React desde cero: de JavaScript a tus primeros componentes

Creado por Manuel Herrera Mesén · 24 de noviembre de 2026

Parte 1 de 4 · react · typescript · javascript · frontend · serie-react

EN ESTE ARTÍCULO

- Antes de empezar
- Qué es React y por qué existe
- El JavaScript que React exige
- Tu primer proyecto con Vite
- Componentes y JSX
- Props y children
- Por qué nace TypeScript
- Condicionales y listas
- Estado y eventos
- Elevar el estado y useReducer
- Estilos
- Si vienes de Vue o Angular
- Lo que aprendiste
- Qué sigue
- Bibliografía

> Parte 1 de 4 de la serie «Aprende React construyendo una app real».

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

Abres una tienda en línea, escribes «zapatillas» en el buscador y la lista cambia mientras escribes, sin recargar la página. Marcas un favorito y el corazón se pinta al instante. Detrás de algo tan común hay un problema que durante años se resolvió a mano: mantener lo que ves en pantalla de acuerdo con los datos.

React nació para ese problema. En esta serie lo aprendes construyendo **Dex 506**, la Pokédex de un entrenador: explorar los primeros 151 Pokémon, abrir su ficha, armar un equipo de seis y retar a ocho gimnasios inventados en Costa Rica. Me gusta entender el recorrido completo de una aplicación, del servidor a la pantalla, y en esta serie lo recorremos juntos, con React como la pieza de la interfaz.

Cómo leer esta guía. Si nunca usaste React ni TypeScript, lee el texto principal: va paso a paso y no da nada por sabido, salvo JavaScript básico. Los recuadros **Para profundizar** salen cerrados: algunos repasan JavaScript que vas a necesitar (ábrelos si el tema no te resulta familiar) y otros tienen el detalle fino para quien ya programa con otro framework. Al final hay una tabla para quien viene de Vue o Angular.

Todo el código lo ejecuté con React 19.3.1, TypeScript 7.0.2, Vite 8.3.8 y Tailwind CSS 4.3.5: cada ejemplo tiene pruebas automáticas que pasan, y los mensajes de error que ves son los que dio la herramienta, copiados de su salida.

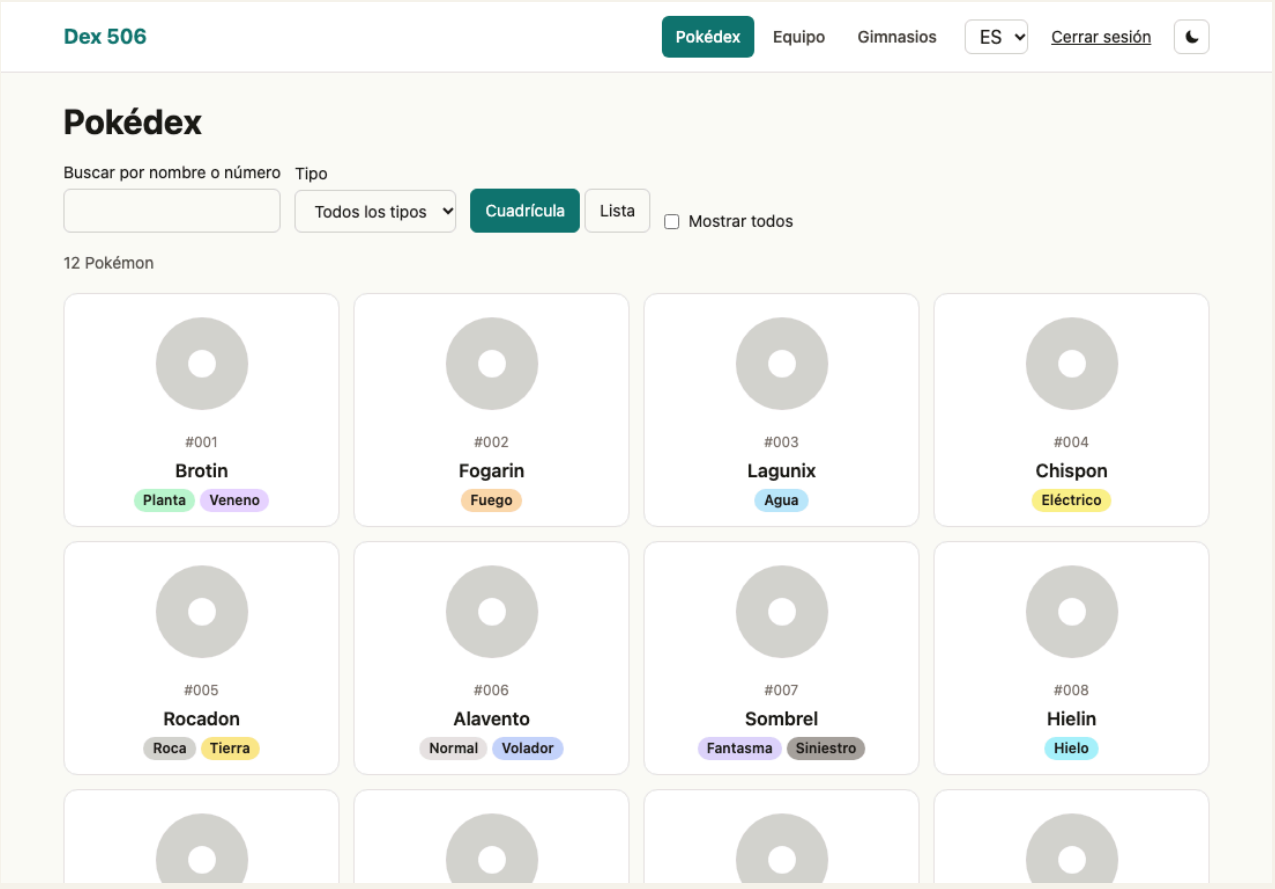
Al terminar esta parte podrás:

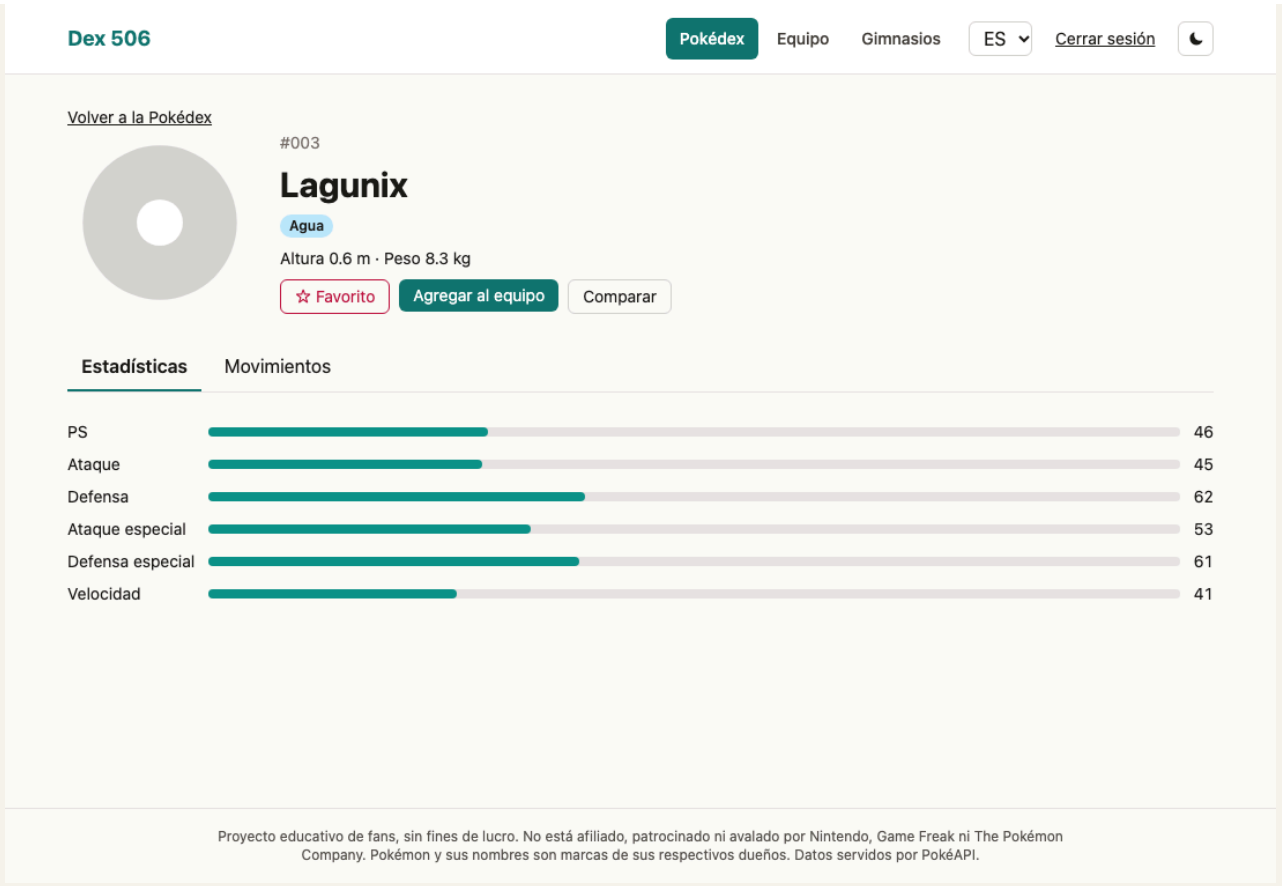
1. explicar qué problema resuelve React;
2. crear un proyecto de React con TypeScript usando Vite;
3. explicar por qué existe TypeScript y escribir los tipos de las props de un componente;
4. construir componentes con props, estado, eventos, listas y estilos;
5. armar una mini Pokédex con búsqueda por nombre y filtro por tipo.

