Universidad Icesi

Introducción a React

Computación en Internet 2 · Semana 14

Universidad Icesi

Universidad ICESI

Lo que veremos hoy

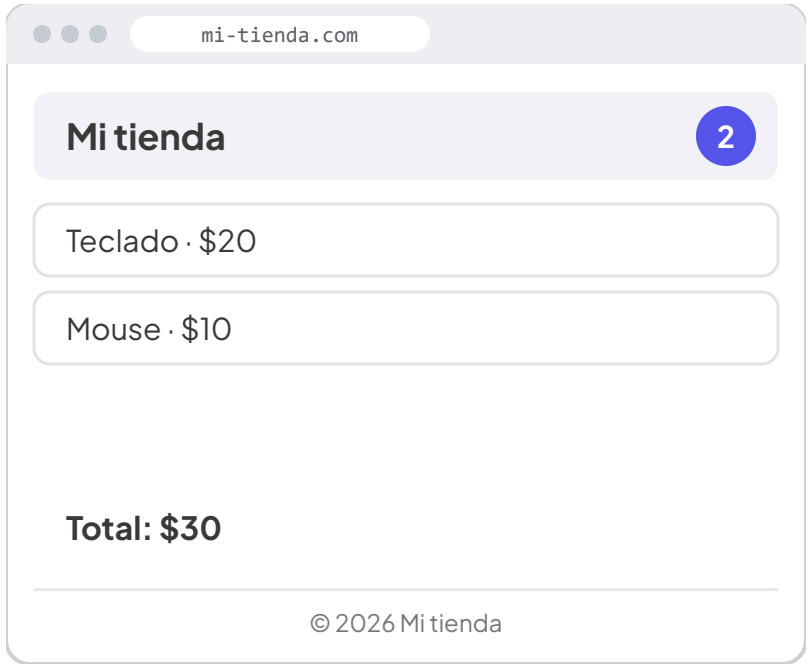
1. Antes de React: la web y el DOM
2. Client Side Rendering
3. Virtual DOM
4. Componentes y composición
5. Instalar React con Vite
6. Estructura del proyecto
7. Tu primer componente
8. Props

