# Advanced React: Performance, React 19, Testing, and Architecture

Created by Manuel Herrera Mesén · November 28, 2026

Part 3 of 4 · react · typescript · rendimiento · testing · frontend · serie-react

▼ IN THIS ARTICLE


- How React Works Under the Hood
- Custom Hooks
- Refs and the DOM
- Advanced TypeScript for React
- Performance
- React 19 in Depth
- Testing
- Architecture, Quality, and Security
- The Bigger Picture and the Road to Senior
- What You Learned
- What's Next
- Bibliography


> Part 3 of 4 of the series "Learn React by Building a Real App". It follows part 2: Intermediate React.

> **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 2 fetches data, navigates, validates forms, and remembers preferences. Then one day something odd happens: you switch Pokémon and the notes field keeps what you wrote for the previous one; an effect runs even though nothing it uses changed; the list lags behind every key you press. None of those problems shows up when you read the component. To understand them you need to know what React does under the hood, and to avoid them, you need to measure and test.

That is what this part is about: the model of how React renders, the tools for writing code that others can reuse, measure, and test, and what separates an exercise from a professional application.

**How to read this guide.** Same as in the previous parts: the main text goes step by step, and the **Going deeper** boxes hold the fine detail. This part assumes you did parts 1 and 2; when a concept comes from them, the text links to the section where it was explained. When it says "the example's test", it means the automated tests I checked each example with: in "Testing" you learn to write them, and several are shown there. The code is in English and the text it shows on screen is in Spanish (*Vistas* is views, *Buscar* is search).

I ran all of the code with React 19.3.1, TypeScript 7.0.2, TanStack Query 5.108.0, Vitest 5.1.0, Testing Library 16.3.3, and MSW 3.0.7: every example has automated tests that pass, and the error messages you see are the ones the tools printed. I ran Dex 506's end-to-end tests with Playwright 1.65.1 in Chromium.

The examples keep using the Vite project from the previous parts. This part adds no libraries to the app: Vitest, Testing Library, and MSW are installed as development dependencies in "Testing". To see an example, put it in part 2's `Layout`, below `<Outlet />`. If it takes props, pass it test values: `onSearch={console.log}`, `onSave={console.log}` or, for a `save` that returns a promise, `save={() => new Promise((resolve) => setTimeout(resolve, 500))}`. The error files, such as `list-typo.tsx`, exist to show the compiler's message: you do not need to copy them.

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

1. explain when and why React renders a component again;
2. design custom hooks that others can reuse;
3. use refs to touch the DOM or keep values without rendering;
4. type components and hooks with advanced TypeScript;
5. measure performance before optimizing it, React Compiler included;
6. use the React 19 APIs;
7. test your code at three levels and organize a real project with quality and security criteria.

## How React Works Under the Hood

In this section you will build the mental model that explains almost every "odd behavior" in React.

### Render and commit

In part 1 you saw that rendering is calling the component to find out what to show. Now it needs to be more precise, because updating the screen takes two steps:

- The **render phase**: React calls your components and works out what the screen should look like. It does not touch the DOM. That is why render has to be pure: React may call it, throw the result away, and call it again.
- The **commit phase**: React applies to the DOM the differences it worked out during render, by comparing the new result with the previous one. Then the effects run.

A component rendering does not mean the DOM changes: if the result matches the previous one, the commit touches nothing.

### What makes a component render

A component renders again for three reasons:

1. **its state changed**, through a **`set`** from **`useState`** or a **`dispatch`** from **`useReducer`**;
2. **its parent rendered**: by default, when a component renders, the components it creates in its **`return`** render too, even if their props are the same;
3. **a Context it reads changed**, as you saw in "Context".

The second one is the most surprising. To see it, this **`logRender`** writes to the console every render a **`Profiler`** reports, the React component that measures renders and that you met in part 2:

src/render/logRender.ts

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

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

src/render/ParentRenders.tsx

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

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

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

**`TrainerBadge`** takes no props and has no state, yet every click on "Vistas" renders it again: the example's test logs one **`mount`** and then one **`update`** per click. **`phase`** says whether it was the first render (**`mount`**) or a later one (**`update`**), and **`actualDuration`**, how long it took.

That is almost always fine: rendering is cheap, and the commit only touches what changed. In "Performance" you will see when it stops being fine and what to do.

> – Going deeper

the **`Profiler`** measures in development. In the production build it is turned off, unless you use a special profiling build (React, *Profiler*).

> – Going deeper
>
> a component that arrives as **`children`** was already created by another component. If the one receiving it renders because of its own state, **`children`** is the same element as before, and React does not render it again. Passing components as **`children`** is a way to avoid renders without memoizing anything.

## State lives at a position in the tree

To decide what changed, React compares the new tree with the previous one position by position: if the same spot holds a component of the same type, React keeps it, along with its state. That process is the **reconciliation** you saw in part 1. The consequence: state does not belong to the data a component shows but to its place in the tree.

This notes field keeps what you type in its own state:

src/render/NoteField.tsx

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

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

BEFORE

src/render/NotesWithoutKey.tsx

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

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

AFTER

src/render/NotesWithKey.tsx

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

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

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

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

In the "before" block, you write a note about Brotín, switch to Lagunix, and the note is still there: to React it is the same **`NoteField`**, in the same place, just with a different prop. In the "after" block, **`key={creature}`** tells React that each creature is a different field. When the **`key`** changes, React throws away the previous component, state included, and creates a new one: the field starts empty. It is the same **`key`** from part 1's lists, used outside a list to reset a component.

## Identity: two equal objects are not the same object

To know whether something changed (an effect dependency, a state value, the props of a memoized component, which you will see in "Performance"), React uses **`Object.is`**: it compares numbers, strings, and booleans by value, but objects, arrays, and functions by **identity**, that is, whether they are the same object in memory. Two objects created separately are never the same, even when they hold the same data.

