Intermediate React: Data, Routing, Forms, and Global State

Created by Manuel Herrera Mesén · November 26, 2026

Part 2 of 4 react typescript tanstack-query react-router frontend serie-react

▼ IN THIS ARTICLE

- From Render to Effects
- useEffect Without Fear
- When You Do NOT Need an Effect
- Talking to the Server
- TanStack Query
- Routing with React Router
- Context
- Forms
- Global State
- What You Learned
- What's Next
- Bibliography

> Part 2 of 4 of the series "Learn React by Building a Real App". It follows part 1: React from Scratch.

> **Note:** Non-profit educational fan project. Not affiliated with, sponsored or endorsed by Nintendo, Game Freak or The Pokémon Company. Pokémon and their names are trademarks of their respective owners. Data provided by PokéAPI.

The mini Pokédex from part 1 lives in an array written by hand. A real application does not: the data sits on a server, takes time to arrive, sometimes never arrives, and the person using the app expects to share a link to a Pokémon's page, sign in, and find their preferred view still there tomorrow.

In this part we take the mini Pokédex into that world. What React does stays the same; everything around the component changes, and each tool in this part exists because solving that by hand got hard.

How to read this guide. Same as in part 1: the main text goes step by step, and the **Going deeper** boxes hold the fine detail. When a concept comes from part 1, the text links to the section where it was explained. The code is in English and the text it shows on screen is in Spanish (*Cargando...* is loading, *Guardar* is save).

I ran all of the code with 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, and react-redux 9.3.1: every example has automated tests that pass, and the messages you see are the ones the tools printed.

The examples keep using the Vite project you created in part 1. Before you start, install this part's libraries; each one is explained in its own section:

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

To see each example, do as in part 1: import it in **`App.tsx`** and place it in its **`return`**. In "Routing with React Router", **`App.tsx`** becomes the router, and that is where I tell you where to keep trying them.

By the end of this part you will be able to:

1. use effects correctly and recognize when you do not need one;
2. fetch data from an API with axios and with TanStack Query;
3. move between screens with React Router;
4. share data between components with Context;
5. build validated, accessible forms;
6. decide where each piece of data lives: in the component, in the URL, on the server, in the form, or in global state.

From Render to Effects

In this section you will see why fetching data during render does not work.

Render has to be pure

In part 1 you saw that to render is to call the component to find out what to show, and that React may do it many times. That is why render has to be a **pure function**: with the same props and the same state, it returns the same thing, and it changes nothing outside itself.

A **side effect** is exactly what render must not do: change something outside the component. Fetching data from a server, changing the tab title, or listening to the window size are all side effects.

The problem: fetching data in render

This component asks PokéAPI for a Pokémon right in the body of the function:

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)` says what shape the state has: an object with a **`name`**, or **`null`** while nothing has arrived. Those **`< >`** are TypeScript's generics syntax, which you will see in part 3; for now, read them as "this state holds this". And **`pokemon?.name`** uses **optional chaining** (**`?.`**): if **`pokemon`** is **`null`**, it gives **`undefined`** instead of failing.

Every render fires a request. When the response arrives, **`setPokemon`** stores the object that **`response.json()`** returned, which is a new object every time, even when it carries the same name. React only skips a render when the new value is equal to the old one, and two different objects never are: there is another render, which fires another request, and so on forever. In the automated test I checked it with, the server always answers with the same data, and there are still more than three requests in 300 milliseconds; they would keep coming if the test did not stop them. Code that touches the outside world needs a place outside render: effects.

useEffect Without Fear

In this section you will synchronize a component with something outside React.

What it is

`useEffect` is the hook that runs code after React has applied the render to the screen. Technically, it takes a function (the effect) and a list of **dependencies**: the values that, when they change, make the effect run again. It is for synchronizing the component with something React does not control: the tab title, a browser listener (a function the browser calls every time an event happens, like the window being resized), a request to a server.

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>
}
```

When **`pokemonName`** changes, React renders, applies the change to the screen, and then runs the effect, which updates the title. If **`pokemonName`** does not change, the effect does not run again.

Cleanup

Some effects leave something running. This one listens to the window size:

src/effects/WindowWidth.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>
}
```

The function the effect returns is the **cleanup**: React calls it before running the effect again and when the component leaves the screen. A component appearing is called **mounting**; disappearing, **unmounting**. Without the cleanup, every mount would leave one more listener behind, even after the component is gone. The empty dependency list (`[]`) says the effect runs only on mount.