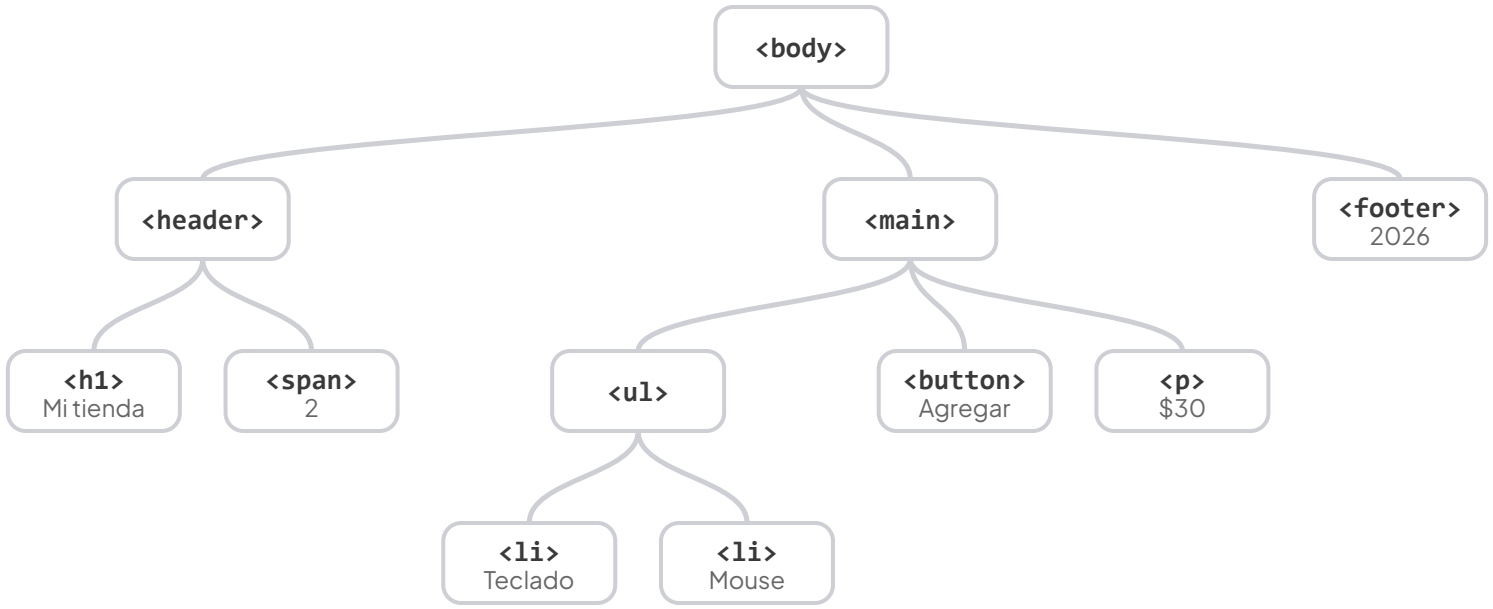
Universidad ICESI

1. Antes de React

La web y el DOM

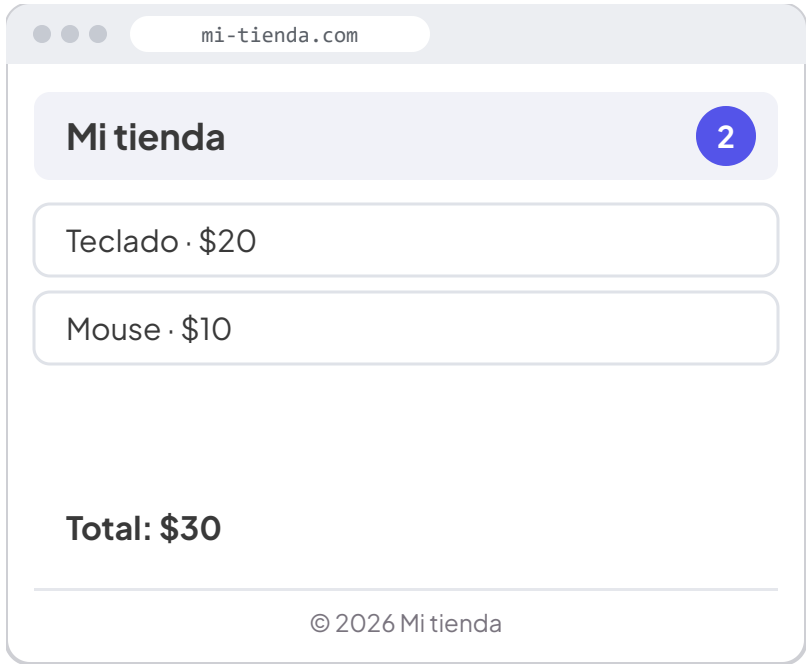
La página es un árbol: el DOM

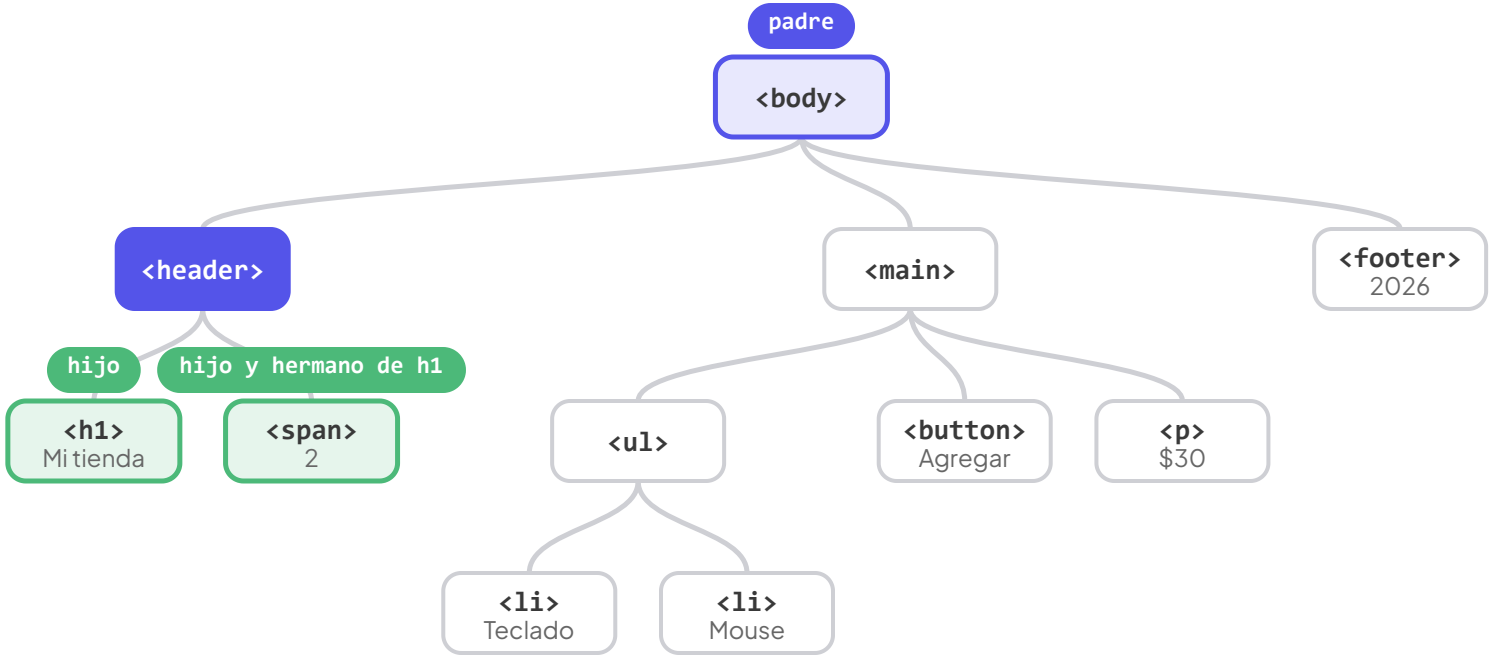




El navegador convierte el HTML en un **árbol de nodos**: el DOM. JavaScript puede leerlo y modificarlo.

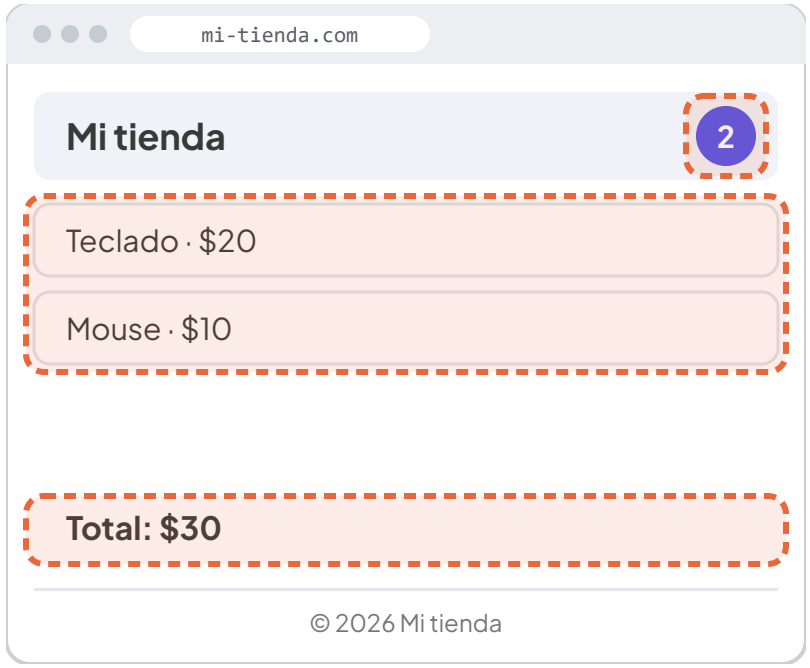
Padres, hijos y hermanos

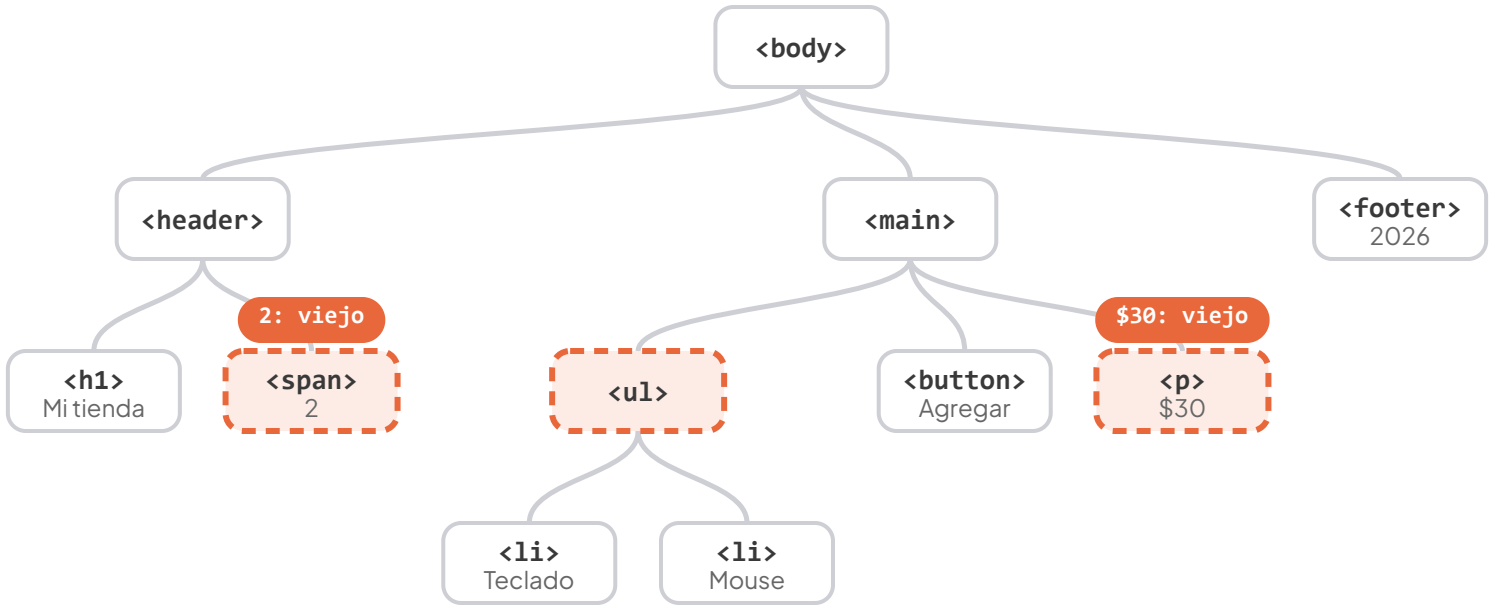




Cada etiqueta es un nodo. **header** contiene a **h1** y **span**: son sus hijos y hermanos entre sí.

Cambia el dato, no la pantalla





Llega un tercer producto. El dato cambió, pero **el DOM no sabe nada**: tres partes de la pantalla quedaron desactualizadas.

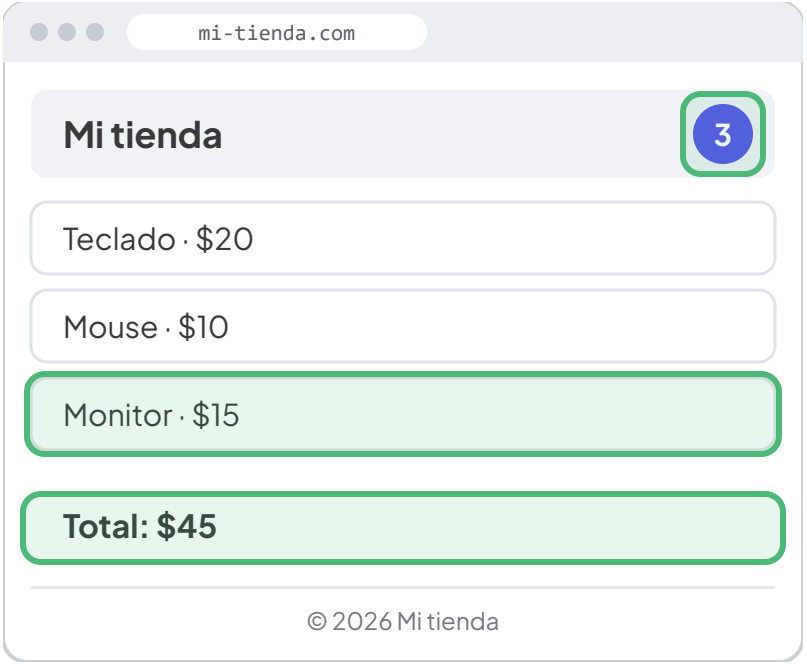
Hay que actualizar cada nodo a mano

carrito.js

```
const contador = document.querySelector(".badge");
contador.textContent = "3";

const lista = document.querySelector("ul");
const item = document.createElement("li");
item.textContent = "Monitor · $15";
lista.appendChild(item);

const total = document.querySelector(".total");
total.textContent = "Total: $45";
```



Buscas cada nodo con `querySelector`, lo cambias y recuerdas **todos los lugares** que dependen del dato: contador, lista y total.