This component builds a filters object on every render and uses it as an effect dependency:

BEFORE

src/render/TypeSearchWithObject.tsx

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

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

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

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

AFTER

src/render/TypeSearch.tsx

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

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

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

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

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

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

In the "before" version, every render creates a new **`filters`**, the effect sees a different dependency, and calls **`onSearch`** again, even though only the views changed. The example's test checks it: one click on "Vistas" triggers one extra search. The linter already flags it:

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

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

You will see the two tools the linter suggests later on, and for an object neither is the right one: the right one memoizes values and is in "Performance". The simplest way out is the "after" version: depend on the primitive value, **`type`**, and build the object inside the effect.

> *"Coming from Vue: Vue's reactivity system knows which components read each piece of data and renders only those again. In React, rendering a component renders the ones it creates, and what prevents that is composition with **`children`** or memoization, written by you or done by React Compiler, as you will see in "Performance".*"

## Custom Hooks

In this section you will design hooks that others can reuse without reading how they are built.

## The rules of hooks

In part 2 you wrote your first custom hook, **`usePokemonList`**. Before designing more, it is worth understanding the two rules every hook follows (**`use`** is the exception to the first one: you can call it inside an **`if`** or a loop, but it still only goes in components or hooks):

1. **Call them only at the top level** of the component or hook: never inside an **`if`**, a loop, or an inner function.
2. **Call them only from components or from other hooks**, not from plain functions.

The reason is how React stores state: not by name but by order. A component's first **`useState`** is box 1, the second, box 2. If a hook is only called some of the time, the order changes from one render to the next: React notices and fails with an error ("Rendered more hooks than during the previous render"), or, if the number of hooks happens to match, gives each one another's state. This component breaks the first rule:

conditional-hook.tsx

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

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

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

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

The fix is to always call the hook, before any **`return`**, and decide what to show afterward.

## useDebounce

A search box that fetches on every key makes five requests to type "lagun". To **debounce** is to wait until a value stops changing for a while before using it. That logic does not depend on what you search for, so it goes in a custom hook:

src/hooks/useDebounce.ts

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

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

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

  return debounced
}
```

Every time **`value`** changes, the effect schedules a timer; if **`value`** changes again before it fires, the cleanup cancels it and another one is scheduled. Only when **`delayMs`** milliseconds pass without changes does **`debounced`** take the new value.

## Composing hooks

A custom hook can use other hooks, including a library's. **`usePokemon`** wraps part 2's TanStack Query query in a name that says what it does:

src/hooks/usePokemon.ts

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

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

TanStack Query passes **`queryFn`** a **`signal`** to cancel the request, the same idea as part 2's **`AbortController`**. **`enabled: name !== ''`** avoids fetching a Pokémon with no name.

With both hooks, the search box looks like this:

src/hooks/PokemonSearch.tsx

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

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

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

The component does not know how the waiting or the fetching works: it reads top to bottom. The example's test types "Brotin" and counts a single request, with the full name. In the browser, type the name of a real Pokémon: the creatures in the tests are made up.

## A badly designed hook

This hook does the same thing as the search above, but mixes everything together:

src/hooks/usePokemonSearch.ts

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

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

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

  return { text, setText, pokemon }
}
```

It works, but, compared with **`PokemonSearch`**:

- **it does three jobs**: it keeps the text, changes the tab title, and fetches data. Anyone who only wants the search gets the title too;
- **it owns the text**: the component cannot pass it a text that comes from somewhere else, such as the URL;
- **it fetches on every letter and never cancels**: the test counts six requests for "brotin", and an old response can overwrite a new one, part 2's race.

The comparison gives you the rules for designing a hook: it does **one thing**; it takes what it needs as **parameters** instead of keeping it inside; it returns named values; and it has no effects its user does not expect.

## Refs and the DOM

In this section you will keep values that do not affect the screen and touch the DOM when React is not enough.

### The problem: a value that is not state

A training stopwatch needs to keep the identifier of the interval that makes it tick, so it can stop it later. That identifier is never shown: storing it in state would cause a render for no reason. And a plain variable does not work: it is lost, because every render runs the component function again from the top.

### useRef

A **ref** is a box React keeps between renders and that does not cause a render when it changes. Technically, **`useRef`** returns an object **`{ current }`**: the same object on every render of the component, whose **`current`** you can read and change whenever you want, except during render.

src/refs/Stopwatch.tsx

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

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

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

  function stop() {
```

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

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

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

- **`intervalId.current`** holds the identifier while the stopwatch runs, and **`null`** when it is stopped: that way pressing "Empezar" (start) twice does not create two intervals. The example's test checks it.
- The seconds are state, because they are shown.
- The effect, with an empty dependency list, only has a cleanup: if the component unmounts while the stopwatch is running, it stops the interval and sets the ref back to **`null`**, ready to start again.

> Watch out: do not read or change **`ref.current`** during render. Render has to be pure, and a ref does not announce when it changes: the screen would show a stale value. Refs are used in event handlers and in effects.

## Refs to the DOM

The other use of a ref is reaching a DOM element. If you pass the ref in an element's **`ref`** attribute, React stores the DOM node there after the commit. With that, this form moves focus to the field with the error, something props and state cannot do:

src/refs/TrainerForm.tsx

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

export function TrainerForm({ onSave }: { onSave: (name: string) => void }) {
  const nameRef = useRef<HTMLInputElement>(null)
  const id = useId()
  const [error, setError] = useState<string | null>(null)
```

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

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

**`flushSync`**, from **`react-dom`**, forces React to apply the state change to the DOM before moving on. Without it, focus would reach the field before it has **`aria-invalid`** and **`aria-describedby`**, and a screen reader would announce it without the error.

**`useId`** generates a unique, stable identifier for each instance of the component. With a fixed **`id`** such as **`"name"`**, two forms on the same page would have two fields with the same **`id`**, and the second label's **`htmlFor`** would point at the first field. With **`useId`**, each form has its own; the example's test mounts two and checks they differ. It is not for a list's **`key`**s: those come from the data.

## ref as a prop

Sometimes the ref has to reach an element inside another component. Since React 19, a function component receives **`ref`** as just another prop (React, *React v19*):

src/refs/SearchInput.tsx

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

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

src/refs/SearchAgain.tsx

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

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

**`Ref<HTMLInputElement>`** is the type of a ref that points at an **`<input>`**. Before React 19 this required wrapping the component in **`forwardRef`**; the React team has announced it will deprecate it.