> – **Going deeper**
>
> in development, with the template's **`StrictMode`**, React mounts each component and runs the effect, the cleanup, and the effect again: one extra cleanup and one extra effect (React, *StrictMode*).
> It is not a bug: it is there so a missing cleanup shows up right away. It does not happen in the production build.

Racing requests

Fetching data in an effect has a trap. If you switch quickly from one Pokémon to another, the responses can arrive out of order: the first one, slower, lands later and overwrites the second. To prevent it, the cleanup cancels the old request with an **`AbortController`**, a browser object that lets you cancel a request in flight:

src/effects/PokemonDetails.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` is a function that asks PokéAPI for one Pokémon; you will write it in "Talking to the Server". It takes the **`AbortController`**'s signal (**`controller.signal`**, the object that tells the request it was canceled), and if you abort, the request stops. Every time **`name`** changes, the cleanup cancels the previous request and the effect starts a new one. A canceled request ends with an error on purpose, which is why the **`catch`** ignores it with **`axios.isCancel`**. The **`unknown`** in **`error: unknown`** is the type of a value TypeScript knows nothing about: you have to check what it is before using it, as **`axios.isCancel`** does.

Until the Pokémon that arrived is the one requested, the component shows "Cargando…". If the request fails, it stays there and the error only shows up in the console: in "Talking to the Server" you will show it on screen.

useEffectEvent

Sometimes an effect needs to read a value without depending on it. This one logs each visit to a Pokémon along with the current theme, but changing the theme is not a new visit:

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` pulls out the part of the effect that is an event: **`logVisit`** always sees the latest values of **`pokemonName`** and **`theme`**, but it is not a dependency, so changing the theme does not rerun the effect. It arrived in React 19.2 (React, *React 19.2*).

When You Do NOT Need an Effect

In this section you will spot the most common mistake people make when they start with effects.

The problem: computing with an effect

A team's total level can be computed from its members. This component stores it in state and keeps it updated with an effect:

BEFORE

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>
}
```

AFTER

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>
}
```

The first one renders twice: once with the total at **`0`** and again with the right value, after the effect computes it. I checked it with an automated test and a **`Profiler`**, a React component that reports every time its content renders: the first version logs two renders, and the second version logs one. The template's linter already flags it:

```
⚠ 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.
```

Derived data is data computed from other data. You compute it during render, as in the second block: it is the same idea as the search box's **`visible`** in part 1. An effect is for synchronizing with something outside React, not for transforming data you already have.

> **Watch out:** something that happens because of a user action does not go in an effect either. If marking a favorite has to save it on the server, that belongs in the click handler, which is where you know it happened.

Talking to the Server

In this section you will fetch data from a real API and show its loading, error, and empty states.

Just enough HTTP

An application talks to a server over **HTTP**: it sends a request and gets a response back. The request uses a **method** that says what it wants: **`GET`** to read, **`POST`** to create, **`PUT`** to replace, **`DELETE`** to remove. The response carries a **status code**: **`200`** means it went well, **`404`** that what you asked for does not exist, **`500`** that the server failed. Data usually travels as **JSON**, a text format shaped much like JavaScript objects, though without functions or dates.

axios

The browser ships **`fetch`** for making requests. This series uses **axios**, a library that handles what **`fetch`** leaves up to you: it parses JSON on its own, treats a **`404`** or a **`500`** as an error, and lets you configure every request in one place. That place is an **instance**:

src/api/pokeapi.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 is the start of every address; **`timeout: 10_000`** stops a request that takes longer than 10 seconds (the **_** only separates the thousands so the number reads better); and an **interceptor** is a function every response passes through before it reaches your code: here, it logs errors to the console, except the cancellations from "useEffect Without Fear", which are not PokéAPI failures. With the instance in place, this part's requests look like this, in the same **`pokeapi.ts`** file, below the instance:

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) }
}
```

The two types at the top, **`PokemonPage`** and **`PokemonDetail`**, are what each function returns. In **`pokeApi.get<…>`**, the generics say what shape we assume PokéAPI's response has: TypeScript does not check it while the program runs. Each function takes that response and trims it down to what the mini Pokédex uses.

A custom hook

Requesting the list, storing the result, and knowing whether it is loading or has failed is logic you will repeat. A **custom hook** is a function of yours that groups that logic using other hooks. Technically, it is a function whose name starts with **`use`** and that calls React hooks; it follows the same rules they do.

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` plays the role the **`AbortController`** played earlier, without canceling the request: when **`page`** changes, the cleanup sets it to **`true`**, and the old response, if it arrives late, is no longer stored.

The hook remembers which page the response was for. If the requested page does not match the stored response's page, it is still loading: that way there is no need to set a "loading" state inside the effect, which is what the linter flagged in "When You Do NOT Need an Effect". The component uses the hook and shows the four states of a list that comes from a server: loading, error, empty, and with data.

src/fetching/ManualList.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"` makes a screen reader announce the message as soon as it appears.

It works. But look at what is missing: if two components ask for the same page, it is fetched twice; if you go back to a page, it is fetched again; if the server changed, nobody finds out. Solving that by hand is a lot of code, and it is what the tool in the next section does.

TanStack Query

In this section you will manage server data with TanStack Query.

The problem: server state is not UI state

What you keep with **`useState`** is yours: it only changes when your code changes it. Server data is not: someone else owns it, it can change without you noticing, and you want to keep it around for a while so you do not fetch it on every screen. It is a different kind of state, and **TanStack Query** is a library built for it.

The pieces

A **query** is a named data request that TanStack Query stores and manages. The name is the **query key**, an array that identifies that data: **`['pokemon', 'page', 2]`** is page 2. The response is stored in a

cache, a memory of responses, under that key: if another component asks for the same key, it gets what was stored.

Stored data goes from **fresh** to **stale** according to **`staleTime`**: while it is fresh, TanStack Query does not fetch it again. By default data is stale as soon as it arrives, and a stale query is fetched again when a component uses it again, when the window regains focus, or when the network comes back (TanStack Query, *Important Defaults*). That new request is called a **refetch**. Queries nobody uses are removed from the cache after **`gcTime`**, five minutes by default.

All of that lives in a **`QueryClient`**, which you create once and wrap around the app with **`QueryClientProvider`**:

src/query/queryClient.ts

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

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

Here **`staleTime`** is one minute: during that minute, going back to a page does not fire a new request. In **`main.tsx`**, wrap **`<App />`** in **`<QueryClientProvider client={queryClient}>`**. For now it is the only provider; the full **`main.tsx`**, with Redux's, is at the end of "Global State".

useQuery

The paginated list, with **`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>
    </>
  )
}
```

Compared with the hand-written hook, there is no effect and no state for the data (the **`useState`** is only for the page): **`useQuery`** returns **`isPending`** (no data yet), **`isError`**, and **`data`**. Each page is a separate cache entry, because its key changes.

With **`placeholderData: keepPreviousData`**, moving to page 2 keeps showing page 1 until the new one arrives, instead of flashing "Cargando...". Meanwhile **`isPlaceholderData`** is **`true`**, and the list's **`aria-busy`** marks it as "updating" for assistive technologies such as screen readers.

Mutations and invalidation

Reading is a query; changing something on the server is a **mutation**. A Pokémon's favorite flag is saved to a fake server that lives in the browser, because PokéAPI is read-only:

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` runs the change. When it succeeds, the favorites list stored in the cache is out of date: **invalidating** the query (**`invalidateQueries`**) marks it stale and, since the button is using it, fetches it again, so the screen shows the new favorite. While it loads or saves, the button stays disabled. In **`favorites.data?.includes(name) ?? false`**, the **`??`** (**nullish coalescing**) gives **`false`** when the left side is **`null`** or **`undefined`**, that is, while the list has not arrived yet.

> – Going deeper
>
> TanStack Query also retries a failing query three times before showing the error (TanStack Query, *Important Defaults*). In tests it pays to turn that off with **`retry: false`**, so an error shows up right away.

Routing with React Router

In this section you will give the mini Pokédex its own addresses, one per screen.

The problem: a page you cannot share

The mini Pokédex is an **SPA** (*Single Page Application*): a single HTML page that JavaScript keeps changing. If a Pokémon's page has no address of its own, you cannot share it, the back button does not return to the list, and reloading sends you back to the start.