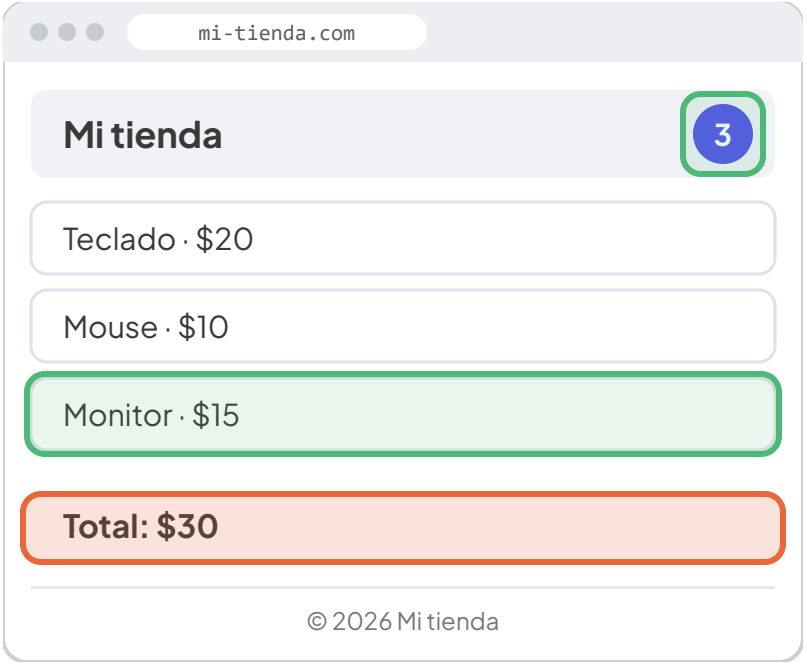
Si olvidas una línea, la pantalla miente

carrito.js

```
const contador = document.querySelector(".badge");
contador.textContent = "3";

const lista = document.querySelector("ul");
const item = document.createElement("li");
item.textContent = "Monitor · $15";
lista.appendChild(item);

const total = document.querySelector(".total");
// total.textContent = "Total: $45"; // se te olvidó
```



Muestra 3 productos pero el total sigue en $30, y **nada avisa del error**.

Universidad ICESI

Qué nos dio el DOM y qué nos costó

VENTAJAS

- Funciona en **todos los navegadores**, sin herramientas ni instalación.
- Puedes **tocar cualquier nodo** de la página.
- Solo HTML y JavaScript: **sin dependencias**.
- Va bien en páginas pequeñas.

DESVENTAJAS

- Tú decides **cómo y dónde** actualizar: estilo imperativo.
- Mantener dato y pantalla sincronizados es **trabajo manual**.
- Con cientos de elementos el código se enreda y es fácil **olvidar un lugar**.

Universidad ICESI

2. Client Side Rendering

Quién cocina la página

CSR: el servidor entrega una página casi vacía

- El servidor entrega un **HTML casi vacío** y el JavaScript
- El **navegador** ejecuta el código y arma la página
- Es el modo por defecto de una app de React hecha con Vite

index.html (lo que entrega el servidor)

```
<body>
  <div id="root"></div>
  <script type="module" src="/assets/app.js"></script>
</body>
```

Universidad ICESI

La analogía: un BBQ coreano

EL COCINERO

Es el servidor.

Prepara lo que entrega: en CSR, solo ingredientes crudos.

EL MAYORDOMO

Es el medio por el que viaja la información.

Lleva el pedido y trae la respuesta; no cocina.

LA MESA CON PARRILLA

Es tu navegador.

Recibe los ingredientes (HTML casi vacío y JavaScript) y los cocina.

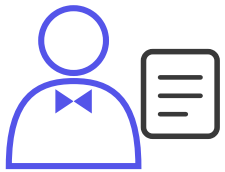
Haces tu pedido y el mayordomo lo lleva

Tu mesa
el navegador



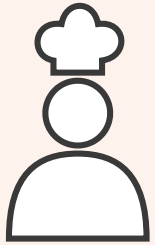
Parrilla

`pedido: GET /`



Mayordomo
el medio: lleva y trae

La cocina
el servidor



Servidor: el cocinero

Tu navegador (la mesa) pide la página. El mayordomo, el medio por el que viaja la información, lleva el pedido hasta el servidor: el cocinero.

En CSR el cocinero entrega ingredientes crudos

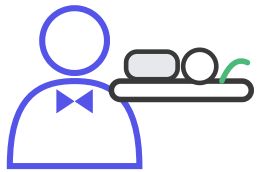
Tu mesa
el navegador



Parrilla

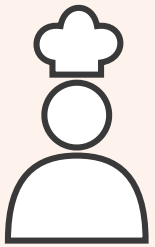
Qué ves: pantalla en blanco

`HTML casi vacío + app.js`



Mayordomo
el medio: lleva y trae

La cocina
el servidor

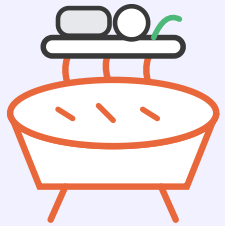


Servidor: el cocinero

El servidor no prepara la página terminada: entrega **un HTML casi vacío y el JavaScript**. El mayordomo solo transporta, no cocina.

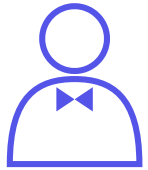
Tú cocinas en tu parrilla

Tu mesa
el navegador



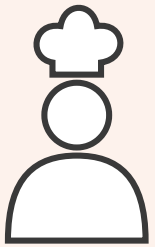
Parrilla

Qué ves: pantalla en blanco



Mayordomo
el medio: lleva y trae

La cocina
el servidor

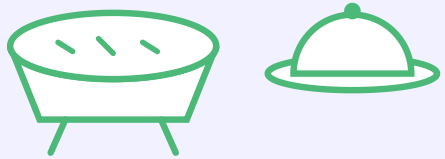


Servidor: el cocinero

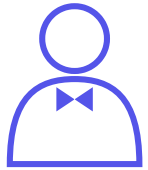
El navegador ejecuta `app.js` y React arma la página elemento por elemento.

A comer: la página está lista

Tu mesa
el navegador

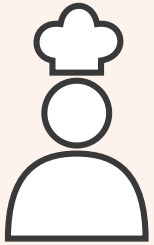


Parrilla **La página**



Mayordomo
el medio: lleva y trae

La cocina
el servidor



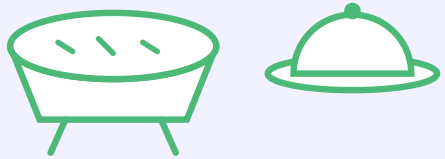
Servidor: el cocinero

Qué ves: la página lista

Ya ves la página y es interactiva. **Todo el trabajo de armarla lo hizo tu navegador.**

Quieres algo más: pides solo los datos

Tu mesa
el navegador

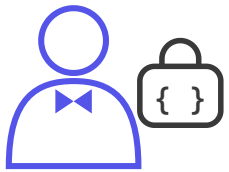


Parrilla

La página

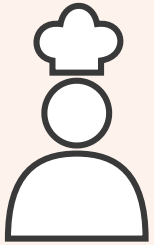
Qué ves: la página lista

JSON (solo datos)



Mayordomo
el medio: lleva y trae

La cocina
el servidor



Servidor: el cocinero

Cambiar algo: sin recargar

Para actualizar la lista el mayordomo lleva un pedido pequeño y trae **solo información**, no una página completa. React actualiza únicamente lo que cambió.

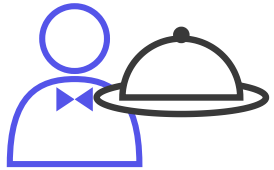
Y si el cocinero entrega el plato terminado: SSR

Tu mesa
el navegador



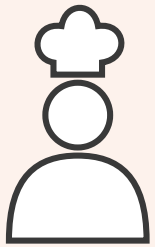
Parrilla

`HTML completo`



Mayordomo
el medio: lleva y trae

La cocina
el servidor



Servidor: el cocinero

Qué ves: la página, completa

Cambiar algo: otro plato, se recarga

En Server Side Rendering el cocinero hace todo el trabajo y deja el **HTML completo**. La primera pantalla llega lista, pero cada cambio pide otro plato completo.

Universidad ICESI

CSR frente a SSR

CSR: COCINAS TÚ

- Servidor liviano: entrega ingredientes.
- Interacciones fluidas: solo viajan datos.
- Primera carga con **pantalla en blanco**.
- Depende de la potencia del equipo.

SSR: COCINA EL SERVIDOR

- Primera pantalla lista al llegar.
- El servidor trabaja más.
- Cada cambio puede **recargar la página**.
- Frameworks como Next.js combinan ambos enfoques.