## useImperativeHandle

Passing the whole **`<input>`** gives the parent access to the entire element. **`useImperativeHandle`** lets the component decide what its ref exposes: here, only **`focus`** and **`clear`**.

src/refs/LimitedSearch.tsx

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

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

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

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

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

The parent creates the ref with **`useRef<SearchHandle>(null)`**, passes it as **`ref`**, and calls, for example, **`searchRef.current?.clear()`** from a handler. It receives an object with those two methods and nothing else, and the component can change how it works inside without breaking whoever uses it.

> – **Going deeper**
>
> the **`ref`** attribute also accepts a function, a **callback ref**. React calls it with the node when the element appears, and since React 19 the function can return a cleanup, which React calls when the element goes away (React, *React v19*). It is useful, for example, to start observing an element when it mounts and stop when it unmounts.

> *"Coming from Vue or Angular: a DOM ref is Vue's template ref (**`ref="input"`** and **`useTemplateRef`**) and Angular's **`viewChild`**."*

## Advanced TypeScript for React

In this section you will use types so that impossible states do not compile.

### Generics

You have already used several: **`useState<{ name: string } | null>`**, **`pokeApi.get<…>`**, **`z.infer<typeof MemberSchema>`**. A **generic** is a type that takes another type as a parameter, the way a function takes values. Technically, it is a type or function with a **type parameter** (by convention, **`T`**) that gets fixed when it is used.

A list that works for any data:

src/typescript/List.tsx

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

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

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

src/typescript/Creatures.tsx

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

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

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

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

**`List<T>`** does not know what is in **`items`**. When you use it, TypeScript **infers** **`T`** from what you pass: since **`TEAM`** is a **`Creature[]`**, **`T`** is **`Creature`**, and inside **`getKey`** and **`renderItem`** each **`creature`** is a **`Creature`**. That is why a typo never gets to run:

list-typo.tsx

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

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

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

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

> *"In Dex 506: the app's **useDebounce** is generic. It works for a string, a number, or an object, and the type it returns is the same type it receives:"*

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

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

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

  return debounced
}
```

## Discriminated unions

A request can be loading, have failed, or have succeeded. With three loose fields, nothing prevents combinations that make no sense:

BEFORE

src/typescript/requestFlags.ts

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

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

AFTER

src/typescript/RequestView.tsx

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

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

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

**impossible** compiles: loading, failed, and with data at the same time. A **discriminated union** is a union of objects that share a field with a different literal value in each, the **discriminant** (here, **status**). When you check **state.status**, TypeScript knows which variant you are in, the same narrowing from part 1: inside **case 'error'** there is a **message**, and inside **case 'success'** there is **data**. Inside the **error** variant there is no **data** to read.

The **default** does an **exhaustiveness check**. **never** is the type with no values: if every variant was handled, nothing reaches **default** and **state** is **never**. If someone adds a variant tomorrow and does not handle it, **state** is no longer **never** and the compiler complains:

new-status.tsx

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

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

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

## Inheriting an element's props

Your own button should accept everything a **<button>** accepts: **disabled**, **onClick**, **aria-label**. **ComponentProps<'button'>** is the type of a **<button>**'s props:

src/typescript/Button.tsx

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

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

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

The **`&`** adds the button's own prop, **`variant`**, and **`...props`** gathers everything else to pass it to the **`<button>`** with part 1's spread. **`type="button"`** comes before the spread: by default, a **`<button>`** inside a form submits it, and this button should not unless whoever uses it passes another **`type`**, which the spread puts on top.

## Utility types, satisfies, and as const

TypeScript ships **utility types**: generics that transform another type.

src/typescript/types.ts

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

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

| Type | What it does |
| --- | --- |
| **`Pick<T, 'a' \| 'b'>`** | keeps only those properties |
| **`Omit<T, 'a'>`** | removes those properties |
| **`Partial<T>`** | makes every property optional |
| **`Record<K, V>`** | an object with keys of type **`K`** and values of type **`V`** |

**`satisfies`** checks that a value matches a type without changing the type TypeScript infers for it. **`as const`** locks the literal values of an array or object, so they do not widen to **`string`**:

src/typescript/types.ts

```tsx
export const TYPE_COLORS = {
  grass: '#3f9b4a',
  fire: '#d9572b',
  water: '#2f7fd1',
```

```
} satisfies Record<string, string>

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

**`VIEWS`** is **`readonly ['grid', 'list']`**: a read-only array holding exactly those two values. **`typeof VIEWS`** is its type, and **`[number]`** asks for the type of any of its elements (what **`VIEWS[0]`** or **`VIEWS[1]`** would have): the union **`'grid' | 'list'`**. The difference between **`satisfies`** and a type annotation shows up with a typo:

colors-typo.ts

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

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

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

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

There is only one error. With **`satisfies`**, TypeScript keeps the real keys (**`grass`** and **`fire`**) and flags **`grasss`**. With the **`Record<string, string>`** annotation, any key is valid, and the typo slips through.

### Event types

Handlers receive typed events: **`SubmitEvent<HTMLFormElement>`** for a form's **`onSubmit`** (you used it in part 2), **`ChangeEvent<HTMLInputElement>`** for a field's **`onChange`**, and **`MouseEvent<HTMLButtonElement>`** for a click. They are imported from **`react`**, with **`import type { MouseEvent } from 'react'`**: without that import, **`MouseEvent`** and **`SubmitEvent`** are the DOM's, which are not generic. The generic says which element the event comes from, which is why **`event.currentTarget`**, the element that has the handler, has the right type. When you write the handler inline (**`onChange={(event) => …}`**), TypeScript infers it for you; when you write it separately, like **`TrainerForm`**'s **`handleSubmit`**, you add the type yourself.

## Performance

In this section you will measure before optimizing and get to know the tools that avoid extra work.

## The problem: optimizing blind

Optimizing without measuring is guessing, and every optimization has a cost: code that is harder to read and comparisons that take time too. The order is: measure, find the slow part, optimize that part, and measure again.