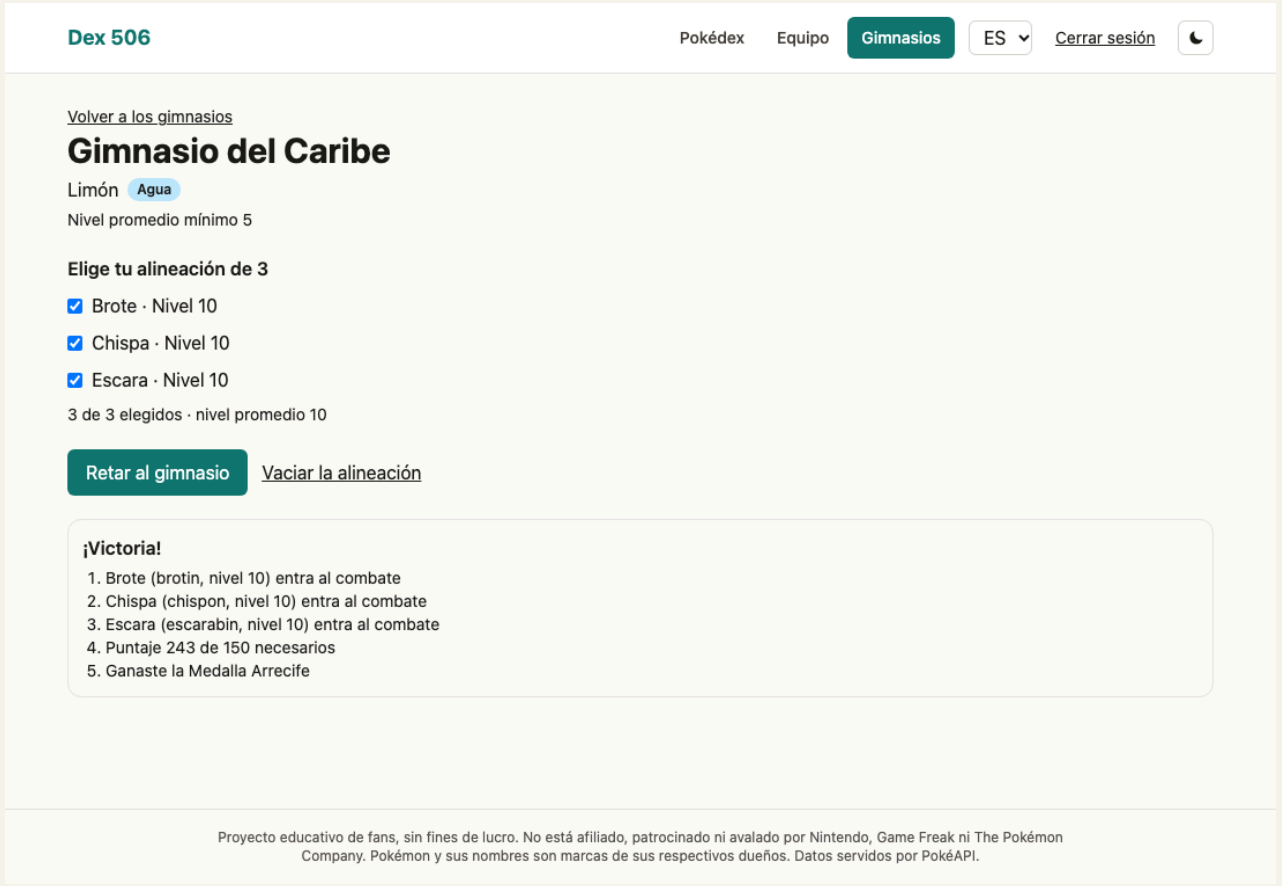
Antes de empezar

En esta sección vas a ver adónde llega la serie y cómo usar los ejemplos.

Esto es lo que vas a construir en la parte 4: Dex 506 completa, con su Pokédex, la ficha de cada Pokémon, tu equipo y los gimnasios. Las capturas usan criaturas inventadas, las mismas con las que se prueba la app; con los datos reales de PokéAPI (una API pública y gratuita con los datos de los Pokémon), cada tarjeta muestra su Pokémon. En esta parte 1 trabajas con seis criaturas inventadas, que vas a ver en el primer ejemplo de datos.







La serie tiene cuatro partes, y cada una exige la anterior:

Parte	Nivel	Qué aprendes
1	Desde cero	Componentes, props, TypeScript, estado, eventos, listas y estilos
2	Intermedio	Efectos, datos del servidor, rutas, Context, formularios y estado global
3	Avanzado	Cómo renderiza React, hooks propios, rendimiento, React 19, pruebas y arquitectura
4	Práctica	Dex 506 de la carpeta vacía al build de producción, con su API en Node

Esta serie no enseña JavaScript desde cero. Si todavía no lo manejas, empieza por la guía de JavaScript de MDN y vuelve cuando escribas funciones, arreglos y objetos sin pensarlo.

Los ejemplos de esta parte están completos en el texto, con sus **`import`**: puedes copiarlos en el proyecto que creas en la sección «Tu primer proyecto con Vite». El título de cada bloque de código dice en qué carpeta va el archivo, como **`src/data/pokemon.ts`**. El código de la app de la parte 4 se descarga paso a paso desde el artículo de la parte 4.

Qué es React y por qué existe

En esta sección vas a ver qué problema resuelve React y la idea que lo hace distinto.

El problema: la pantalla y los datos se separan

Una página web es un árbol de elementos: un título, una lista, un botón. El navegador lo representa con el **DOM** (*Document Object Model*): el árbol de objetos que JavaScript puede leer y cambiar para modificar lo que ves. Para tocarlo, el navegador trae funciones como **`querySelector`** (busca un elemento), **`addEventListener`** (reacciona a un clic) y la propiedad **`textContent`** (el texto de un elemento).

Durante años, mostrar un cambio fue tarea tuya: decidir qué elemento tocar y escribirle el valor nuevo. Este contador de favoritos, en JavaScript sin React, lo hace así:

ANTES

src/intro/counter-dom.js

```
export function mountCounter(root) {
  root.innerHTML = '<p>Favoritos: <span
class="count">0</span></p>
<button>+1</button>'
  let count = 0
  const label =
root.querySelector('.count')

root.querySelector('button').addEventList
ener('click', () => {
    count = count + 1
    label.textContent = String(count)
```

DESPUÉS

src/intro/Counter.tsx

```
  })
}
```

```
import { useState } from 'react'

export function Counter() {
  const [count, setCount] = useState(0)
  return (
    <>
      <p>Favoritos: {count}</p>
      <button onClick={() =>
setCount(count + 1)}>+1</button>
    </>
  )
}
```

El primer bloque es **imperativo**: le dice al navegador cada paso. Busca el texto, suma uno, escribe el número. Con un contador, funciona. Con una aplicación entera, cada dato que cambia puede afectar a varios lugares de la pantalla, y es tu trabajo acordarte de todos.

El segundo es **declarativo**: describe cómo se ve la pantalla para un valor de **`count`**, y React se encarga de que el DOM coincida. No hace falta que entiendas todavía su sintaxis: el **`useState`** lo vas a escribir tú en la sección «Estado y eventos», y las etiquetas dentro de JavaScript, en «Componentes y JSX». Lo que importa ahora es la diferencia: no hay ningún **`textContent`**.

Cómo nació

React nació dentro de Facebook. Pete Hunt, del equipo de React, lo resumió así: en una aplicación de JavaScript tradicional, tienes que ver qué datos cambiaron y modificar el DOM de forma imperativa; React brilla cuando los datos cambian con el tiempo (Pete Hunt, *Why did we build React?*). Facebook publicó React como código abierto el 29 de mayo de 2013 (repositorio **`facebook/react`**, commit «Initial public release»).

Las palabras que vas a usar

React es una **biblioteca** de JavaScript para construir interfaces. Una biblioteca es código que tú llamas cuando lo necesitas. Un **framework**, en cambio, define la estructura de tu aplicación y es el que llama a tu código. React se ocupa de la interfaz y deja lo demás (navegación, datos) a otras bibliotecas; en la parte 2 vas a elegirlas.