Universidad ICESI

3. Virtual DOM

Cómo React actualiza la página

React describe qué, no los pasos

DOM A MANO

Tú dices cómo.

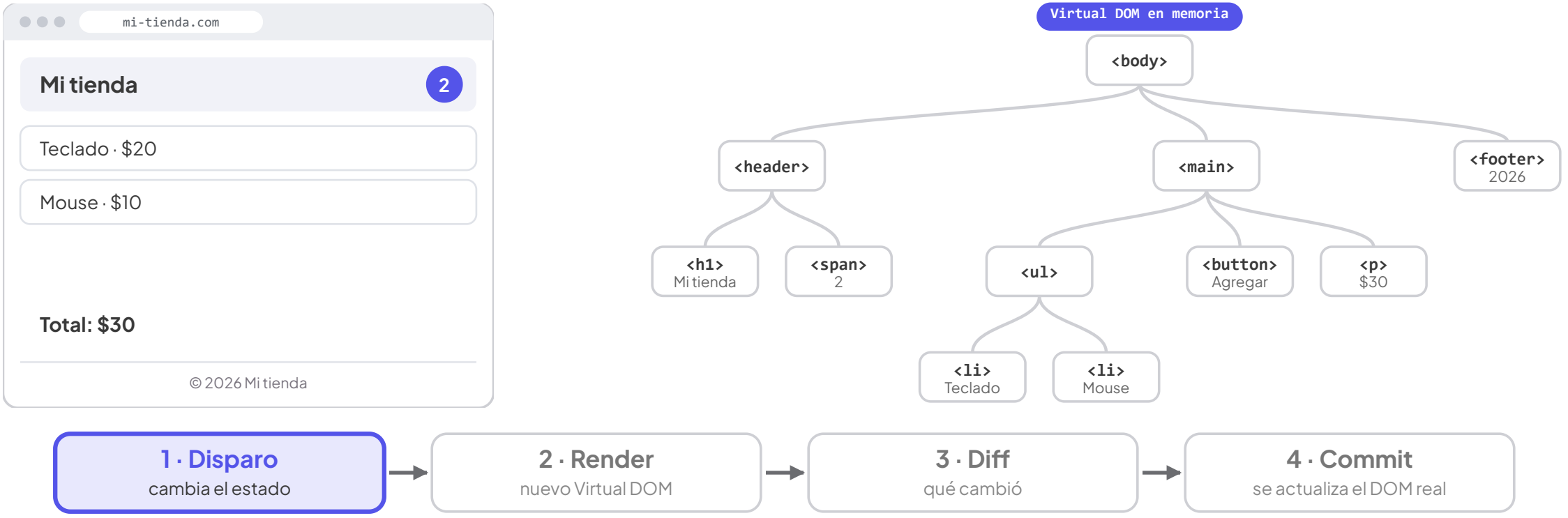
- Buscas cada nodo.
- Lo cambias uno por uno.
- Recuerdas todos los lugares que dependen del dato.

REACT

Tú dices qué.

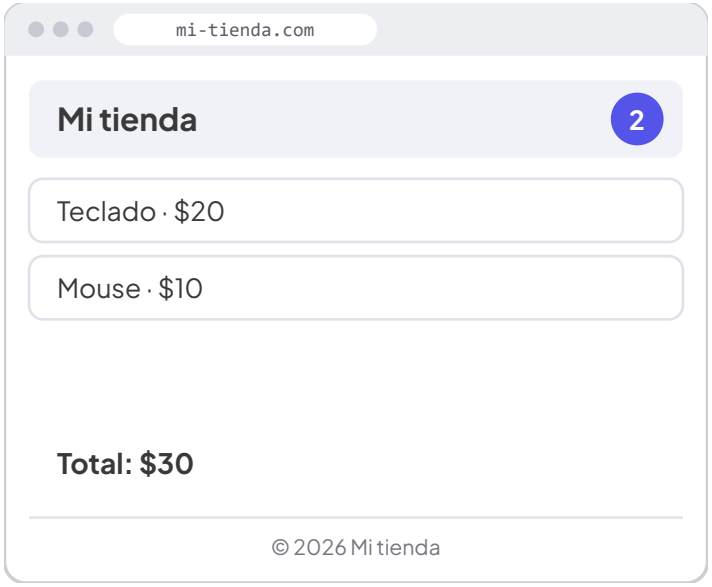
- Describes cómo debe verse la pantalla para un estado.
- React calcula qué cambió.
- Y actualiza solo eso en el DOM real.

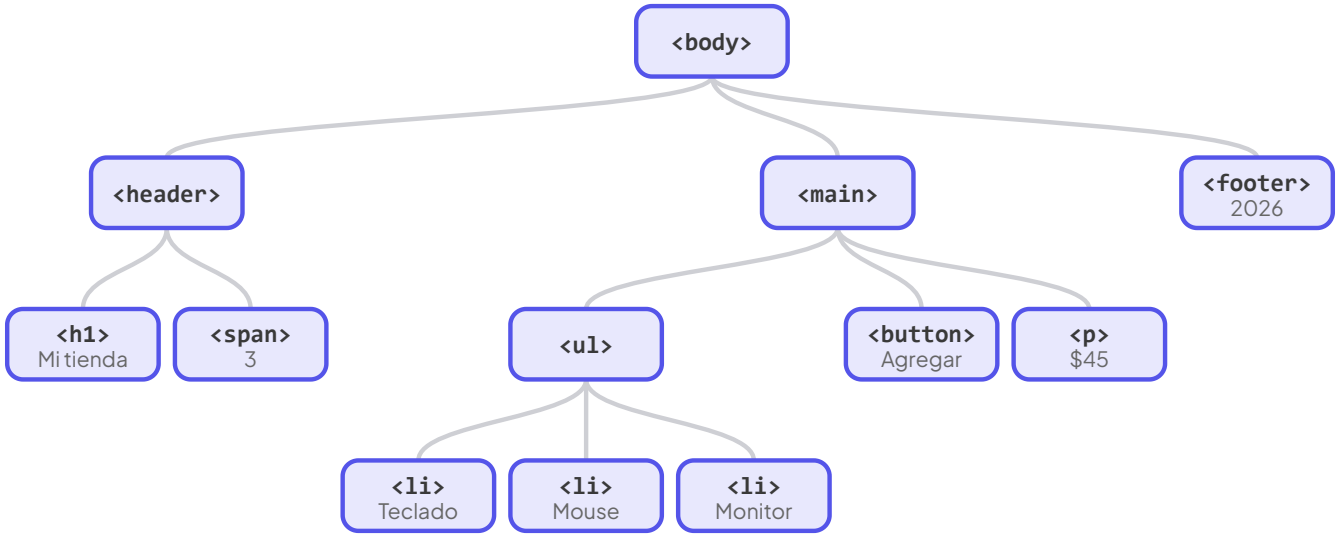
Disparo: el estado cambia



A la izquierda, el DOM real; a la derecha, el **Virtual DOM**: una copia ligera en memoria. El usuario agrega un producto y cambia el estado: la pantalla todavía no cambió.