To measure you have the **`Profiler`** from "How React Works Under the Hood", the Profiler tab in React DevTools (a browser extension) and, since React 19.2, React's own tracks in the Chrome DevTools Performance panel (React, *React 19.2*).

## memo

The **memoization** you saw in part 2 with **`createSelector`**, applied to components: **`memo`** wraps a component so it does not render again when its props are equal to the previous render's, compared one by one with **`Object.is`**.

src/performance/CreatureList.tsx

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

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

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

export const CreatureList = memo(CreatureListBase)
```

The **`Profiler`** sits inside the list on purpose: that way it only reports when the list really renders. Wrapped around **`CreatureList`** from outside, it would report every time the parent renders, even when **`memo`** skipped the list.

## Why memo is not enough

**`memo`** compares props, and the identity from "How React Works Under the Hood" shows up again. Both pages use this list of names:

src/performance/names.ts

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

BEFORE

src/performance/PageWithNewProps.tsx

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

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

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

AFTER

src/performance/PageWithStableProps.tsx

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

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

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

In the "before" version, every render creates a new **`names`** array with **`filter`** and a new function for **`onSelect`**. The props are never equal, and **`memo`** prevents nothing: the example's test logs one list render for every click on "Vistas". In the "after" version:

- **`useMemo`** keeps a computation's result and returns the same one while its dependencies do not change: **`names`** is the same array until **`selected`** changes;
- **`useCallback`** does the same for a function: **`handleSelect`** is the same on every render.

Now a click on "Vistas" does not render the list, and picking a creature does, because **`names`** changed.

> **Watch out:** memoizing has a cost: memory to keep the results, comparisons on every render, and code that is harder to read. Use it when a measurement shows a slow render, or when identity matters: props of a `memo` component or effect dependencies.

## React Compiler

**React Compiler** does that memoization for you, at build time: it analyzes each component and hook and memoizes the values and elements that need it, without `memo`, `useMemo`, or `useCallback` by hand. Its stable 1.0 release came out on October 7, 2025 (Lauren Tan, Joe Savona, and Mofei Zhang, *React Compiler v1.0*). It assumes your code follows the rules of React (pure render and hooks in order): if it detects a component that does not, it leaves that component unoptimized. **Babel** is a JavaScript translator that runs during the build, and the compiler is a Babel plugin. In Dex 506 it runs through the `@rolldown/plugin-babel` plugin inside the Vite configuration; in part 4 you install it and see the whole file, imports included:

apps/web/vite.config.ts

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

This part's examples do not use it, so you can see what happens without memoization.

Is it worth it? I measured it in Dex 506 with the app's most expensive case: typing in the Pokédex search box with all 151 Pokémon on screen, since every keystroke re-renders the page. I used the production build, a CPU slowed down four times, and five runs of ten keystrokes, timing each one from the press to the next frame. I ran two rounds of measurements:

| Variant | Round 1: median · p75 | Round 2: median · p75 |
|---|---|---|
| No memoization | 9.9 ms · 10.8 ms | 9.8 ms · 10.2 ms |

| Variant | Round 1: median · p75 | Round 2: median · p75 |
|---|---|---|
| **`memo`** by hand on the card | 5.0 ms · 5.5 ms | 4.9 ms · 5.7 ms |
| React Compiler | 5.6 ms · 8.9 ms | 5.1 ms · 5.5 ms |

*Apple M3 Pro, Chromium 159, November 17, 2026. p75 is the 75th percentile: three out of four key presses took that long or less.*

The compiler cuts the median nearly in half, the same as **`memo`** by hand, without touching a single component. In the first round its 75th percentile was higher (8.9 ms), a run-to-run variation the second round did not repeat. And it has to be said plainly: without memoization, 9.8 ms already fits in a 16 ms frame (one frame at 60 frames per second), so in this app the difference exists but you would not notice it. Dex 506 keeps the compiler on for what happens as the app grows: nobody has to remember to memoize.

## Splitting code and virtualizing

Two more techniques, for problems Dex 506 does not have:

- **Code splitting** means dividing the JavaScript into pieces that download when they are needed. You already did it in part 2 with the detail page's lazy loading: Vite puts each dynamic **`import()`** in a separate file.
- **Virtualizing** a list means rendering only the rows on screen and swapping them as you scroll. With thousands of rows, it is the difference between a list that stutters and one that scrolls smoothly; with Dex 506's 151 it is not needed, as the measurement showed. Libraries such as TanStack Virtual handle it.

## useDeferredValue and useTransition

In part 2 you saw that a transition is a change React can make without blocking the screen. Now the full definition: an update inside a **transition** is low priority, and if an urgent one arrives (a key press, a click) React throws away the half-finished transition render, handles the urgent one, and then starts it over.

**`useDeferredValue`** takes a value and returns a version that lags behind while there is urgent work:

src/performance/DeferredSearch.tsx

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

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

```
    </ul>
  )
})

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

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

The field uses **`text`** and updates on every keystroke; the list uses **`deferredText`**, which React updates in the background. While they differ, **`isStale`** dims the list with an inline **`style`**, an object of CSS properties. **`Results`** uses **`memo`** on purpose: without it, it would re-render on every keystroke anyway, because its parent renders.