Un **componente** es una pieza de la interfaz con nombre propio, como **`Counter`**. Técnicamente, es una función de JavaScript que devuelve la descripción de lo que se ve. Una aplicación de React es un árbol de componentes: una página contiene una lista, la lista contiene tarjetas.

Renderizar es lo que hace React cuando llama a tu componente para saber qué mostrar. Lo hace cuando cambia el estado del componente (lo vas a ver más adelante) o cuando se vuelve a renderizar el componente que lo usa, que así le pasa sus datos nuevos. Compara el resultado nuevo con el anterior, y

ese proceso de comparar se llama **reconciliación**; después aplica en el DOM solo lo que cambió, un paso que se llama **commit**.

Por ahora, piénsalo así: la interfaz es una función de los datos, en el sentido de las matemáticas: para los mismos datos, la misma pantalla. En la parte 3 vas a ver qué hace React por dentro y por qué esa frase es una buena aproximación, no toda la historia.

El JavaScript que React exige

En esta sección vas a repasar las partes de JavaScript que React usa a cada rato.

React se escribe en JavaScript. Lo único que agrega es JSX, una forma de escribir etiquetas dentro del código que vas a ver en «Componentes y JSX». Por eso, lo que en otros frameworks hace una **directiva** de la plantilla (una instrucción especial dentro del HTML, como **`v-for`** en Vue), en React lo hace JavaScript común.

Todos los ejemplos de esta parte usan estas seis criaturas inventadas, cada una con un **`id`**, un nombre y sus tipos. Son los datos del archivo **`src/data/pokemon.ts`**, que vas a armar completo en «Por qué nace TypeScript», cuando veas cómo se describe su forma:

src/data/pokemon.ts

```
{ id: 1, name: 'Brotín', types: ['grass', 'poison'] },
{ id: 2, name: 'Fogarín', types: ['fire'] },
{ id: 3, name: 'Lagunix', types: ['water'] },
{ id: 4, name: 'Chispón', types: ['electric'] },
{ id: 5, name: 'Rocadón', types: ['rock', 'ground'] },
{ id: 6, name: 'Escarabín', types: ['bug', 'grass'] },
```

Si alguna de estas herramientas de JavaScript no te resulta familiar, abre su recuadro antes de seguir:

> – **Para profundizar**
>
> **`map`** y **`filter`**. Los dos recorren un arreglo y devuelven uno nuevo, sin cambiar el original: **`filter`** se queda con los elementos que cumplen una condición, y **`map`** transforma cada uno. En React vas a armar listas así.
>
> src/js/transform.ts
>
> ```
> import { creatures } from '../data/pokemon'
>
> const grassNames = creatures
>   .filter((creature) => creature.types.includes('grass'))
>   .map((creature) => creature.name)
> ```

Con las criaturas de arriba, **`grassNames`** es **`['Brotín', 'Escarabín']`**: las dos que tienen el tipo **`grass`**.

– Para profundizar

Funciones flecha, desestructuración y spread. Una **función flecha** (**`(x) => x + 1`**) es una forma corta de escribir una función. La **desestructuración** saca propiedades de un objeto en variables, y el **spread** (**`...`**) copia las propiedades de un objeto en otro nuevo.

src/js/transform.ts

```
const lagunix = { id: 3, name: 'Lagunix', types: ['water'] }
const renamed = { ...lagunix, name: 'Lagu' }
const { name, types } = renamed
```

`renamed` es un objeto nuevo con el nombre cambiado; **`lagunix`** no se tocó. Crear datos nuevos en lugar de modificar los que existen se llama **inmutabilidad**, y React la necesita para saber qué cambió. Un detalle: el spread copia solo el primer nivel. **`renamed.types`** y **`lagunix.types`** son el mismo arreglo.

La desestructuración también funciona con arreglos: toma los elementos por posición. Vas a usarla en «Estado y eventos».

src/js/transform.ts

```
const [first, second] = ['Brotín', 'Fogarín']
```

– Para profundizar

Módulos. Cada archivo es un **módulo**: lo que quiere compartir lo marca con **`export`**, y otro archivo lo trae con **`import { nombre } from './archivo'`**. Un archivo también puede tener una **exportación por defecto** (**`export default`**), que se importa sin llaves: **`import App from './App'`**. Y **`import './index.css'`**, sin nombre, solo carga ese archivo.

– Para profundizar

Callbacks, promesas y `async`/`await`. Un **callback** es una función que le pasas a otra para que la llame: la que pasas a **`map`** es un callback. Pedir datos a un servidor tarda, y JavaScript no espera

parado: sigue con lo demás y te avisa después, llamando a un callback. Una **promesa** es un objeto que representa ese resultado futuro. Con **`async`/`await`**, el código que espera una promesa se lee de arriba a abajo. Las tres versiones piden lo mismo a un servidor de mentira:

src/js/fetching.js

```
import { creatures } from '../data/pokemon.ts'

// Un «servidor» de mentira: responde después de 10 ms, como lo haría uno real tras la red.
export function findWithCallback(id, done) {
  setTimeout(() => done(creatures.find((creature) => creature.id === id)), 10)
}
```

src/js/fetching.js

```
export function find(id) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(creatures.find((creature) => creature.id === id)), 10)
  })
}
```

src/js/fetching.js

```
export async function describeCreature(id) {
  const creature = await find(id)
  return creature ? `${creature.name}: ${creature.types.join(' / ')}` : 'No existe'
}
```

Así se usan las tres, y esto imprime:

src/js/demo.js

```
import { describeCreature, find, findWithCallback } from './fetching.js'

findWithCallback(2, (creature) => console.log('callback:', creature.name))

find(3).then((creature) => console.log('promesa:', creature.name))

console.log('async:', await describeCreature(5))
```

```
callback: Fogarín
promesa: Lagunix
async: Rocadón: rock / ground
```

> Una función que recuerda las variables del lugar donde se creó, aunque se ejecute después, es un **closure**: el callback de **`setTimeout`** en **`findWithCallback`** recuerda **`id`** y **`done`**. Los manejadores de eventos de React, que vas a ver en «Estado y eventos», también lo son.

Tu primer proyecto con Vite

En esta sección vas a crear un proyecto de React y a recorrer sus archivos.

Por qué una herramienta para crear el proyecto

Un proyecto de React necesita varias piezas antes de la primera línea: traducir a JavaScript el JSX y el TypeScript (los vas a ver en las próximas secciones), juntar los archivos para el navegador y recargar la página mientras trabajas. Antes se configuraban a mano. Hoy, una herramienta crea el proyecto listo.

Un **bundler** junta tus archivos y sus dependencias (las bibliotecas que usa tu proyecto) en los pocos archivos que descarga el navegador. **Vite** es la herramienta que usa esta serie: crea el proyecto, levanta un **servidor de desarrollo** (un servidor local que muestra tu app mientras la escribes) y genera el **build de producción**, la versión final y optimizada que se publica.

Crear el proyecto

Vas a necesitar **Node.js**, el programa que ejecuta JavaScript fuera del navegador. Vite exige Node 20.19 o posterior, o 22.12 o posterior (Vite, *Getting Started*); esta serie usa Node 26, la versión LTS (de soporte extendido) actual. Con Node viene **npm**, el gestor de **paquetes**: los paquetes son las bibliotecas publicadas, y npm las descarga. En la terminal:

```
npm create vite@latest mi-pokedex -- --template react-ts
cd mi-pokedex
npm install
npm run dev
```

La primera línea ejecuta **`create-vite`**, el creador de proyectos de Vite, con la plantilla **`react-ts`** (React con TypeScript); el **`--`** de en medio hace que la opción **`--template`** le llegue a **`create-vite`** y no a npm. **`npm install`** descarga las dependencias, y **`npm run dev`** levanta el servidor de desarrollo. La terminal te muestra una dirección (**`http://localhost:5173`**); ábrela en el navegador.

Con el servidor andando, cambia un texto en **`src/App.tsx`** y guarda: la página se actualiza sin recargarse. Eso es **HMR** (*Hot Module Replacement*): Vite reemplaza solo el módulo que cambió.

Los archivos

Vas a ver archivos **`.tsx`** y **`.ts`**: son TypeScript, con JSX y sin JSX. Por ahora, léelos como JavaScript; en la sección «Por qué nace TypeScript» vas a ver por qué existen.

`index.html` es la página que carga el navegador. Tiene un **`div`** vacío con **`id="root"`** y carga **`src/main.tsx`**:

index.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>mi-pokedex</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

`src/main.tsx` es donde React toma ese **`div`**:

src/main.tsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'

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

- **`createRoot`** crea la raíz de React dentro del **`div#root`**, y **`render`** dibuja ahí el componente **`App`**.
- **`StrictMode`** es un componente de React que no se ve: en desarrollo hace controles extra para avisarte de errores comunes. En la parte 2 vas a ver uno de sus efectos visibles.
- El **`!`** después de **`getElementById(...)`** le asegura a TypeScript que el resultado no es **`null`**. No lo comprueba: si el elemento faltara, fallaría al ejecutar.
- **`App.tsx`** exporta por defecto el componente principal, **`App`**.