Render: un Virtual DOM nuevo

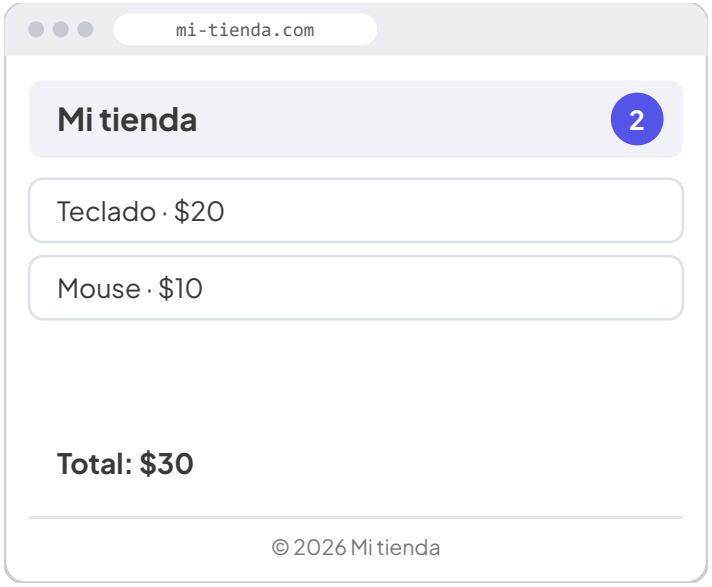


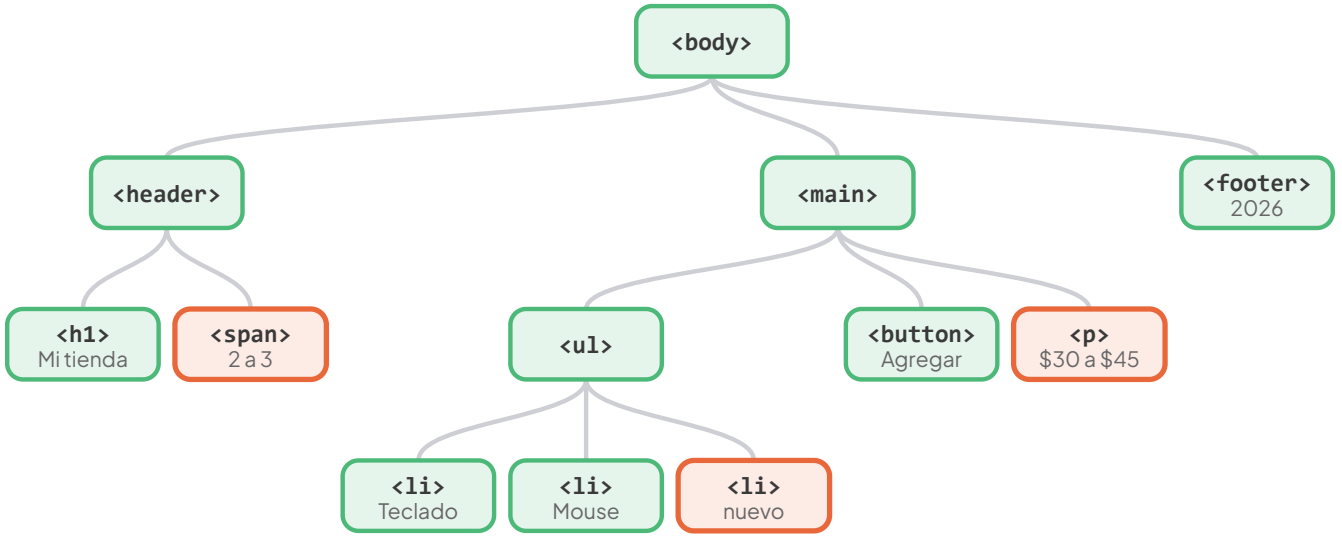


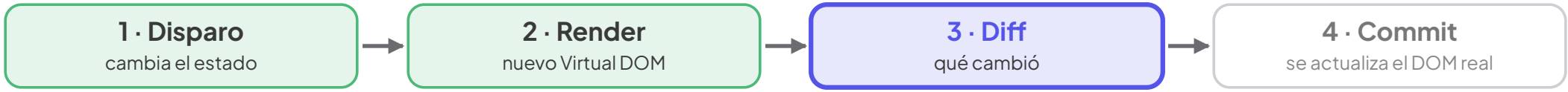
1 · Disparo cambia el estado → **2 · Render** nuevo Virtual DOM → **3 · Diff** qué cambió → **4 · Commit** se actualiza el DOM real

React vuelve a ejecutar tus componentes y construye un árbol nuevo. **Todo ocurre en memoria**: la página sigue igual.

Diff: qué cambió

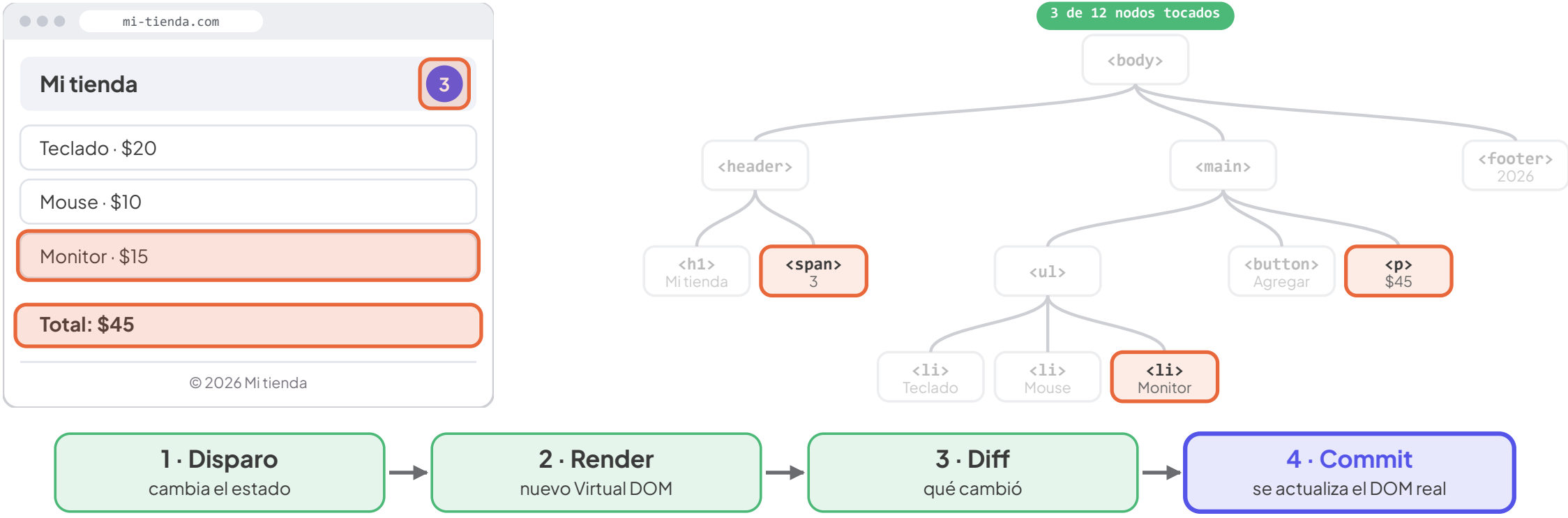






React compara el árbol nuevo con el anterior (**reconciliación**). Solo 3 de 12 nodos son distintos; el resto no se toca.