**`useTransition`** is for when you make the change yourself, with a **`set`**. This example switches between two panels of a detail page, stats and moves, and the moves panel has a filter:

src/performance/panels.tsx

```
import { useState } from 'react'

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

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

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

src/performance/Tabs.tsx

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

type Tab = 'stats' | 'moves'

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

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

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

**`startTransition`** marks the tab change as a transition, and **`isPending`** is **`true`** while React prepares it. The rule for choosing: if you have the **`set`**, **`useTransition`**; if you receive the value from elsewhere (a prop, another hook's result), **`useDeferredValue`**.

With examples this small you will not see the dimmed list or "Cambiando de pestaña..." (switching tabs): they show up when the list or the panel is slow to render, such as a list with thousands of items.

> *"Coming from Vue or Angular: Vue's **`computed`** and Angular's already cache their result until their dependencies change. In React, **`useMemo`** is the version you write by hand, and React Compiler, the automatic one."*

## React 19 in Depth

In this section you will use the React 19 APIs that replace what used to take a library or an effect.

### use, Suspense, and error boundaries

In part 2 you used **use** to read a Context. It also reads a promise: if the promise has not settled, the component **suspends**, and the nearest **Suspense** shows its fallback content.

src/react19/PokemonName.tsx

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

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

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

src/react19/PickPokemon.tsx

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

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

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

  return (
    <>
      <button onClick={show}>Ver el Pokémon número 3</button>
```

```
      {request && <PokemonName key={request.id} pokemonPromise={request.promise} />}
    </>
  )
}
```

- **Suspense** shows "Cargando..." (loading) while the promise is pending and the name when it settles.
- If the promise fails, the error travels up to the nearest **error boundary**, which shows its fallback instead of leaving a blank screen.
- The promise is created on click, outside render. If it were created during render, every render would create a new promise and the component would suspend forever.
- Each request gets a different **`key`**: that way, a new request after an error mounts a new error boundary, without the previous error. It is the **`key`** from "How React Works Under the Hood".

The error boundary is the only place in the series where you will write a **class**: today there is no way to write one as a function component (React, *Component*). A class is the way components were defined before functions: **`extends Component`** makes it a component, **`state`** is its state, **`this.props`** are its props, and **`static`** marks a method that belongs to the class rather than to each component created from it.

src/react19/ErrorBoundary.tsx

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

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

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

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

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

**`getDerivedStateFromError`** is the method React calls when a component inside fails while rendering; the state it returns makes **`render`** show the fallback. In a real project you can also use the **`react-error-boundary`** library, which the React docs mention as an alternative. An error boundary does not catch errors from event handlers or from asynchronous code React does not see, such as a **`setTimeout`** or a stray **`.then`**: you handle those yourself, with **`.catch`** or with **`try`**/**`catch`** around an **`await`**.

> – Going deeper

> part 2's React Router **`errorElement`** is an error boundary the router sets up for you on the routes where you declare it.

## useOptimistic

When you mark a favorite, waiting for the server's response before showing the star makes the app feel slow. An **optimistic update** shows the result before the server confirms it, and undoes it if it fails. **`useOptimistic`** handles it:

src/react19/OptimisticFavorite.tsx

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

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

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

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

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

- **`useOptimistic(favorite)`** returns **`shown`**, which equals **`favorite`** except while an Action is in progress, and **`setShown`**, which changes **`shown`** only during that Action.

- **setShown** has to be called inside a transition or a React Action; that is why everything goes in **startTransition**, which accepts async functions.
- If **save** succeeds, **setFavorite** stores the real value, and **shown** matches it. It goes in another **startTransition** because it comes after an **await**, as in the React docs' examples.
- If **save** fails, the transition ends without **favorite** having changed, and React shows the real value again: the star turns off on its own, with no code to undo it (React, *useOptimistic*). The example's test checks it with a **save** that fails.

## ViewTransition

A **View Transition** is a browser animation between two states of the page. With **<ViewTransition>**, stable since React 19.3 (React, *React 19.3*), React triggers it for you:

src/react19/CardToDetail.tsx

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

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

It only animates updates inside a transition (**startTransition**), a **Suspense**, or a **useDeferredValue**; a plain **set** changes the screen without animating (React, *ViewTransition*). With no other options, the animation is a cross-fade. In the automated tests, which run in a simulated DOM with no browser view transition API, the change happens anyway, without animation.

> *"In Dex 506: the detail page's image has the same **name** as its card's image in the Pokédex. When one leaves and the other enters in the same transition, the browser animates the image from one place to the other:"*

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

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

## Activity

Go back to the tabs from "Performance": **`DetailTabs`** shows one panel or the other with a ternary, so switching tabs unmounts the other panel and the filter loses what you typed. **`<Activity>`**, which arrived in React 19.2 (React, *React 19.2*), hides without unmounting:

src/react19/ActivityTabs.tsx

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

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

With **`mode="hidden"`**, React hides the content with **`display: none`** and keeps its state and its DOM; it also unmounts its effects, and mounts them again when shown (React, *Activity*). The example's test types "hoja" in the filter, switches tabs, comes back, and the text is still there; with **`DetailTabs`**, the same test finds the field empty.

## Document metadata

In part 2 you changed the tab title with an effect. In React 19, **`<title>`** and **`<meta>`** can be written inside any component, and React places them in the document's **`<head>`** (React, *title*; React, *meta*):

src/react19/PokemonHead.tsx

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

**`<title>`** takes a single string: that is why it uses a template literal and not **`{name} · Mini Pokédex`**, which would be two children. And there should be only one **`<title>`** on screen at a time.

## Testing

In this section you will test your code at three levels, with the same tools that checked every example in this series.

### The testing pyramid

An automated test is code that uses your code and checks that it does what it should; it runs in seconds, on every change, and warns you before the person using the app notices.

Tests come in three levels, from the base to the top:

- A **unit test** tests one piece on its own, such as a component or a function, with no network or server. It is fast and points at exactly what failed.
- An **integration test** tests several pieces together: a component with its data functions, axios, and a simulated network.
- An **end-to-end test** (**e2e**) opens the real app in a real browser and uses it like a person. It gives the most confidence and is the slowest.

It is called a **pyramid** because you want many tests at the base and few at the top: unit tests are cheap, and e2e tests are expensive to run and to maintain.

### The tools

- **Vitest** runs the tests. It uses the same configuration system as Vite, so it understands TypeScript and JSX with no extra steps. If a **`vitest.config.ts`** exists, it uses that instead of **`vite.config.ts`**: that is why it repeats the **`react()`** plugin.
- **jsdom** simulates the browser's DOM inside Node, so tests can render components.

- **Testing Library** renders components and queries them the way a person would: by role, label, or text. **`user-event`**, from the same family, simulates key presses and clicks.