Para probar un ejemplo de esta guía, impórtalo en **`src/App.tsx`** y devuélvelo desde **`App`**. Con el contador de «Qué es React», **`App.tsx`** queda así:

src/App.tsx

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

export default function App() {
  return <Counter />
}
```

Componentes y JSX

En esta sección vas a escribir tu primer componente.

El problema: una interfaz entera en un solo lugar

Si escribes toda la interfaz en un solo archivo, cada cambio te obliga a leerlo entero. Los componentes la dividen en piezas con nombre propio, que se escriben y se prueban por separado.

Un componente de función

Un **componente de función** es una función de JavaScript que devuelve lo que se ve, y cuyo nombre empieza con mayúscula. La mayúscula no es estética: en JSX, **`<p>`** en minúscula es una etiqueta de HTML, y **`<StaticCard>`** en mayúscula es un componente tuyo. Esta es la tarjeta de un Pokémon, todavía con datos fijos:

src/jsx/StaticCard.tsx

```tsx
export function StaticCard() {
  return (
    <article className="pokemon-card">
      <h2>Lagunix</h2>
      <p>water</p>
    </article>
  )
}
```

Ese HTML dentro de JavaScript es **JSX**: una extensión de la sintaxis de JavaScript que se parece a HTML y que se traduce a llamadas de JavaScript antes de llegar al navegador. Para usar el componente, lo escribes como una etiqueta: **`<StaticCard />`**.

Tres reglas de JSX que vas a usar siempre:

- **Las llaves `{ }` insertan una expresión de JavaScript**: una variable, una suma, una llamada a una función. Todo lo que devuelve un valor. Un **`if`** no devuelve un valor, así que no va entre llaves; en «Condicionales y listas» vas a ver cómo se elige qué mostrar.

- **`className` en lugar de `class`**, como la propiedad `className` de los elementos del DOM (`class` es además una palabra reservada de JavaScript).
- **Cada bloque de JSX tiene un solo elemento raíz.** Si necesitas varios sin un contenedor extra, envuélvelos en un **Fragment**: `<>…</>`, una etiqueta vacía que no aparece en el DOM.

src/jsx/ExpressionCard.tsx

```
const creature = { id: 3, name: 'Lagunix', types: ['water'] }

export function ExpressionCard() {
  return (
    <article className="pokemon-card">
      <h2>{creature.name.toUpperCase()}</h2>
      <p>#{String(creature.id).padStart(3, '0')}</p>
    </article>
  )
}
```

Aquí las llaves muestran el nombre en mayúsculas y el número con tres cifras: **`#003`**.

Errores frecuentes

El error típico: devolver dos elementos sin un contenedor.

two-roots.tsx

```
export function TwoRoots() {
  return (
    <h2>Lagunix</h2>
    <p>water</p>
  )
}
```

```
two-roots.tsx(3,5): error TS2657: JSX expressions must have one parent element.
```

La corrección es envolverlos en un **`<article>`** o en un Fragment **`<>…</>`**.

El error típico: escribir **`class`**, como en HTML.

class-attribute.tsx

```
export function WithClass() {
  return <article class="pokemon-card">Lagunix</article>
}
```

```
class-attribute.tsx(2,19): error TS2322: Type '{ class: string; children: string;
}' is not assignable to type 'DetailedHTMLProps<HTMLAttributes<HTMLElement>,
HTMLElement>'.
  Property 'class' does not exist on type
'DetailedHTMLProps<HTMLAttributes<HTMLElement>, HTMLElement>'. Did you
mean 'className'?
```

El mensaje ya te dice la corrección: **`className`**. Lo demás es la descripción completa de los atributos que acepta una etiqueta de HTML en React; la palabra «children» es el texto de adentro de la etiqueta, y la vas a ver en la próxima sección.

Props y children

En esta sección vas a hacer que un componente reciba datos.

El problema: una tarjeta que siempre dice lo mismo

`StaticCard` siempre muestra a Lagunix. Para mostrar cualquier criatura, el componente tiene que recibir sus datos desde afuera.

Props

Las **props** son los datos que un componente recibe de quien lo usa. Técnicamente, React junta los atributos que escribes en la etiqueta (**`<TypeBadge type="grass" />`**) en un objeto y se lo pasa a la función como su primer argumento. Por eso se lee con desestructuración:

src/props/TypeBadge.tsx

```tsx
export function TypeBadge({ type }: { type: string }) {
  return <span className="type-badge">{type}</span>
}
```

El **`: { type: string }`** describe qué recibe: una prop **`type`** que es un texto. Escríbelo así por ahora. Sin esa descripción, el editor marca un error:

ANTES

untyped-props.tsx

```tsx
export function TypeBadge({ type }) {
  return <span className="type-badge">
{type}</span>
}
```

DESPUÉS

```
untyped-props.tsx(1,29): error TS7031:
Binding element 'type' implicitly has an
'any' type.
```

El mensaje dice que el nombre **`type`**, que sacaste con la desestructuración (un *binding element*), no tiene un tipo declarado. En la sección «Por qué nace TypeScript» vas a ver por qué eso es un error y cómo se escriben los tipos.

Las props son **de solo lectura**: un componente no cambia las que recibe. Si un dato tiene que cambiar, lo cambia quien se lo pasa; en «Estado y eventos» vas a ver cómo.

Una prop puede tener un **valor por defecto**, para cuando no te la pasan. El **`?`** marca que es opcional:

src/props/LevelBadge.tsx

```tsx
export function LevelBadge({ level = 1 }: { level?: number }) {
  return <span className="level">Nivel {level}</span>
}
```

`<LevelBadge />` muestra «Nivel 1».

children y composición

A veces un componente no sabe de antemano qué va a envolver, como una tarjeta que puede contener cualquier cosa. Para eso existe **`children`**: la prop especial con lo que escribes entre la etiqueta de apertura y la de cierre. Su tipo, **`ReactNode`**, significa «cualquier cosa que React pueda mostrar», y se importa de **`react`** con **`import type`**, la forma de importar solo un tipo.

src/props/Card.tsx

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

export function Card({ title, children }: { title: string; children: ReactNode }) {
  return (
    <section className="card">
      <h2>{title}</h2>
      {children}
    </section>
  )
}
```

Armar componentes grandes metiendo unos dentro de otros se llama **composición**:

src/props/Composition.tsx

```tsx
import { Card } from './Card'
import { TypeBadge } from './TypeBadge'

export function BrotinCard() {
  return (
    <Card title="Brotín">
```

```
      <TypeBadge type="grass" />
      <TypeBadge type="poison" />
    </Card>
  )
}
```

Card no sabe nada de tipos de Pokémon, y **TypeBadge** no sabe nada de tarjetas. Cada uno hace una cosa, y juntos arman la ficha.

Por qué nace TypeScript

Al terminar esta sección vas a poder leer los tipos de TypeScript que aparecen en el resto de la serie y entender sus mensajes de error.

El problema: un error que nadie avisa

En la sección anterior viste que un componente recibe datos por props. Imagina ahora una tarjeta que recibe el nombre (**name**) y los tipos (**types**) de una criatura, y un error de tipeo al usarla: escribes **nmae** en lugar de **name**.

src/App.jsx

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

export default function App() {
  return <PokemonCard nmae="Brotín" types={['grass', 'poison']} />
}
```

Mira qué pasa con la tarjeta en JavaScript y con la misma tarjeta en TypeScript, que agrega una descripción de las props que recibe. En la descripción, **string** es un texto y **string[]** es un arreglo de textos. (Para probar la versión en JavaScript, crea un proyecto con la plantilla **react** en lugar de **react-ts**.)

ANTES

src/PokemonCard.jsx

```
export function PokemonCard({ name, types
}) {
  return (
    <article className="pokemon-card">
      <h2>{name}</h2>
      <p>{types.join(' / ')}</p>
    </article>
  )
}
```

DESPUÉS

src/PokemonCard.tsx

```
type PokemonCardProps = {
  name: string
  types: string[]
}

export function PokemonCard({ name, types
}: PokemonCardProps) {
  return (
    <article className="pokemon-card">
      <h2>{name}</h2>
```