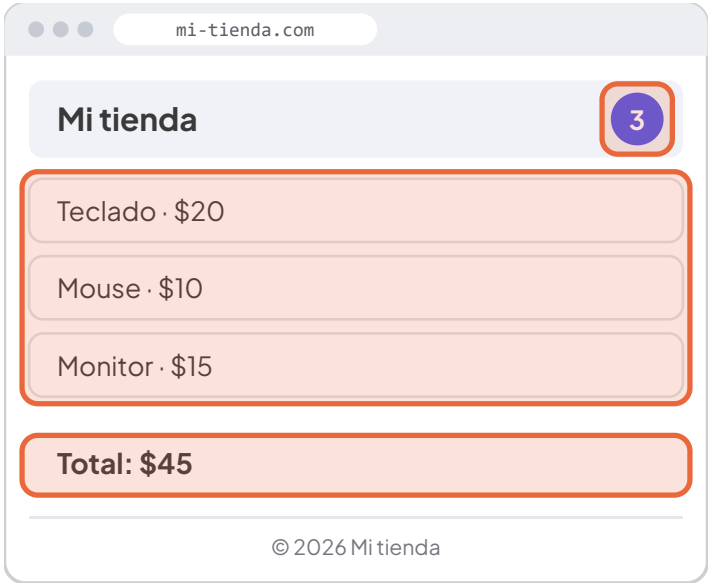
Commit: solo se toca lo necesario

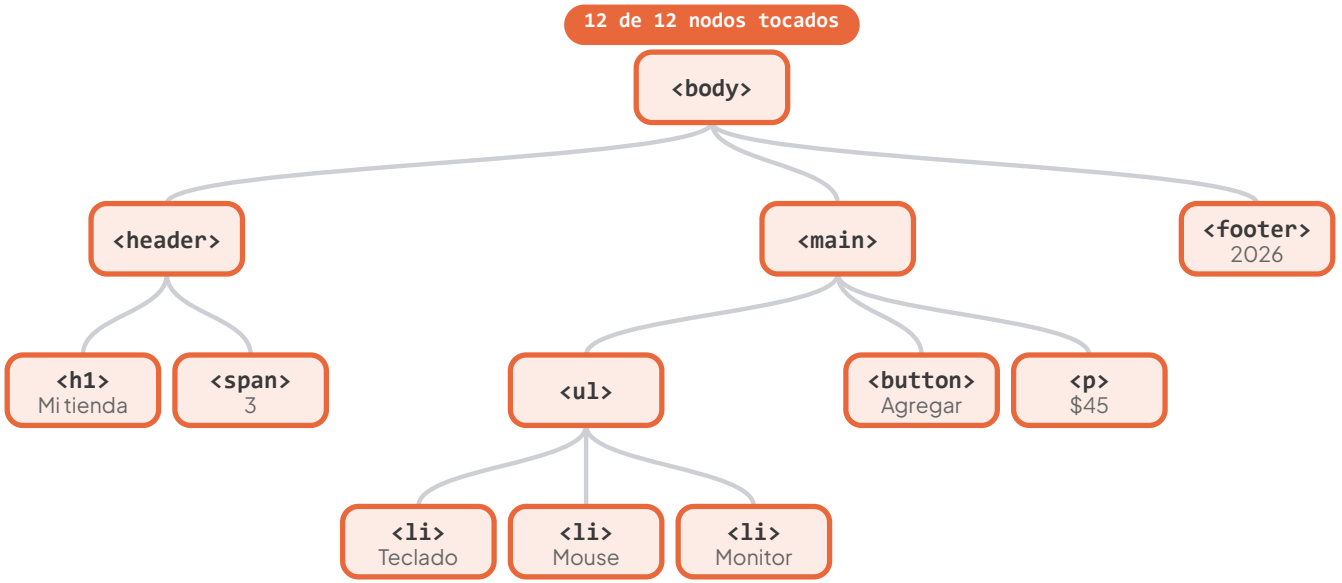


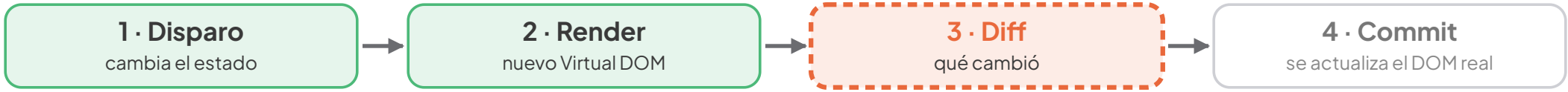
1 · Disparo cambia el estado → **2 · Render** nuevo Virtual DOM → **3 · Diff** qué cambió → **4 · Commit** se actualiza el DOM real

React aplica al DOM real **solo esos tres cambios**. El resto de la página queda exactamente como estaba.

Sin Virtual DOM: reconstruir todo







Sin una copia con qué comparar, no se sabe qué cambió: se reconstruye toda la página. Se pierde lo que el usuario tenía en curso (texto escrito, foco, scroll).

Universidad ICESI

Qué ganas al trabajar con React

Código declarativo
Describes qué se muestra para cada estado, no los pasos para llegar a él.

Pantalla siempre consistente
La interfaz se calcula a partir de los datos: no puedes olvidar un lugar.

Cambios mínimos en el DOM
React toca solo lo que cambió.

Componentes reutilizables
Divides la interfaz en piezas independientes.

Es una librería, no un framework
Se enfoca en la interfaz; el resto lo eliges tú.

Universidad ICESI

1 · Disparo
cambia el estado

↓

2 · Render
nuevo Virtual DOM

↓

3 · Diff
qué cambió

↓

4 · Commit
se actualiza el DOM real

¿Es más rápido el Virtual DOM?

No necesariamente. Modificar el DOM a mano, con mucho cuidado, puede ser igual de rápido.

Lo valioso es que **no tienes que decidir tú qué tocar**: React lo calcula de forma razonablemente eficiente y tu código se mantiene simple.

Universidad ICESI

4. Componentes y composición

Cortar la página en piezas

Universidad ICESI

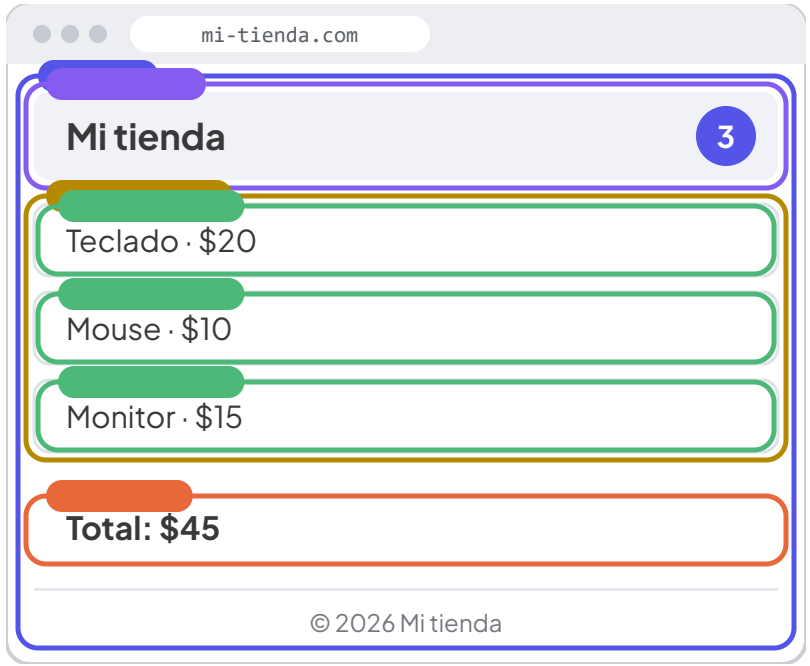
Un componente es una parte de la pantalla

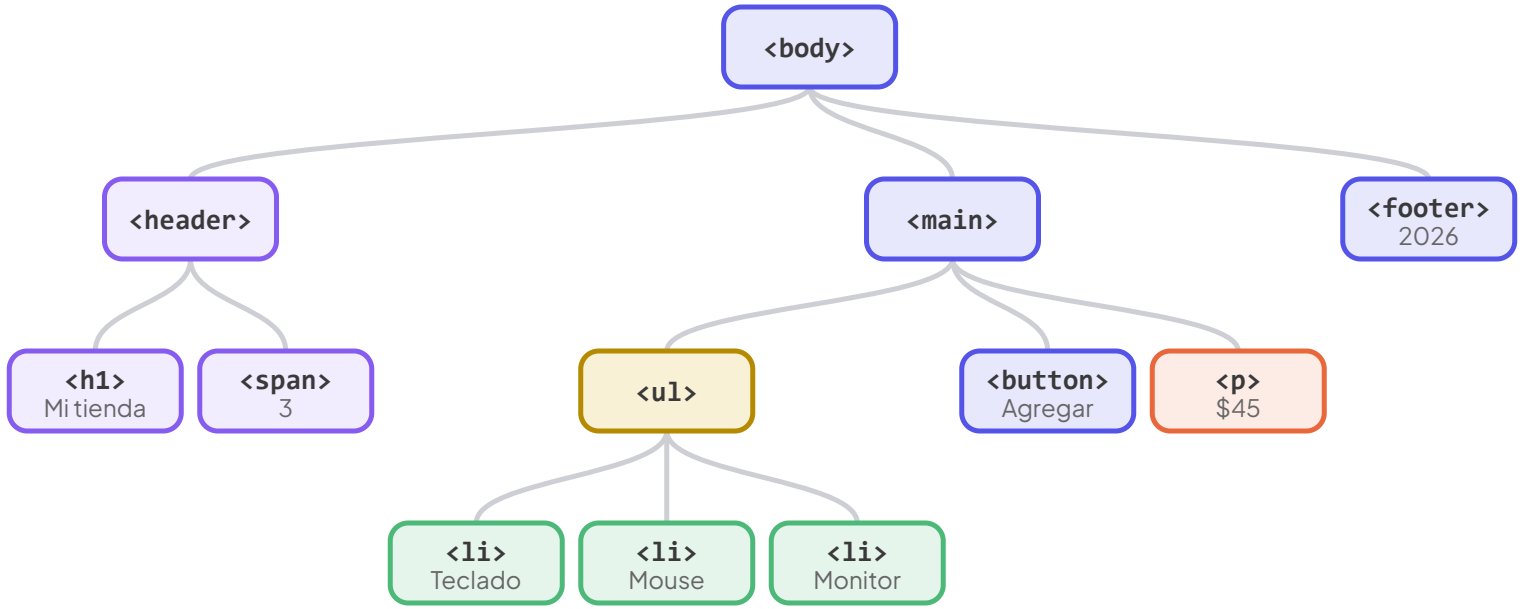
- Es una **función** de JavaScript
- Devuelve una parte de la interfaz, escrita como **JSX**
- Se usa como una **etiqueta propia**: `<Header />`

Header.jsx

```jsx
function Header() {
  return (
    <header>
      <h1>Mi tienda</h1>
      <span className="badge">3</span>
    </header>
  );
}
```