They are installed as development dependencies:

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

Vitest's configuration goes in its own file, at the project root:

vitest.config.ts

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

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

**`environment: 'jsdom'`** turns on the simulated DOM, **`testTimeout`** gives each test up to 15 seconds (the first run is slower), and **`setupFiles`** points at a file that runs before the tests. It imports MSW's **`server`**, which you build in "MSW: the simulated network" below; until then, leave out the lines that use it:

src/test/setup.ts

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

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

**`@testing-library/jest-dom/vitest`** adds checks such as **`toBeInTheDocument`** or **`toHaveFocus`**; **`beforeAll`**, **`afterEach`**, and **`afterAll`** run something before all the tests in the file, after each one, and at the end; **`cleanup`** unmounts whatever each test left behind.

To run them, **`npx vitest`** runs them and keeps watching for changes, and **`npx vitest run`** runs them once, as **continuous integration** (*CI*) does: the server that runs the tests on every change you push. Vitest finds **`*.test.*`** and **`*.spec.*`** files (**`.ts`**, **`.tsx`**, **`.js`**...) on its own.

## A unit test

A Pokémon's card:

src/testing/PokemonCard.tsx

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

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

src/testing/PokemonCard.test.tsx

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

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

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

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

- **`it`** declares a test with a sentence that says what it checks.

- **`userEvent.setup()`** prepares a simulated user, the recommended way to use **`user-event`**; **`user.click`** and **`user.type`** click and type like a person.
- **`render`** mounts the component in the simulated DOM, and **`vi.fn()`** creates a fake function that remembers how it was called. **`screen`** is the whole screen, where queries start from.
- **`screen.getByRole('button', { name: 'Marcar como favorito' })`** finds a button by its **accessible name**, the one a screen reader would read. Testing Library recommends querying by role first: if you cannot find an element by its role, your interface may not be accessible (Testing Library, *About Queries*).
- **`getBy…`** queries look for the element right now and fail if it is not there; **`findBy…`** queries wait for it to appear, which is why they return a promise and go with **`await`**. **`getByLabelText`** finds a field by its label's text.
- **`expect(…)`** checks a result: if it does not hold, the test fails with a message.

## Testing a form

The form from "Refs and the DOM", tested the way a person would use it:

src/testing/TrainerForm.test.tsx

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

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

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

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

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

The test does not know there is a ref or a **`useId`**: it checks what you see (the message tied to the field, and the focus) and what happens (**`onSave`** is not called until the name is valid). That is why it keeps passing if you change how the form works inside.

## MSW: the simulated network

**MSW** (*Mock Service Worker*) intercepts your code's network requests and answers with whatever you define, without changing the app's code: axios makes its request as always, and MSW answers it. **Handlers** say what to answer for each address:

src/test/handlers.ts

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

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

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

src/test/server.ts

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

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

An **origin** is the combination of an address's protocol, domain, and port: **`https://pokeapi.co`** and your app's page are different origins. The browser (and jsdom) only lets your code read a response from another origin if that server allows it with the **`Access-Control-Allow-Origin`** header (a **header** is an extra piece of data that travels with a request or a response): that is **CORS** (*Cross-Origin Resource Sharing*). That is why the PokéAPI handler includes it, while the **`/api`** ones you will see below, which go to the page's own origin, do not need it. **`/^\d+$/`** is a regular expression that matches text made only of digits: that way the handler accepts a number or a name, like the real PokéAPI.

**`setupServer`** builds the simulated server for Node. In **`setup.ts`**, **`onUnhandledFrame: 'error'`** fails any request that has no handler: that way no test calls a real API by accident.

An integration test: a form that adds a member to the team, and a server that rejects it with a **422**, the HTTP code for "I understood the request, but the data is not valid":

src/testing/teamApi.ts

```ts
import axios from 'axios'

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

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

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

src/testing/AddMember.tsx

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

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

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

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

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

src/testing/AddMember.test.tsx

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

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

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

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

**`server.use`** adds a handler for this test only, and **`setup.ts`**'s **`resetHandlers`** removes it at the end. The test covers the whole path: the component, **`addMember`**, axios, the simulated network, **`errorMessage`**, and the message on screen.

## End-to-end tests with Playwright

**Playwright** opens a real browser, starts the app, and drives it with the same queries by role. This Dex 506 test goes through the app end to end: it searches, opens a detail page, marks a favorite, builds a team, and wins a gym.

e2e/journey.spec.ts

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

  await page.getByRole('link', { name: 'Pokédex' }).click()
  await page.getByRole('searchbox').fill('lag')
```

```
  await expect(page.getByText('1 Pokémon')).toBeVisible()
  await expect(page).toHaveURL(/\?q=lag/)

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

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

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

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

**`signIn`** and **`addMember`** are functions in the same file that sign in and add a team member. In part 4 you write them, install Playwright, and set up its configuration, which starts the API and the web app before the tests. Dex 506's six e2e tests pass in Chromium, with the API running with made-up creatures (it is built with **Express**, a Node framework for web servers, and you build it in part 4) and the web app on the Vite development server.

## Common mistakes

> The usual mistake: a test with **`use`** and a promise stays on "Cargando…" forever, even though the promise settles. In a browser the same component works. Wrap the action that makes the component suspend in **`await act(async () => …)`**; **`act`** is the React function that groups a test's updates and processes them before moving on:

src/react19/PickPokemon.test.tsx

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

it('use lee la promesa y Suspense muestra la espera', async () => {
  const user = userEvent.setup()
  render(<PickPokemon />)
  await act(async () => {
```

```
    await user.click(screen.getByRole('button', { name: 'Ver el Pokémon número
3' }))
    expect(screen.getByText('Cargando…')).toBeInTheDocument()
  })
  expect(await screen.findByRole('heading', { name: '#3
lagunix' })).toBeInTheDocument()
})
```

> The usual mistake: an MSW handler that answers something different on every request, when the real API always answers the same. The test passes because of the handler, not your code, and it hides bugs the real API does show. Make handlers answer like the real API.

> The usual mistake: with Playwright, **`getByRole('button', { name: 'Escara' })`** also finds Dex 506's "Agregar a escarabin" (add escarabin) button: by default, **`name`** matches part of the text and ignores case. Use **`exact: true`** when one element's name contains another's.