```
        <p>{types.join(' / ')}</p>
      </article>
    )
}
```

SALIDA

```
App.tsx(4,23): error TS2322: Type '{ nmae: string; types: string[]; }' is not assignable
to type 'IntrinsicAttributes & PokemonCardProps'.
  Property 'nmae' does not exist on type 'IntrinsicAttributes & PokemonCardProps'.
```

En JavaScript no aparece ningún error. La aplicación arranca, la tarjeta se muestra y el título queda vacío: como nadie le pasó **`name`**, su valor es **`undefined`**, y React no muestra nada en pantalla para **`undefined`**. Lo comprobé con una prueba automática del ejemplo: el título se renderiza sin texto. Lo descubre quien use la aplicación.

En TypeScript, el error aparece sin ejecutar nada. El mensaje de arriba es lo que imprime **`tsc`**, el compilador de TypeScript (un programa que traduce y revisa el código; más abajo lo ves en detalle), al revisar el archivo **`App.tsx`**: el mismo código de **`App.jsx`**, con la extensión de TypeScript. Se lee así:

- **`App.tsx(4,23)`** es el archivo, la línea y la columna: la línea 4, donde está **`nmae`**.
- **`TS2322`** es el código del error, útil para buscarlo.
- **`IntrinsicAttributes`** son los atributos que React acepta en cualquier componente, como **`key`**, que vas a ver en «Condicionales y listas». El **`&`** une ese tipo con **`PokemonCardProps`**.
- La segunda línea dice lo importante: **`nmae`** no existe entre las props de la tarjeta.

En **`PokemonCard.tsx`**, la anotación **`: PokemonCardProps`** describe el objeto de props completo, del que la función saca **`name`** y **`types`**.

> **Cuidado:** el servidor de desarrollo de Vite no revisa tipos: solo los quita para que el navegador pueda ejecutar el código (Vite, *Features*). Si ignoras el error, **`npm run dev`** arranca igual y ves la tarjeta sin nombre. Lo detectan tu editor, mientras escribes, y el **`npm run build`** de la plantilla, que corre **`tsc`** antes de generar el build. Trata cada línea roja como un error de verdad.

De dónde viene

No es un problema nuevo. El 1 de octubre de 2012, Microsoft presentó TypeScript con una meta concreta: que el desarrollo en JavaScript escalara a aplicaciones grandes (S. Somasegar, *TypeScript: JavaScript Development at Application Scale*).

En marzo de 2025 el equipo anunció que portaría el compilador y las herramientas del editor a Go, un lenguaje que se compila a código de máquina, para que fueran unas diez veces más rápidos (Anders Hejlsberg, *A 10x Faster TypeScript*). Esa versión salió el 8 de julio de 2026 como TypeScript 7 (Daniel

Rosenwasser, *Announcing TypeScript 7.0*), y es la que usa esta serie. Para ti, cambia la velocidad: los errores aparecen antes en el editor. El lenguaje que escribes es el mismo.

Los tipos, uno por uno

JavaScript tiene **tipado dinámico**: una variable puede guardar un número y después un texto, y nada se revisa antes de ejecutar. Un error de tipo, si se nota, se nota al ejecutar; y muchas veces no se nota, como el **`nmae`** de arriba. Con **tipado estático**, el tipo de cada variable, prop o parámetro se conoce antes de ejecutar, y un programa revisa que el código no lo contradiga.

Un **tipo** es la descripción de qué valores acepta una variable, una prop o un parámetro. Dicho con precisión, es un conjunto de valores posibles: **`number`** acepta **`5`** y **`3.14`**; **`string`**, cualquier texto.

Una **anotación de tipo** es ese tipo escrito a mano, después de dos puntos: en **`name: string`**, la anotación es **`string`**. No hace falta escribirla siempre. Con la **inferencia**, TypeScript deduce el tipo a partir del valor:

src/typescript/basics.ts

```
let level = 5
level = level + 1
```

`level` empezó siendo **`5`**, así que TypeScript infiere que es un **`number`**, y rechaza que más adelante reciba un texto:

ANTES

level-as-text.ts

```
let level = 5
level = 'five'
```

DESPUÉS

```
level-as-text.ts(2,1): error TS2322: Type
'string' is not assignable to type
'number'.
```

Para describir un objeto con nombre propio tienes dos formas, **`type`** e **`interface`**. Para las props de una tarjeta, las dos dicen lo mismo:

src/typescript/basics.ts

```
type CardProps = {
  name: string
  types: string[]
}

interface CardPropsInterface {
  name: string
  types: string[]
}
```

Hay diferencias finas entre las dos; están en el recuadro «Para profundizar» al final de esta sección. En esta serie usamos **`type`**, porque también sirve para las uniones, que vienen ahora, y **`interface`** no.

Con **`type`** también se describen los datos de los ejemplos. Este es el archivo completo: **`Creature`** describe una criatura, y **`Creature[]`** es «un arreglo de **`Creature`**», el tipo de la lista **`creatures`** que importan los demás ejemplos:

src/data/pokemon.ts

```
export type Creature = {
  id: number
  name: string
  types: string[]
}

export const creatures: Creature[] = [
  { id: 1, name: 'Brotín', types: ['grass', 'poison'] },
  { id: 2, name: 'Fogarín', types: ['fire'] },
  { id: 3, name: 'Lagunix', types: ['water'] },
  { id: 4, name: 'Chispón', types: ['electric'] },
  { id: 5, name: 'Rocadón', types: ['rock', 'ground'] },
  { id: 6, name: 'Escarabín', types: ['bug', 'grass'] },
]
```

Una **unión** es un tipo que acepta una opción entre varias, separadas por **`|`**. Puede unir tipos amplios, como **`string | null`** («un texto o nada»), o valores concretos. Un texto concreto, como **`'grid'`**, también es un tipo: el que acepta solo ese texto. Se llama **tipo literal**. Con una unión de tipos literales describes un valor que solo puede ser unos pocos textos, como la vista de la Pokédex:

src/typescript/basics.ts

```
type View = 'grid' | 'list'
let view: View = 'grid'
view = 'list'
```

Cualquier otro texto es un error:

ANTES

unknown-view.ts

```
type View = 'grid' | 'list'
const view: View = 'table'
```

DESPUÉS

```
unknown-view.ts(2,7): error TS2322: Type
'"table"' is not assignable to type
'View'.
```

Con una unión, TypeScript solo te deja hacer con el valor lo que vale para todas sus opciones: por ejemplo, no te deja devolver un **`string | null`** donde se espera un **`string`**. Para lo demás, primero tienes que

comprobar cuál es. El **estrechamiento** (*narrowing*) es esa comprobación: después de un **`if`**, TypeScript sabe qué opción quedó.

En la ficha de un Pokémon hay dos imágenes: **`spriteUrl`**, la chica, que siempre existe, y **`artworkUrl`**, la grande, que puede faltar (**`null`**). Esta función devuelve un texto, y por eso su tipo de retorno, después de los paréntesis, es **`: string`**. El objeto que recibe está descrito ahí mismo, sin nombre propio: es la tercera forma de escribir un tipo de objeto. En un tipo, las propiedades se separan con **`;`**, con **`,`** o con un salto de línea. Si devuelve la imagen grande sin comprobar nada, TypeScript protesta, porque podría ser **`null`**:

ANTES

without-narrowing.ts

```ts
function imageOf(pokemon: { spriteUrl: string; artworkUrl: string | null }): string {
  return pokemon.artworkUrl
}
```

DESPUÉS

```
without-narrowing.ts(2,3): error TS2322: Type 'string | null' is not assignable to type 'string'.
  Type 'null' is not assignable to type 'string'.
```

Con el **`if`**, después de descartar el **`null`**, TypeScript sabe que **`artworkUrl`** es un **`string`**:

src/typescript/basics.ts

```ts
function imageOf(pokemon: { spriteUrl: string; artworkUrl: string | null }): string {
  if (pokemon.artworkUrl === null) {
    return pokemon.spriteUrl
  }
  return pokemon.artworkUrl
}
```

El tipo de una función

Una función también tiene tipo: el de sus parámetros y el de lo que devuelve, unidos por una flecha. **`(text: string) => void`** es «una función que recibe un texto y no devuelve nada»: **`void`** es el tipo de retorno de una función que no devuelve un valor. Vas a usarlo cuando un componente reciba una función por props:

src/typescript/basics.ts

```ts
type OnTextChange = (text: string) => void

const logText: OnTextChange = (text) => {
  console.log(text)
}
```

Revisar y borrar: dos trabajos distintos

El navegador no entiende TypeScript, solo JavaScript. Antes de ejecutar, hay dos trabajos:

1. **Revisar los tipos.** Lo hace **`tsc`**, y tu editor usa el mismo motor de TypeScript para subrayar los errores mientras escribes.
2. **Quitar los tipos** para obtener JavaScript. Lo puede hacer **`tsc`** o, en un proyecto con Vite, una herramienta que Vite usa por debajo y que es mucho más rápida porque no revisa nada.

Un **compilador** es un programa que traduce código de un lenguaje a otro. Cuando los dos lenguajes se parecen tanto como TypeScript y JavaScript, la traducción también se llama **transpilar**.

Al traducir, los tipos **se borran**: el JavaScript que llega al navegador no tiene ni uno. Este es el resultado de pasar **`greet.ts`** por **`tsc`** (también cambia el formato, con otra sangría y un **`;`**, pero lo que importa es que no quedó ningún tipo):

ANTES

greet.ts