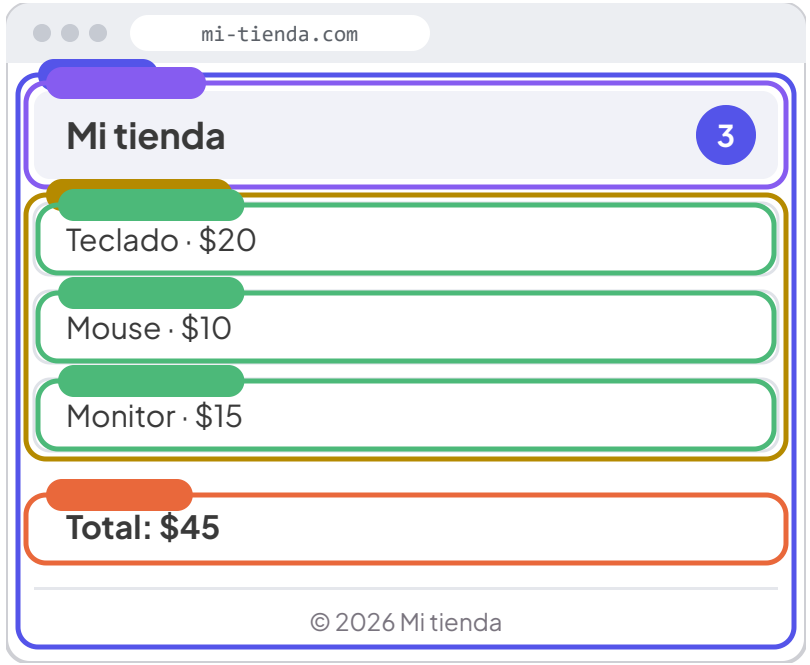
Cortar la página en componentes

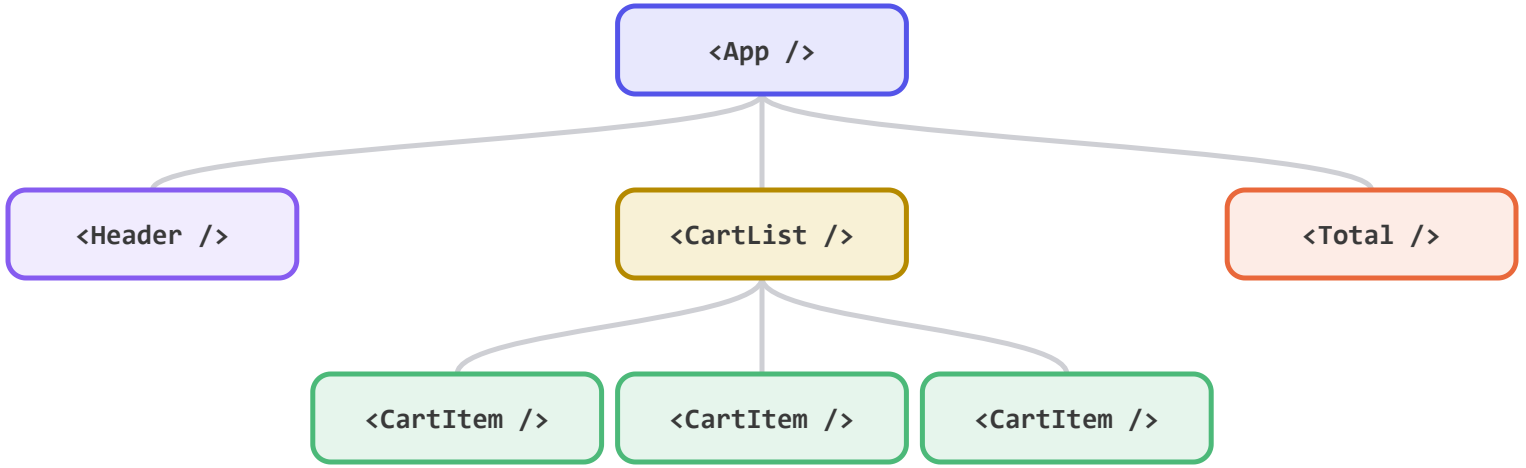




Cada color es un componente. `Header` produce tres nodos del DOM (header, h1 y span): un componente **no es un nodo**, es una forma de agrupar.

El árbol de componentes





Los componentes forman su propio árbol: **App** contiene a Header, a CartList (que contiene tres CartItem) y a Total.

Reutilizar: una función, tres copias

CartItem.jsx

```jsx
function CartItem({ nombre, precio }) {
  return <li>{nombre} · ${precio}</li>;
}
```



`CartItem` se escribe una vez y se usa tres veces. Lo que cambia en cada copia son los datos que recibe: sus **props**.

Un cambio, tres copias

CartItem.jsx

```jsx
function CartItem({ nombre, precio }) {
  return <li>{nombre} · ${precio} · en stock</li>;
}
```

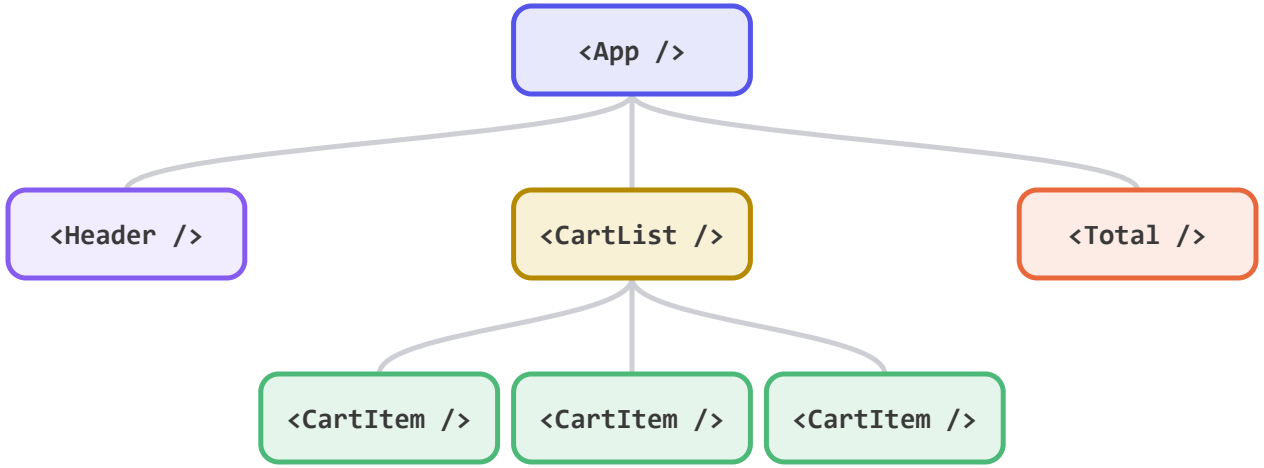


Cambias **una sola línea** del componente y las tres copias se actualizan a la vez.

Componer es anidar

App.jsx

```jsx
function App() {
  return (
    <div>
      <Header />
      <CartList />
      <Total />
      <footer>© 2026 Mi tienda</footer>
    </div>
  );
}
```



Armar una pantalla es colocar componentes dentro de otros. **App no sabe cómo se dibuja un CartItem**: solo lo usa.

Universidad ICESI

5. Instalar React con Vite

Comandos y para qué sirve Vite

Crear el proyecto con Vite

```
Terminal
node -v    # comprueba que tienes Node.js
npm -v

npm create vite@latest mi-app

Project name: mi-app
Select a framework: React
Select a variant: JavaScript
Which linter to use?: Oxlint
Install with npm and start now?: No
```

Pregunta	Elegimos
Framework	`React`
Variant	`JavaScript` (sin TypeScript)
Linter	`Oxlint`, el de por defecto
Instalar y arrancar	`No`: lo hacemos a mano

> Para saltarte las preguntas: `npm create vite@latest mi-app -- --template react`

Instalar y arrancar

Terminal

```
cd mi-app
npm install
npm run dev
```

Lo que imprime la terminal

```
VITE v8.x.x  ready in xxx ms

➜  Local:   http://localhost:5173/
```

Universidad ICESI

Los scripts que usarás en el curso

Comando	Qué hace
`npm run dev`	Arranca el servidor de desarrollo con recarga instantánea
`npm run build`	Genera la carpeta `dist/` lista para publicar
`npm run preview`	Sirve `dist/` en local para probar el build
`npm run lint`	Revisa el código con el linter

¿Y Create React App? Ya no se mantiene: la documentación de React recomienda un framework o una herramienta de build como Vite.

El navegador solo entiende HTML, CSS y JavaScript



Tu app está escrita con **JSX** e importa paquetes (react, react-dom). El navegador no entiende ninguno de los dos: falta alguien en el medio.

npm run dev: Vite transforma y resuelve al vuelo



Vite sirve el proyecto sin empaquetar todo antes: cuando el navegador pide `main.jsx`, **convierte el JSX en JavaScript** y reescribe los imports como `"react"` a rutas reales.

HMR: solo viaja lo que cambió



Con **Hot Module Replacement** la página se actualiza sin recargar y, normalmente, el estado de React (como el contador) se conserva.

npm run build: empaquetar y publicar dist/



Vite junta los módulos en pocos archivos, los minifica y les agrega un hash (**Rolldown** en Vite 8). `dist/` queda listo para cualquier servidor estático; `npm run preview` lo prueba en local.