## Architecture, Quality, and Security

In this section you will see what separates an exercise from a professional application, with Dex 506 as the example.

### Organizing by feature

The folders of Dex 506's web app:

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

- **app** puts the application together: the routes, the providers, and the Redux store.
- **features** has one folder per **feature**: login, the Pokédex, the detail page, the team, and the gyms. Each keeps its components, hooks, queries, and tests together.
- **shared** holds what several features use: the axios client, generic hooks, the theme, and UI components.
- **i18n** holds the interface text in each language.

Organizing by feature, and not by file type (every component in one folder, every hook in another), means a change to the team touches one folder and not five.

## Layers and contracts

Inside each feature, the code splits into three layers:

- the **interface**: the components, which only show things and report events;
- the **state**: Context, Zustand, or Redux, following part 2's "Not all state is the same" table;
- the **data**: the TanStack Query queries and the axios client.

The Express API and the web app share a package, **packages/contracts**, with the Zod schemas for each piece of data. A **contract** is that shared definition of what gets sent and received: if the API changes the shape of a piece of data, the web app stops compiling instead of failing in production. And since the API and the web app can end up on different versions between two deployments, the web app also validates every response it receives with **parse**. This is the contract for an error:

packages/contracts/src/schemas.ts

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

Every API error has that shape, which is why the web app always shows them the same way: the **errorMessage** from "Testing" reads exactly **error.message**.

> – **Going deeper**
>
> Dex 506 uses four state mechanisms (TanStack Query, Context, Zustand, and Redux Toolkit) so you get to know all four. In an app this size, Redux is more than you need: Zustand is enough for the gym

> lineup. Redux pays off in a large app, with many people working on the same state, where having every change go through a named action and be recorded is worth its cost.

## Accessibility

An accessible app can be used with a keyboard, with a screen reader, and with low vision. Three rules cover most cases:

- **Semantics:** use the element that fits: **`<button>`** for actions, **`<a>`** to navigate, a **`<label>`** for each field. A **`<div>`** with **`onClick`** cannot be focused with the keyboard, and a screen reader does not announce it as a button.
- **Keyboard:** everything you can do with the mouse has to be doable with the keyboard, and the focus has to be visible.
- **Contrast:** normal text needs a contrast ratio of at least 4.5:1 against its background (WCAG, the W3C's web accessibility guidelines, level AA).

Tests by role help with the first one, though they do not guarantee it: a **`<div role="button">`** also shows up as a button. On top of that, Dex 506 passes an automated accessibility audit in both of its themes; that audit found a coral text with a 4.21:1 contrast in the dark theme, which was fixed.

## Security: XSS

An **XSS** (*cross-site scripting*) attack gets your page to run someone else's script, for example one hidden in a user's bio. React protects you by default: whatever you put between curly braces is shown as text, never as HTML. Inside JSX, the most direct way around that protection has a name that warns you (outside JSX, writing **`innerHTML`** by hand through a ref carries the same risk):

BEFORE

src/architecture/Bio.tsx

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

AFTER

src/architecture/Bio.tsx

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

With a bio that contains **`<img src="x" onerror="alert(1)">`**, the "before" version really creates that **`<img>`**, and in a browser the **`onerror`** would run. The "after" version shows the text as is. The example's tests check both cases. If you ever need to show HTML from a user, clean it first with a library made for that, such as DOMPurify.

## Security: where the session lives

A **token** is a secret string that identifies the session. A common way to handle it is to store the token in **`sessionStorage`**, which is like part 2's **`localStorage`** but cleared when the tab closes, and send it with every request in the **`Authorization: Bearer …`** header, the standard way to send it:

BEFORE

src/architecture/session.ts

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

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

AFTER

apps/api/src/routes/auth.ts

```
router.post('/login', async (req, res) => {
  const { email, password } = LoginInputSchema.parse(req.body)
  const user = store.read().users.find((candidate) => candidate.email === email.toLowerCase())
  // The hash is checked even for an unknown email: the response time does not reveal which
  // accounts exist.
  const valid = await verifyPassword(password, user?.passwordHash ?? FAKE_HASH)
  if (!user || !valid) {
    throw new HttpError(401, 'INVALID_CREDENTIALS', 'Correo o contraseña incorrectos')
  }
  const token = randomBytes(32).toString('hex')
  await store.update((data) => {
    // Expired sessions are cleaned up on each login, so the file does not grow forever.
    for (const [old, session] of Object.entries(data.sessions)) {
      if (session.expiresAt <= Date.now()) delete data.sessions[old]
    }
    data.sessions[token] = { userId: user.id, expiresAt: Date.now() + SESSION_MS }
  })
  // httpOnly: the page's JavaScript cannot read it, so an injected script cannot steal it.
  res.cookie(SESSION_COOKIE, token, {
    httpOnly: true,
    sameSite: 'lax',
    secure: options.secureCookies,
```

```
    path: '/',
    maxAge: SESSION_MS,
  })
  res.json({ id: user.id, email:
user.email, trainerName: user.trainerName
} satisfies User)
})
```

The problem with the "before" version: any script running on your page can read **`sessionStorage`**. One XSS, and the attacker walks away with the session. The "after" version is code from Dex 506's Express API, which you write in part 4; what matters is **`res.cookie(…)`**: the API keeps the session in an **`httpOnly` cookie**, which the browser sends on its own with every request and which the page's JavaScript cannot read. The web app stores nothing. An XSS could still make requests on the user's behalf while the page is open, but it can no longer steal the session.

- **`sameSite: 'lax'`** keeps the browser from sending the cookie on requests that come from another site, except when someone navigates to your site with a **`GET`**, for example from a link: a **`POST`** form, a **`fetch`**, or an image from another site goes out without it. That stops **CSRF** (*cross-site request forgery*), the attack where another page makes requests to your API with its visitor's session, as long as a **`GET`** request never changes data.
- **`secure`** makes the cookie travel only over HTTPS.
- So that the cookie can travel without any CORS setup, the web app and the API are served from the same origin: in development, Vite's proxy, a go-between that forwards requests, sends **`/api`** to Express, as the configuration in "Performance" shows.