Routes

A **route** is the link between an address and what is shown: **`/pokedex`** shows the list and **`/pokedex/lagunix`** shows Lagunix's page. **React Router** is the library that handles those routes in an SPA. It can be used in several ways, which its documentation calls modes; this series uses **data mode**: the router is created with **`createBrowserRouter`**, outside the components, and that unlocks features like the lazy loading you will see below. Routes are written as an array of objects:

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 /> },
    ],
  },
]
```

- The **/** route, which draws **Layout**, is the **parent route** of the others and acts as the layout: the part several screens share (here, the navigation). Its child routes are drawn where **Outlet** sits.
- The routes in **children** are **nested routes**: **pokedex** is added to the parent's **/**.
- **NavLink** is a link that knows whether its route is active, so the navigation can highlight it. **Link** is the plain link: it changes screens without reloading the page.
- **path: '*'** catches any address that does not match: the 404 page.
- **errorElement** is shown if something fails while rendering or loading a route (including a failed **lazy** import), instead of a blank screen. Since it sits on the **/** route, it replaces the layout too.

With **createBrowserRouter** the routes use the browser's address bar, and **RouterProvider** starts them:

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 is imported from **react-router/dom**, the part of React Router wired to **react-dom**, React's package for the browser.

With the router in place, **App.tsx** stops showing what it had. To try the examples in the next sections, put them in **Layout**, below **<Outlet />**. And open **/pokedex**: at **/** the **Outlet** stays empty, because no child route matches.

Params and search params

In **pokedex/:name**, **:name** is a **param**: a variable part of the address. The detail page reads it with **useParams**. In this example the names belong to made-up creatures and the page only shows the param: in part 4, Dex 506's detail page uses it to fetch the Pokémon from 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>
    </>
  )
}
```

The list keeps the search text in the address, as in **/pokedex?q=lag**. What comes after the **?** are the **search params**, and you read and change them with **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>
    </>
  )
}
```

That way the filter can be shared and survives a reload. **`{ replace: true }`** makes each letter replace the current history entry instead of adding a new one: without it, the back button undoes the search one letter at a time. The URL is one more place where state can live.

Lazy loading

The detail page is not downloaded until someone opens it: its route has a **`lazy`** function that imports it only then, with **`import('./DetailPage')`**. That is a **dynamic import**: it loads the file when the line runs and returns a promise with what the file exports. That lazy loading keeps the initial download light; in part 3 you will see where it fits among the performance techniques.

> **– Going deeper**
>
> with **`lazy`**, React Router waits for the detail page's code before showing it. For parts of a screen that take time to be ready, React has **`Suspense`**, a component that shows fallback content while what is inside it is not ready. You will use it in depth in part 3.

Context

In this section you will share a value with many components without handing it down one by one.

The problem: passing a prop through components that do not use it

The light or dark theme is decided by the page and used by a button three components down: the page contains a list, the list a card, and the card the button. With props, the list and the card receive it only to pass it along:

BEFORE

src/context/PropDrilling.tsx

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

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

AFTER

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>
  )
}
```

Passing a prop through components that do not use it is called **prop drilling**. Across two levels it is tolerable; across ten, every component in between carries props that are not its own.

Context

A **Context** is a channel that lets a component pass a value to everything inside it, at any depth, without props. You create it with **`createContext`**, giving it a default value:

src/context/themeContext.ts

```
import { createContext } from 'react'

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

The component that decides the value **provides** it by wrapping its children in **`<ThemeContext value={theme}>`**; any component inside reads it with **`use(ThemeContext)`**. In the Context version, **`List`** and **`Card`** no longer know anything about the theme.

`use` is a React function that reads a Context (and a promise too, as you will see in part 3). Its rules are close to a hook's: you call it only during render, in a component or a custom hook. The difference is that you can call it inside an **`if`** or a loop. There is also **`useContext`**, which reads a Context with all of a hook's rules.

A typical case is the user's session: almost the whole app needs to know who signed in.

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` keeps the name in its state and provides it along with the functions to sign in and out. **`useAuth`** is a custom hook that reads the Context and, if someone uses it outside the provider, fails with a clear message instead of returning **`null`**. In this part we do not connect it to any component: in part 4, Dex 506 wraps the app in **`AuthProvider`**, and its login calls **`signIn`** when the server accepts the session.

> Watch out: when a Context's value changes, every component that reads it renders again. For data that changes often and that many components read, a Context can cause extra renders; that is what the global state libraries in the last section are for.

Forms

In this section you will build forms with validation and accessible error messages.

Two ways to handle a field

In a **controlled** field, the value lives in state, like part 1's search box. In an **uncontrolled** one, the browser keeps it, and your code reads it only on submit. Both work; the difference is who owns the value.

A login with a React Action

In React 19, a **`<form>`**'s **`action`** attribute accepts a function. A **React Action** is a function, usually asynchronous, that React runs inside a **transition**: a change React can make without blocking the screen, and that lets you know when it finished.

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`** stores the Action's result: the first value is what it returned last time, and the second is the function you pass to the form's **`action`** attribute. Its second argument, **`{ trainerName: null, error: null, email: '' }`**, is the initial result.
- The Action first receives the previous result (**`_previous`**; the underscore signals that this Action does not use it) and then the form data in a **`FormData`**, a browser object holding each field's value by its **`name`**. The fields are uncontrolled.

- When the Action finishes, React clears the form's uncontrolled fields, even when the login fails. That is why the Action returns the email, and the field uses it as its **`defaultValue`**, its initial value: you do not have to type it again.
- **`useFormStatus`** tells a component inside the form whether the form is submitting, without props: that is how the button shows "Entrando..." (signing in).

`login` is the function from the fake server in the TanStack Query section.

A team form, by hand and with React Hook Form

A form to add a member to the team, with a nickname and a level, validated by hand. Whoever uses it passes **`onSave`**, the function that receives the member once the data is valid:

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()` stops what the browser does by default when a form is submitted: send it and reload the page. **`SubmitEvent<HTMLFormElement>`** is the type of that event, imported from **`react`**.

It works, but every field adds a piece of state, every rule an **`if`**, and the messages are not tied to their fields for a screen reader. **React Hook Form** is a form library that registers each field and does the validation for you, and **Zod** is a library for describing the shape of data in a **schema**, an object that knows how to validate:

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>` derives the TypeScript type from that schema, so you do not write it twice. There, **`typeof MemberSchema`** asks for the type of the **`MemberSchema`** variable, and the **`< >`** are generics, from part 3; for now, read the whole line as a recipe: "the type of the data this schema describes".

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`** creates the form; with **`zodResolver`**, the Zod schema does the validation.
- **`register`** connects each field: it returns the props the **`<input>`** needs (**`name`**, **`onChange`**, **`onBlur`**, and **`ref`**, the way to reach the DOM element, which you will see in part 3), and **`{...register('nickname')}`** passes them all with spread. **`valueAsNumber`** turns the level into a number.
- Here **`onSave`** is asynchronous: it returns **`null`** if it saved, or the field and the message if the server rejected the data. To try the form, pass it **`onSave={async (member) => { console.log(member); return null }}`**.
- **`handleSubmit`** validates before calling **`submit`**: if something fails, **`errors`** holds each field's message.
- **`setError`** puts an error that came from the server on its field, as if the form had found it.
- **`htmlFor`** is HTML's **`for`** (like **`className`** instead of **`class`**): it ties the label to the field with that **`id`**, even when the label does not wrap it.
- **`noValidate`** turns off the browser's own validation (a **`type="number"`** field has its own), so the messages are always the schema's.
- **`aria-invalid`** marks the field as invalid, and **`aria-describedby`** ties the message to the field, so a screen reader reads them together. You make that link yourself with these attributes: React Hook

Form gives you the messages, it does not wire them up on its own. The message sits outside the **`<label>`**: inside it, it would become part of the field's accessible name.

Which to choose

For a simple form with few fields, like a login, an Action is enough. For one with complex validation, fields that come and go, or several steps, React Hook Form with Zod. In part 4 you will use both.

Global State

In this section you will decide where each piece of data lives and use Zustand and Redux Toolkit.

Not all state is the same

Kind of state	Example	Where it lives
Local to a component	whether a menu is open	**`useState`**
In the URL	the Pokédex filter	search params
From the server	the list of Pokémon	TanStack Query
In a form	what is being typed	React Hook Form, or the form itself when the fields are uncontrolled
Global UI state	the preferred view, the lineup for a challenge	Zustand or Redux

In Dex 506, a **challenge** is taking on a gym, and the **lineup** is the three team members you pick for that challenge.

Choosing well saves code: if a piece of data comes from the server, TanStack Query already stores it, and copying it into global state only creates two versions that may disagree. Global state is for data that belongs to the interface and is used by components far apart from each other.

Zustand

A **store** is a place outside the components that holds state and announces when it changes. With **Zustand**, a small global state library, a store is a 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