```ts
type Trainer = { name: string }

export function greet(trainer: Trainer):
string {
  return `Hi, ${trainer.name}`
}
```

DESPUÉS

greet.js

```js
export function greet(trainer) {
    return `Hi, ${trainer.name}`;
}
```

De aquí sale una consecuencia que conviene tener clara desde ahora: los tipos te protegen mientras escribes el código, no mientras se ejecuta. Si una **API** (el servidor al que la aplicación le pide datos) responde con otra forma, es decir, con otras propiedades o de otro tipo, TypeScript no se entera. En la parte 2 vas a ver cómo validar esos datos cuando llegan.

Errores frecuentes

> **El error típico:** silenciar un error con **`any`**. **`any`** es el tipo que acepta cualquier cosa: si escribes **`imageOf(pokemon: any)`**, la línea roja desaparece. Pero **`any`** apaga la revisión de ese valor y de todo lo que se calcula a partir de él, así que el error deja de detectarse antes de ejecutar. La corrección es escribir el tipo real.

Si vienes de Vue o Angular

En Angular, TypeScript es el lenguaje de todo proyecto, y su compilador se construye sobre el de TypeScript. Vue 3 tiene soporte completo para TypeScript (**`<script setup lang="ts">`**), aunque muchas apps de Vue siguen en JavaScript.

Un detalle de 2026: según el anuncio de TypeScript 7, los proyectos de Vue tienen que seguir por ahora en TypeScript 6, y en Angular la revisión de las plantillas y el soporte del editor también se quedan en TypeScript 6, aunque TypeScript 7 sirve para revisar el proyecto entero desde la línea de comandos (Daniel Rosenwasser, *Announcing TypeScript 7.0*). En React no hay plantillas aparte: el JSX vive en los archivos **`.tsx`**, el compilador de TypeScript entiende esa sintaxis, y TypeScript 7 la revisa directamente.

> **– Para profundizar**
>
> TypeScript 7.0 no trae todavía una API programática; según su anuncio, llegará una nueva en la 7.1. Las herramientas que la necesitan, como typescript-eslint, funcionan instalando TypeScript 6 al lado, con el paquete **`@typescript/typescript6`**. La plantilla oficial de Vite para React con TypeScript (create-vite 9.3.4) trae oxlint en lugar de ESLint. Esa plantilla tampoco declara **`"strict": true`** en su **`tsconfig`**, y aun así TypeScript rechaza las props sin tipo de la sección anterior, porque desde TypeScript 6.0 el modo estricto viene activado por defecto (Daniel Rosenwasser, *Announcing TypeScript 6.0*). El estrechamiento de **`string | null`** depende de ese modo: sin **`strictNullChecks`**, que es parte de **`strict`**, TypeScript no distinguiría el **`null`**. Y otra diferencia entre **`type`** e **`interface`**: una **`interface`** se puede volver a declarar para agregarle propiedades (*declaration merging*); un **`type`**, no.

Condicionales y listas

En esta sección vas a mostrar u ocultar partes de la pantalla y a repetir un componente por cada dato.

Listas con map

En React, una lista se arma con **`map`**: por cada dato, un elemento.

src/lists/CreatureList.tsx

```
import type { Creature } from '../data/pokemon'

export function CreatureList({ creatures }: { creatures: Creature[] }) {
  return (
    <ul>
      {creatures.map((creature) => (
        <li key={creature.id}>{creature.name}</li>
      ))}
    </ul>
  )
}
```

Cada elemento de una lista lleva una **key**: un valor que lo distingue de los otros elementos de la misma lista. Técnicamente, es la marca con la que React reconoce, entre un render y el siguiente, qué elemento es cuál, para mover, agregar o quitar solo lo necesario. Tiene que ser única dentro de la lista y estable, es decir, la misma para el mismo dato en cada render: el **`id`** de cada criatura.

Por qué no usar el índice como key

El índice del **`map`** parece una key cómoda, pero no identifica al dato: identifica la posición. En esta lista cada criatura tiene un campo para escribir su apodo. El campo no guarda su valor en React: es el navegador el que recuerda lo que escribiste en él. El **`aria-label`** le da al campo un nombre para los lectores de pantalla:

ANTES

src/lists/NicknameList.tsx

```tsx
import type { Creature } from
'../data/pokemon'

export function NicknameListByIndex({
creatures }: { creatures: Creature[] }) {
  return (
    <ul>
      {creatures.map((creature, index) =>
(
        <li key={index}>
          {creature.name} <input aria-
label={`Apodo de ${creature.name}`} />
        </li>
      ))}
    </ul>
  )
}
```

DESPUÉS

src/lists/NicknameList.tsx

```tsx
export function NicknameListById({
creatures }: { creatures: Creature[] }) {
  return (
    <ul>
      {creatures.map((creature) => (
        <li key={creature.id}>
          {creature.name} <input aria-
label={`Apodo de ${creature.name}`} />
        </li>
      ))}
    </ul>
  )
}
```

Imagina que escribes «Brote» en el campo de Brotín, el primero, y que después la lista cambia sin Brotín (por ejemplo, porque quien le pasa la lista la filtra; en «Elevar el estado» vas a armar un filtro así). Con el índice como key, React cree que el elemento de la posición 0 sigue ahí, solo que ahora con otro nombre: el campo con «Brote» queda junto a Fogarín, que pasó a ser el primero. Con **`key={creature.id}`**, el campo se va con Brotín. Las pruebas del ejemplo comprueban los dos casos.

Renderizado condicional

El **renderizado condicional** es mostrar una cosa u otra según los datos. Se escribe con las herramientas de siempre de JavaScript:

- un **retorno temprano**, con un **`if`** antes del **`return`** principal;

- el **operador ternario `condición ? a : b`**, para elegir entre dos;
- **`condición && elemento`**, para mostrar algo solo si la condición se cumple.

src/lists/Conditional.tsx

```tsx
import type { Creature } from '../data/pokemon'

export function Results({ creatures }: { creatures: Creature[] }) {
  if (creatures.length === 0) {
    return <p>Ninguna criatura coincide.</p>
  }
  return (
    <>
      <p>{creatures.length === 1 ? '1 criatura' : `${creatures.length} criaturas`}</p>
      {creatures.length > 5 && <p>Hay muchas: prueba a filtrar.</p>}
    </>
  )
}
```

> Cuidado: React no muestra nada para **`false`**, **`null`** ni **`undefined`**, pero sí muestra los números. Con **`&&`**, si la condición vale **`0`**, el resultado es **`0`** y aparece en pantalla. Por eso el ejemplo compara (**`creatures.length > 5`**) en lugar de escribir **`creatures.length && …`**.

Estado y eventos

En esta sección vas a hacer que la pantalla cambie cuando el usuario actúa.

El problema: una variable común no actualiza la pantalla

Un botón para marcar un favorito, guardando la marca en una variable:

src/state/FavoriteWithVariable.tsx

```tsx
export function FavoriteWithVariable() {
  let favorite = false
  return (
    <button onClick={() => (favorite = !favorite)}>{favorite ? '★ Favorito' : '☆ Marcar'}</button>
  )
}
```

Haces clic y no pasa nada. La variable sí cambia, pero React no se entera: no tiene motivo para volver a llamar al componente, así que la pantalla sigue igual. Y si por otra razón lo volviera a llamar, la función arrancaría de nuevo y **`favorite`** volvería a ser **`false`**.

oxlint, el **linter** que trae la plantilla (un programa que revisa el código en busca de errores comunes; se corre con `npm run lint`), ya lo avisa: «Cannot reassign variable after render completes», no se puede reasignar una variable después de que terminó el render.

> – **Para profundizar**
>
> el aviso completo de oxlint. Menciona React Compiler, una herramienta que optimiza componentes al compilar y que vas a ver en la parte 3.

```
⚠ react(immutability): Cannot reassign variable after render completes
   ╭─[src/state/FavoriteWithVariable.tsx:4:29]
 1 │ export function FavoriteWithVariable() {
 2 │   let favorite = false
   ·       ────┬───
   ·           └── `favorite` is declared here
 3 │   return (
 4 │     <button onClick={() => (favorite = !favorite)}>{favorite ? '★ Favorito' :
'☆ Marcar'}</button>
   ·                               ────┬────
   ·                                   └── Cannot reassign `favorite` after render
completes
 5 │   )
   ╰────

  help: Reassigning `favorite` after render has completed can cause inconsistent
behavior on subsequent renders. Consider using state instead
  note: React Compiler skipped optimizing this component or hook. Additional
guidance: https://react.dev/reference/eslint-plugin-react-hooks/lints/immutability

  ⚠ react(immutability): Cannot modify local variables after render completes
   ╭─[src/state/FavoriteWithVariable.tsx:4:29]
 3 │   return (
 4 │     <button onClick={() => (favorite = !favorite)}>{favorite ? '★ Favorito' :
'☆ Marcar'}</button>
   ·                        ─┬     ────┬────
   ·                         │         └── This modifies `favorite`
   ·                         └── This function may (indirectly) reassign or modify
`favorite` after render
 5 │   )
   ╰────

  help: This argument is a function which may reassign or mutate `favorite` after
render, which can cause inconsistent behavior on subsequent renders. Consider using
state instead
  note: React Compiler skipped optimizing this component or hook. Additional
guidance: https://react.dev/reference/eslint-plugin-react-hooks/lints/immutability

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

Estado

El **estado** es la memoria de un componente: datos que React guarda entre un render y el siguiente, y que al cambiar provocan un render nuevo. Se crea con **`useState`**, que se importa de **`react`**:

src/state/Favorite.tsx

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

```
export function FavoriteWithState() {
  const [favorite, setFavorite] = useState(false)
  return <button onClick={() => setFavorite(!favorite)}>{favorite ? '★ Favorito' : '☆
Marcar'}</button>
}
```

`useState(false)` devuelve un arreglo con dos cosas, que se sacan con desestructuración: el valor actual (**`favorite`**) y la función que lo cambia (**`setFavorite`**). Al llamar a **`setFavorite`**, React anota el valor nuevo y programa un render; en ese render, **`useState`** devuelve el valor nuevo.