## Environment variables

Vite exposes environment variables that start with **`VITE_`** to browser code:

src/architecture/config.ts

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

**`import.meta.env`** is the object where Vite puts those variables. For TypeScript to know about the variable, it is declared in **`src/vite-env.d.ts`**, a file you create in **`src`**:

src/vite-env.d.ts

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

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

The value comes from a **`.env`** file at the project root (a **`VITE_MAPS_KEY=…`** line) or from the command line. "Exposing" means copying that value into the JavaScript every visitor downloads, as long as the code uses it: in the example, **`App.tsx`** reads **`config.mapsKey`** to decide whether to show the map (if nothing used it, Vite would leave it out of the build). To check, build with the variable and search the generated files for the text:

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

The output:

```
clave-de-ejemplo
```

There it is. A **`VITE_`** variable is not secret: a private API key belongs on the server, never in a **`VITE_`** variable (Vite, *Env Variables and Modes*).

## Internationalization

**Internationalization** (**i18n**) is preparing the app for several languages. A simple way, the one Dex 506 uses, is one dictionary per language, with functions for text that carries data, and a Context that provides the active language's dictionary:

src/architecture/i18n.tsx

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

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

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

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

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

export function useI18n(): I18n {
  const i18n = use(I18nContext)
```

```
  if (!i18n) throw new Error('useI18n tiene que usarse dentro de I18nProvider')
  return i18n
}
```

src/architecture/Greeting.tsx

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

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

**`keyof typeof dictionaries`** is the union of the object's keys, **`'es' | 'en'`**. The provider's **`value`** goes through **`useMemo`**: that way it is the same object as long as the language does not change, and it does not re-render the components that read the Context. To use it, **`Greeting`** has to sit inside **`<I18nProvider>`**. Components contain no hard-coded text: they read **`t.greeting(name)`** or **`t.signOut`** from the active language. The exception is language names, which always appear in their own language ("English", "Español"). Dex 506 does the same with one dictionary per language in **`apps/web/src/i18n`**. For an app with many languages, plurals, and date and number formats, there are libraries such as react-i18next or FormatJS.

## The Bigger Picture and the Road to Senior

In this section you will see where an SPA like Dex 506 fits in the React ecosystem.

### Rendering on the server

Dex 506 is an SPA: the server sends an almost empty HTML page, and the browser downloads the JavaScript and builds the screen. There are other ways:

- **SSR** (*server-side rendering*): the server runs the components and sends the HTML already built. The page shows up sooner and search engines read it in full; then React makes it interactive in the browser.
- **Server Components** are components rendered before they reach the browser, at build time or on the server for each request, and whose code is never downloaded. They cannot use state or effects;

for interactivity they are combined with client components, marked with `'use client'` (React, *Server Components*).

- **Server Functions** are async functions that run on the server and that a client component calls as if they were local, marked with `'use server'`. Until September 2024 they were all called Server Actions; today that name is kept for the ones used as an Action (React, *Server Functions*).

All of that requires a **framework**: a tool that, on top of React, takes care of the server, routing, and the build. The React docs recommend starting a new app with one, such as Next.js or React Router in framework mode (React, *Creating a React App*). React Router has three modes: declarative, the data mode you used in part 2, and framework mode, which adds SSR, typed routes, and automatic code splitting (React Router, *Picking a Mode*). This very site is built with Next.js.

> **Watch out:** a Server Component or a Server Function is code that runs on your server, with risks of its own. In December 2025, React published an advisory about a critical vulnerability in Server Components (React, *Critical Security Vulnerability in React Server Components*). A framework brings server responsibilities that an SPA does not have.

### When an SPA and when a framework

An SPA like Dex 506 makes sense when the app lives behind a login, search engines do not need to read its content, and you want to serve it as static files, with no rendering server. A framework makes sense when the first load and search ranking matter, as in a store or a blog, or when you want part of the code to never reach the browser.

### What to study next

- **A framework**: Next.js or React Router in framework mode, to see SSR and Server Components in practice.
- **Accessibility in depth**: the WCAG guidelines, and testing your app with a screen reader.
- **Web performance**: the browser's loading metrics and how to measure them in production.
- **React Native with Expo**, if you want to take what you know to mobile apps.

## What You Learned

React renders in two phases, render and commit, and a component renders when its state changes, when its parent renders, or when a Context it reads changes. State lives at a position in the tree, and the **`key`** resets it; React compares objects and functions by identity. A well-designed custom hook does one thing and takes what it needs. Refs keep values without rendering and reach the DOM. Generics and discriminated unions make impossible states fail to compile. For performance, measure first: **`memo`** skips renders, **`useMemo`** and **`useCallback`** keep the identity **`memo`** needs, and React Compiler applies them for you. React 19 brings **`use`**, **`useOptimistic`**, **`<ViewTransition>`**, **`<Activity>`**, and metadata in components. And a

professional app is tested at three levels, organized by feature, and built with accessibility and security in mind from the start.

## What's Next

In part 4: Build a React + TypeScript App from Zero to Production Build you put it all together: you build Dex 506 from an empty folder to a production build, with its Node API, step by step, with each step's code available to download.

## Bibliography


- React, Profiler. Accessed November 18, 2026.
- React, Component, on error boundaries. Accessed November 18, 2026.
- The React Team, React v19, December 5, 2024. Accessed November 18, 2026.
- The React Team, React 19.2, October 1, 2025. Accessed November 18, 2026.
- The React Team, React 19.3, September 9, 2026. Accessed November 18, 2026.
- Lauren Tan, Joe Savona, and Mofei Zhang, React Compiler v1.0, October 7, 2025. Accessed November 18, 2026.
- React, useOptimistic, ViewTransition, Activity, title, and meta. Accessed November 18, 2026.
- React, Server Components, Server Functions, and Creating a React App. Accessed November 18, 2026.
- React, Critical Security Vulnerability in React Server Components, December 3, 2025. Accessed November 18, 2026.
- React Router, Picking a Mode. Accessed November 18, 2026.
- Testing Library, About Queries. Accessed November 18, 2026.
- Vite, Env Variables and Modes. Accessed November 18, 2026.