```
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>()(…)` has two pairs of parentheses on purpose: the first sets the store's type and the second receives its definition. It is the pattern Zustand requires with TypeScript.

Each component reads from the store only what it uses, with a **selector**: a function that takes the state and returns a piece of it (**`(state) => state.view`**). The component renders again only if that piece changes. **`persist`** is a Zustand **middleware**, a function that wraps the store to add something to it: here, it saves it in **`localStorage`**, browser storage that keeps text per website and survives a reload. That is why the preference is still there after a reload.

Redux Toolkit

Redux is one of the most widely used global state libraries with React, and **Redux Toolkit** is the current way to write it. Its idea is the reducer from part 1: state only changes through actions, and a pure function computes the next state.

src/global/lineupSlice.ts

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

```ts
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`,
)
```

- A **slice** groups a piece of state, its reducers, and the actions that trigger them: **`createSlice`** generates **`toggleMember`**, the function that creates that action, from the reducer of the same name. A **Redux action** is an object like **`{ type: 'lineup/toggleMember', payload: 'brotin' }`**: the same idea as the reducer action in part 1, and unrelated to the React Action from "Forms". **`PayloadAction<string>`** is the type of an action whose **`payload`** is a **`string`**.
- The reducer looks like it modifies the state (**`state.members.push`**), but it does not: Redux Toolkit hands it a draft and produces a new state from the changes.
- **`configureStore`** creates the store. **`makeStore`** wraps it in a function so each automated test (you will see them in part 3) gets a fresh store; the app creates it only once.
- **`RootState`** is the type of the whole state. Read it as "what **`getState`** returns for the store that **`makeStore`** creates".
- **`createSelector`** creates a selector that remembers its result and returns the same one while its inputs do not change. Remembering a computation's result so you do not repeat it is called **memoization**; you will see it in depth in part 3.

src/global/Lineup.tsx

```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` reads from the store and **`useDispatch`** returns the function that sends it actions. For them to work, the app is wrapped in **`<Provider>`**. This is this part's **`main.tsx`**, with both providers and the **`import './index.css'`** that, in your project, keeps loading part 1's styles:

src/main.tsx

```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>,
)
```

The store is created once, outside render: if it were created inside, every render would start with an empty store.

> – **Going deeper**
>
> the **Redux DevTools** are a browser extension, installed separately, that shows every action and the state before and after; **`configureStore`** connects to them with no setup. Redux Toolkit also ships **RTK Query**, its own tool for server data, with an idea similar to TanStack Query. In this series, server data goes through TanStack Query.

Which to choose

For most of the global state in an app like this one, Zustand is enough: it takes little code and reads easily. Redux Toolkit pays off when many people work on the same state and want every change to go through a named action and be recorded. Dex 506 uses both so you get to know them: Zustand for the preferences and Redux for the gym lineup.

What You Learned

Render has to be pure, and whatever touches the outside world goes in an effect, with its cleanup; whatever can be computed is computed during render. Server data is a different kind of state, and TanStack Query fetches it, stores it, and keeps it current. React Router gives each screen its address, and the URL holds filters you can share. Context shares a value without prop drilling. Forms are validated with an Action or with React Hook Form and Zod, with accessible errors. And each piece of data lives in its place: the component, the URL, the server, the form, or a global store.

What's Next

In part 3: Advanced React you will see what React does under the hood: when and why it renders again, how to design custom hooks, advanced TypeScript, how to measure performance, the React 19 APIs, and how to test your code.

Bibliography

- React, StrictMode. Accessed November 18, 2026.
- The React Team, React 19.2, October 1, 2025. Accessed November 18, 2026.
- TanStack Query, Important Defaults. Accessed November 18, 2026.
- PokéAPI, v2 documentation. Accessed November 17, 2026.