`useState` es un **hook**: una función de React que le da a un componente una capacidad, en este caso, recordar. React identifica cada hook por el orden en que se llama, así que un hook se llama siempre en el nivel de arriba del componente (o de un hook propio, que vas a ver en la parte 2): nunca dentro de un **`if`**, de un bucle, de un manejador ni de otra función anidada, ni después de un retorno temprano. Por convención, su nombre empieza con **`use`**.

Un **evento** es algo que pasa en la página: un clic, una tecla, un cambio en un campo. El **manejador** es la función que se ejecuta cuando pasa ese evento: aquí, la que pasas en **`onClick`**.

Cada render es una foto

Por ahora, piénsalo así: cada render es una foto del componente con los valores de ese momento. Dentro de esa foto, **`count`** no cambia aunque llames a **`setCount`**: el valor nuevo aparece en la foto siguiente. Por eso el primer bloque suma uno, no dos:

ANTES

src/state/PlusTwo.tsx

```
export function PlusTwoWrong() {
  const [count, setCount] = useState(0)
  function addTwo() {
    setCount(count + 1)
    setCount(count + 1)
  }
  return <button onClick={addTwo}>Nivel
{count}</button>
}
```

DESPUÉS

src/state/PlusTwo.tsx

```
export function PlusTwoRight() {
  const [count, setCount] = useState(0)
  function addTwo() {
    setCount((current) => current + 1)
    setCount((current) => current + 1)
  }
  return <button onClick={addTwo}>Nivel
{count}</button>
}
```

(Los dos usan **`import { useState } from 'react'`**, como el ejemplo anterior.) Las dos llamadas del primer bloque dicen lo mismo: «pon **`count`** en el valor de esta foto más 1». Las del segundo usan una **actualización funcional**: le pasan a **`setCount`** una función que recibe el valor más reciente y devuelve el siguiente. React las aplica en orden, y el nivel sube de a dos.

Un campo controlado

Un **input controlado** es un campo cuyo valor sale del estado: lo que muestra es **`text`**, y cada tecla actualiza **`text`**. En React, **`onChange`** se dispara con cada tecla, no al salir del campo, y **`event.target.value`** es el texto del campo en ese momento (**`event`** es el objeto que describe el evento, y **`target`**, el elemento donde pasó). Así el valor está siempre disponible para el resto del componente, como en este buscador:

src/state/SearchBox.tsx

```
import { useState } from 'react'
import { creatures } from '../data/pokemon'

export function SearchBox() {
  const [text, setText] = useState('')
  const visible = creatures.filter((creature) =>
    creature.name.toLowerCase().includes(text.toLowerCase()),
  )
  return (
    <>
      <label>
        Buscar <input value={text} onChange={(event) => setText(event.target.value)} />
      </label>
      <ul>
        {visible.map((creature) => (
          <li key={creature.id}>{creature.name}</li>
        ))}
      </ul>
    </>
  )
}
```

`visible` no es un estado aparte: se calcula en cada render a partir de **`text`**. Si un dato se puede calcular a partir de otros, no lo guardes en su propio estado: se desincronizaría.

Elevar el estado y useReducer

En esta sección vas a compartir un dato entre dos componentes y a ordenar sus cambios. Al final vas a tener la mini Pokédex de esta parte.

El problema: dos componentes, un mismo dato

La mini Pokédex tiene un buscador y una lista. El buscador cambia los filtros y la lista los usa. Si cada uno guarda sus filtros en su propio estado, no hay forma de que la lista se entere de lo que se escribió en el buscador.

Elevar el estado

La solución es **elevar el estado**: llevarlo al componente padre de los dos. El padre guarda los filtros, se los pasa a la lista por props y le pasa al buscador funciones para cambiarlos. Así hay una **fuente única de verdad**: un solo lugar donde vive cada dato.

Primero, los filtros y la función que los aplica. El campo se llama **`creatureType`**, y no **`type`**, porque más abajo las acciones del reducer usan su propio campo **`type`** para decir qué pasó, y una acción con dos campos **`type`** no podría existir:

src/lifting/filters.ts

```
import type { Creature } from '../data/pokemon'

export type Filters = { text: string; creatureType: string }

export function applyFilters(creatures: Creature[], filters: Filters): Creature[] {
  return creatures.filter(
    (creature) =>
      creature.name.toLowerCase().includes(filters.text.toLowerCase()) &&
      (filters.creatureType === '' || creature.types.includes(filters.creatureType)),
  )
}
```

El buscador recibe los filtros y dos funciones, con el tipo de función que viste en «Por qué nace TypeScript». No tiene estado propio: muestra lo que recibe y avisa cuando algo cambia. Las funciones que recibe por props, como **`onTextChange`**, son la forma en que un hijo le habla a su padre:

src/lifting/SearchBar.tsx

```
import type { Filters } from './filters'

type SearchBarProps = {
  filters: Filters
  onTextChange: (text: string) => void
  onCreatureTypeChange: (creatureType: string) => void
}

export function SearchBar({ filters, onTextChange, onCreatureTypeChange }:
SearchBarProps) {
  return (
    <div>
      <label>
        Buscar <input value={filters.text} onChange={(event) =>
onTextChange(event.target.value)} />
      </label>
      <label>
        Tipo
        <select
          value={filters.creatureType}
```

```
          onChange={(event) => onCreatureTypeChange(event.target.value)}
        >
          <option value="">Todos</option>
          <option value="grass">grass</option>
          <option value="fire">fire</option>
          <option value="water">water</option>
        </select>
      </label>
    </div>
  )
}
```

La lista recibe los filtros y muestra las criaturas que pasan. El **`type`** dentro de las llaves del **`import`** marca que **`Filters`** es solo un tipo, como **`import type`** pero para un nombre suelto:

src/lifting/FilteredList.tsx

```
import { creatures } from '../data/pokemon'
import { applyFilters, type Filters } from './filters'

export function FilteredList({ filters }: { filters: Filters }) {
  return (
    <ul>
      {applyFilters(creatures, filters).map((creature) => (
        <li key={creature.id}>{creature.name}</li>
      ))}
    </ul>
  )
}
```

Y el padre guarda los filtros en su estado y los comparte. Cada cambio crea un objeto nuevo con el spread, sin modificar el anterior:

src/lifting/MiniPokedex.tsx

```
import { useState } from 'react'
import { FilteredList } from './FilteredList'
import { SearchBar } from './SearchBar'

export function MiniPokedex() {
  const [filters, setFilters] = useState({ text: '', creatureType: '' })
  return (
    <>
      <SearchBar
        filters={filters}
        onTextChange={(text) => setFilters({ ...filters, text: text })}
        onCreatureTypeChange={(creatureType) => setFilters({ ...filters, creatureType:
creatureType })}
```

```
      />
      <FilteredList filters={filters} />
    </>
  )
}
```

useReducer

Cuando un estado tiene varias formas de cambiar, conviene juntarlas en un solo lugar. Un **reducer** es una función que recibe el estado actual y una **acción**, un objeto que describe qué pasó, y devuelve el estado siguiente, sin modificar el anterior:

src/lifting/filters.ts

```ts
export type FiltersAction =
  | { type: 'textChanged'; text: string }
  | { type: 'creatureTypeChanged'; creatureType: string }
  | { type: 'cleared' }

export function filtersReducer(state: Filters, action: FiltersAction): Filters {
  switch (action.type) {
    case 'textChanged':
      return { ...state, text: action.text }
    case 'creatureTypeChanged':
      return { ...state, creatureType: action.creatureType }
    case 'cleared':
      return { text: '', creatureType: '' }
  }
}
```

`FiltersAction` es una unión de tres tipos de objeto (el primer `|` solo ordena la lista). Todos tienen un campo **`type`** con un texto distinto, y el **`switch`** estrecha como el **`if`**: dentro de **`case 'textChanged'`**, TypeScript sabe que la acción trae **`text`**.

