineupSlice'

const CANDIDATES = ['brotin', 'fogarin', 'lagunix', 'chispon']

export function Lineup() {
  const members = useSelector((state: RootState) => state.lineup.members)
  const summary = useSelector(selectLineupSummary)
  const dispatch = useDispatch()
  return (
    <fieldset>
      <legend>Alineación</legend>
      {CANDIDATES.map((name) => (
        <label key={name}>
          <input
            type="checkbox"
            checked={members.includes(name)}
            onChange={() => dispatch(toggleMember(name))}
          />
          {name}
        </label>
      ))}
      <p>{summary}</p>
    </fieldset>
  )
}
```

`useSelector` lee del store y **`useDispatch`** devuelve la función para mandarle acciones. Para que funcionen, la app queda envuelta en **`<Provider>`**. Este es el **`main.tsx`** de esta parte, con los dos proveedores y el **`import './index.css'`** que en tu proyecto sigue cargando los estilos de la parte 1:

src/main.tsx

```
import { QueryClientProvider } from '@tanstack/react-query'
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import { Provider } from 'react-redux'
import App from './App.tsx'
import './index.css'
import { makeStore } from './global/lineupSlice'
import { queryClient } from './query/queryClient'

const store = makeStore()
```

```
createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <Provider store={store}>
      <QueryClientProvider client={queryClient}>
        <App />
      </QueryClientProvider>
    </Provider>
  </StrictMode>,
)
```

El store se crea una vez, fuera del render: si se creara dentro, cada render empezaría con un store vacío.

> – Para profundizar
>
> las **Redux DevTools** son una extensión del navegador, que se instala aparte, y muestran cada acción y el estado antes y después; **`configureStore`** las conecta sin configuración. Redux Toolkit trae también **RTK Query**, su propia herramienta para datos del servidor, con una idea parecida a TanStack Query. En esta serie, los datos del servidor van con TanStack Query.

Cuál elegir

Para la mayoría del estado global de una app como esta, Zustand alcanza: es poco código y se lee fácil. Redux Toolkit conviene cuando muchas personas trabajan sobre el mismo estado y quieren que cada cambio pase por una acción con nombre y quede registrado. Dex 506 usa los dos para que los conozcas: Zustand para las preferencias y Redux para la alineación de los gimnasios.

Lo que aprendiste

El render tiene que ser puro, y lo que toca el mundo exterior va en un efecto, con su limpieza; lo que se puede calcular, se calcula en el render. Los datos del servidor son otra clase de estado, y TanStack Query los pide, los guarda y los mantiene al día. React Router le da a cada pantalla su dirección, y la URL guarda filtros que se pueden compartir. Context comparte un valor sin prop drilling. Los formularios se validan con una Action o con React Hook Form y Zod, con errores accesibles. Y cada dato vive en su lugar: el componente, la URL, el servidor, el formulario o un store global.

Qué sigue

En la parte 3: React avanzado vas a ver qué hace React por dentro: cuándo y por qué vuelve a renderizar, cómo diseñar hooks propios, TypeScript avanzado, cómo medir el rendimiento, las APIs de React 19 y cómo probar tu código.

Bibliografía

- React, StrictMode. 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.
- TanStack Query, Important Defaults. Consultado el 18 de noviembre de 2026.
- PokéAPI, documentación v2. Consultada el 17 de noviembre de 2026.