Con **`useReducer`**, el componente no calcula el estado nuevo: le manda la acción al reducer con **`dispatch`**.

src/lifting/MiniPokedexWithReducer.tsx

```tsx
import { useReducer } from 'react'
import { filtersReducer } from './filters'
import { FilteredList } from './FilteredList'
import { SearchBar } from './SearchBar'

export function MiniPokedexWithReducer() {
  const [filters, dispatch] = useReducer(filtersReducer, { text: '', creatureType: '' })
  return (
    <>
      <SearchBar
```

```
        filters={filters}
        onTextChange={(text) => dispatch({ type: 'textChanged', text: text })}
        onCreatureTypeChange={(creatureType) =>
          dispatch({ type: 'creatureTypeChanged', creatureType: creatureType })
        }
      />
      <button onClick={() => dispatch({ type: 'cleared' })}>Limpiar</button>
      <FilteredList filters={filters} />
    </>
  )
}
```

Ahora todos los cambios posibles de los filtros están a la vista en **`filtersReducer`**, y sumar uno, como «limpiar», es agregar un caso. Una regla práctica: **`useState`** para un valor que cambia de una sola forma; **`useReducer`** cuando hay varias acciones que tocan el mismo estado.

Esta es la mini Pokédex de la parte 1. **`App.tsx`** la muestra (en «Estilos» le vas a dar estilo):

src/App.tsx

```
import { MiniPokedexWithReducer } from './lifting/MiniPokedexWithReducer'

export default function App() {
  return (
    <main>
      <h1>Mini Pokédex</h1>
      <MiniPokedexWithReducer />
    </main>
  )
}
```

Estilos

En esta sección vas a darle estilo a un componente con Tailwind CSS.

Por qué hay tantas formas

Con CSS global, una clase **`.button`** de un archivo puede pisar la de otro. Para eso nacieron varias soluciones: CSS Modules, que vuelve únicas las clases de cada archivo; CSS-in-JS, que escribe el estilo dentro del componente; y el CSS de utilidades, que lo arma con clases pequeñas y fijas. Esta serie usa la tercera, con Tailwind.

Tailwind

Tailwind CSS es un framework de CSS **utility-first**: en lugar de inventar una clase por componente, combinas clases pequeñas que hacen una sola cosa (**`px-4`** es el relleno horizontal, **`text-white`** el color del

texto). Esta serie usa la versión 4. Se agrega a Vite con su **plugin**, un paquete que le suma una capacidad a Vite (Tailwind CSS, *Installing with Vite*):

```
npm install tailwindcss @tailwindcss/vite
```

La plantilla ya trae **`vite.config.ts`** con el plugin de React; agrégale el de Tailwind:

vite.config.ts

```
import tailwindcss from '@tailwindcss/vite'
import react from '@vitejs/plugin-react'
import { defineConfig } from 'vite'

// https://vite.dev/config/
export default defineConfig({
  plugins: [react(), tailwindcss()],
})
```

Después, reemplaza todo el contenido de **`src/index.css`**, el archivo de estilos que **`main.tsx`** ya carga, por esto. Los estilos que traía la plantilla tienen reglas que le ganarían a las clases de Tailwind.

src/index.css

```
@import 'tailwindcss';

@theme {
  --color-brand: #0f766e;
}
```

`@import 'tailwindcss'` trae Tailwind, y la directiva **`@theme`** (una instrucción que Tailwind lee dentro del CSS) define los valores propios del proyecto. Cada variable de color (**`--color-`** más un nombre, como **`--color-brand`**) genera una familia de clases: **`bg-brand`** para el fondo, **`text-brand`** para el texto, **`border-brand`** para el borde. En este botón, **`rounded-lg`** redondea las esquinas y **`py-2`** es el relleno vertical:

src/styles/TailwindButton.tsx

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

export function TailwindButton({ children }: { children: ReactNode }) {
  return <button className="rounded-lg bg-brand px-4 py-2 text-white">{children}
</button>
}
```

> **Cuidado:** Tailwind genera solo las clases que encuentra escritas completas en tu código. Si armas el

nombre por partes (`` `bg-${color}` ``), la clase no existe. Escribe el nombre entero.

– Para profundizar

el mismo botón con las otras dos formas. Con **CSS Modules**, que Vite soporta sin instalar nada, las clases de un archivo **`.module.css`** se vuelven únicas:

src/styles/Button.module.css

```
.button {
  background: #0f766e;
  color: white;
  border-radius: 0.5rem;
  padding: 0.5rem 1rem;
}
```

src/styles/ModuleButton.tsx

```
import type { ReactNode } from 'react'
import styles from './Button.module.css'

export function ModuleButton({ children }: { children: ReactNode }) {
  return <button className={styles.button}>{children}</button>
}
```

Con **styled-components**, el CSS-in-JS más conocido (se instala con **`npm install styled-components`**), el estilo es parte del componente:

src/styles/StyledButton.tsx

```
import styled from 'styled-components'

export const StyledButton = styled.button`
  background: #0f766e;
  color: white;
  border-radius: 0.5rem;
  padding: 0.5rem 1rem;
`
```

styled-components está en modo mantenimiento desde el 17 de marzo de 2025: su mantenedor principal recomienda no adoptarlo en proyectos nuevos, aunque sigue publicando correcciones (Evan Jacobs, *Thank you*; la versión 6.6.0 salió en octubre de 2026). Para un proyecto nuevo, Tailwind o CSS Modules.

Si vienes de Vue o Angular

Una tabla para ubicar lo que ya conoces:

En Vue	En Angular	En React
`ref()`	`signal()`	`useState`
`computed()`	`computed()`	un valor calculado en el render (o `useMemo`, parte 3)
`v-model`	`[(ngModel)]`	`value` + `onChange` (input controlado)
`props`	`@Input()` / `input()`	props
`emit`	`@Output()` / `output()`	una función recibida por props
slots	`<ng-content>`	`children`
`v-if`	`@if`	`if` antes del `return`, ternario o `&&`
`v-for` con `:key`	`@for` con `track`	`map` con `key`
plantilla `.vue` / HTML	plantilla HTML	JSX dentro del componente

Tres diferencias que la tabla no muestra:

- **El estado de React es una foto.** `ref.value` y una `signal()` devuelven el valor nuevo apenas lo cambias; en React, el valor nuevo llega en el render siguiente, que vuelve a ejecutar la función del componente entera.
- **El estado se reemplaza, no se muta.** Con un `ref` de Vue puedes cambiar una propiedad de un objeto y la pantalla se entera; en React tienes que crear un objeto nuevo, como hace el spread de la mini Pokédex.
- **`computed()` guarda su resultado** y solo lo recalcula cuando cambian sus dependencias. Un valor calculado en el render se recalcula en cada render; para guardarlo existe `useMemo`, que vas a ver en la parte 3.

Lo que aprendiste

React nació para que la pantalla siga a los datos sin que tengas que tocar el DOM a mano. Lo escribes con componentes, funciones que describen lo que se ve; les pasas datos con props y los compones con `children`. TypeScript describe esos datos y te avisa de un error antes de ejecutar. El estado le da memoria a un componente, y cada cambio programa un render nuevo; cuando dos componentes comparten un dato, lo elevas al padre, y si sus cambios crecen, los ordenas con un reducer. Con eso armaste una mini Pokédex con búsqueda y filtro por tipo.

Qué sigue

En la parte 2: React intermedio la mini Pokédex sale al mundo real: efectos, datos de un servidor con TanStack Query, rutas con React Router, Context, formularios validados y estado global.

Bibliografía

- Pete Hunt, Why did we build React?, blog de React, 5 de junio de 2013. Consultado el 18 de noviembre de 2026.
- Facebook, repositorio `facebook/react`, commit «Initial public release» del 29 de mayo de 2013. Consultado el 18 de noviembre de 2026.
- S. Somasegar, TypeScript: JavaScript Development at Application Scale, Microsoft, 1 de octubre de 2012. Consultado el 17 de noviembre de 2026.
- Anders Hejlsberg, A 10x Faster TypeScript, Microsoft, 11 de marzo de 2025. Consultado el 17 de noviembre de 2026.
- Daniel Rosenwasser, Announcing TypeScript 6.0, Microsoft, 23 de marzo de 2026. Consultado el 17 de noviembre de 2026.
- Daniel Rosenwasser, Announcing TypeScript 7.0, Microsoft, 8 de julio de 2026. Consultado el 17 de noviembre de 2026.
- Vite, Getting Started y Features. Consultados el 18 de noviembre de 2026.
- Tailwind CSS, Installing Tailwind CSS with Vite. Consultado el 18 de noviembre de 2026.
- Evan Jacobs, Thank you, styled-components, 17 de marzo de 2025. Consultado el 18 de noviembre de 2026.
- MDN, Guía de JavaScript. Consultada el 18 de noviembre de 2026.