6. Estructura del proyecto

Qué entrega Vite y cómo organizar src/

Lo que entrega la plantilla de React

```
$ tree -a -L 3 --dirsfirst -I node_modules
.
├── public/
│   ├── favicon.svg
│   └── icons.svg
├── src/
│   ├── assets/
│   │   ├── hero.png
│   │   ├── react.svg
│   │   └── vite.svg
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .gitignore
├── .oxlintrc.json
├── index.html
├── package.json
├── README.md
└── vite.config.js

3 directories, 15 files
```

Una página HTML
index.html es la única; Vite empieza aquí.

Tu código en src/
main.jsx y App.jsx: el resto lo decides tú.

Configuración
package.json, vite.config.js y el linter.

Salida de `tree` con la plantilla de React en JavaScript (sin `node_modules`).

index.html: la puerta de entrada

```
$ tree -a -L 3 --dirsfirst -I node_modules
.
├── public/
│   ├── favicon.svg
│   └── icons.svg
├── src/
│   ├── assets/
│   │   ├── hero.png
│   │   ├── react.svg
│   │   └── vite.svg
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .gitignore
├── .oxlintrc.json
├── index.html
├── package.json
├── README.md
└── vite.config.js

3 directories, 15 files
```

La única página HTML. Vive en la raíz, **no en public/**, porque es el punto de entrada de Vite.

index.html (body)

```html
<body>
  <div id="root"></div>
  <script type="module" src="/src/main.jsx"></script>
</body>
```

> El `div#root` es donde React dibuja toda la interfaz.

main.jsx: donde arranca React

```
$ tree -a -L 3 --dirsfirst -I node_modules
.
├── public/
│   ├── favicon.svg
│   └── icons.svg
├── src/
│   ├── assets/
│   │   ├── hero.png
│   │   ├── react.svg
│   │   └── vite.svg
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .gitignore
├── .oxlintrc.json
├── index.html
├── package.json
├── README.md
└── vite.config.js

3 directories, 15 files
```

main.jsx

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

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

Conecta React con el `div#root` y dibuja `App`, el primer componente: aquí empiezas a escribir.

Configuración: package.json y vite.config.js

```
$ tree -a -L 3 --dirsfirst -I node_modules
.
├── public/
│   ├── favicon.svg
│   └── icons.svg
├── src/
│   ├── assets/
│   │   ├── hero.png
│   │   ├── react.svg
│   │   └── vite.svg
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .gitignore
├── .oxlintrc.json
├── index.html
├── package.json
├── README.md
└── vite.config.js

3 directories, 15 files
```

package.json (scripts)

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "lint": "oxlint",
  "preview": "vite preview"
}
```

vite.config.js

```
export default defineConfig({
  plugins: [react()],
})
```

`dependencies` : react y react-dom. El plugin de React entiende JSX y activa Fast Refresh.

public/ frente a src/assets/

```
$ tree -a -L 3 --dirsfirst -I node_modules
.
├── public/
│   ├── favicon.svg
│   └── icons.svg
├── src/
│   ├── assets/
│   │   ├── hero.png
│   │   ├── react.svg
│   │   └── vite.svg
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .gitignore
├── .oxlintrc.json
├── index.html
├── package.json
├── README.md
└── vite.config.js

3 directories, 15 files
```

public/

Se sirve tal cual y se pide por URL: `/favicon.svg`. Vite no lo procesa.

src/assets/

Se importa desde el código. Vite lo procesa y en el build le agrega un hash.

Vite no decide cómo organizas src/: una propuesta

```
$ tree -L 2 src --dirsfirst

src
├── assets/
│   ├── hero.png
│   ├── react.svg
│   └── vite.svg
├── components/ nuevo
│   └── Button.jsx nuevo
├── hooks/ nuevo
│   └── useFetch.js nuevo
├── pages/ nuevo
│   └── HomePage.jsx nuevo
├── routes/ nuevo
│   └── AppRoutes.jsx nuevo
├── services/ nuevo
│   └── productService.js nuevo
├── utils/ nuevo
│   └── formatPrice.js nuevo
├── App.css
├── App.jsx
├── index.css
└── main.jsx

7 directories, 13 files
```

components
Piezas de interfaz reutilizables

pages
Una por pantalla

hooks
Lógica reutilizable con estado

services
Comunicación con el backend

utils
Funciones puras sin React

routes
Qué pantalla va en cada URL

Seis carpetas, seis responsabilidades

components/

`Button.jsx` : piezas de interfaz que reciben props.

pages/

`HomePage.jsx` : una por pantalla.

hooks/

`useFetch.js` : lógica con estado reutilizable.

services/

`productService.js` : peticiones HTTP, sin JSX.

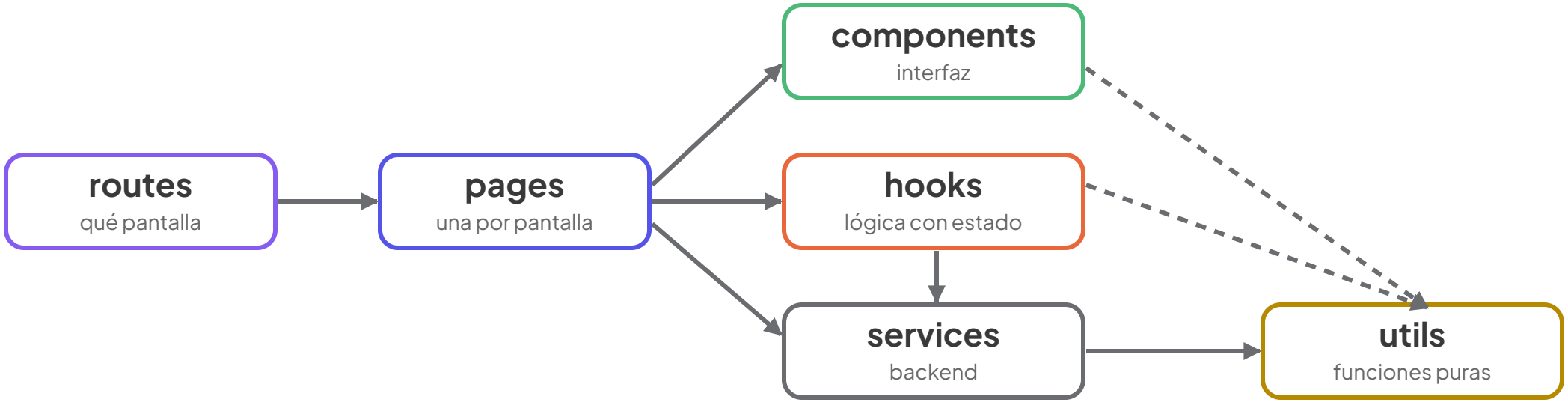
utils/

`formatPrice.js` : funciones puras, sin React.

routes/

`AppRoutes.jsx` : qué pantalla va en cada URL.

Quién usa a quién



La dependencia va en un solo sentido: **routes** usa pages; pages usa components, hooks y services; hooks usa services; todos pueden usar utils.

Universidad ICESI

7. Tu primer componente

Anatomía de un archivo .jsx

Un componente completo

ProductCard.jsx

```jsx
function ProductCard({name, price}) {
  return (
    <article className="card">
      <h3>{name}</h3>
      <p>Precio: ${price}</p>
    </article>
  );
}

export default ProductCard;
```

El archivo .jsx
Contiene JSX

La función
Nombre en mayúscula

Los parámetros
Las props

El return
Lo que se dibuja

El export
Para usarlo en otro archivo

El archivo .jsx y la función

ProductCard.jsx

```jsx
function ProductCard({name, price}) {
  return (
    <article className="card">
      <h3>{name}</h3>
      <p>Precio: ${price}</p>
    </article>
  );
}

export default ProductCard;
```

Extensión .jsx

Indica que el archivo mezcla JavaScript con etiquetas parecidas a HTML (JSX). Vite solo procesa JSX en .jsx o .tsx.

Un componente es una función

Un archivo por componente, con el mismo nombre.

Nombre con mayúscula

Así React distingue `<ProductCard />` de `<article>`.

Las props llegan como parámetro

ProductCard.jsx

```jsx
function ProductCard({name, price}) {
  return (
    <article className="card">
      <h3>{name}</h3>
      <p>Precio: ${price}</p>
    </article>
  );
}

export default ProductCard;
```

Un solo objeto
React llama a la función con las props que le pasó el padre.

Desestructuración
`{ name, price }` saca las dos propiedades directamente.

Solo lectura
El componente las usa, no las modifica.

return y JSX: lo que se dibuja

ProductCard.jsx

```jsx
function ProductCard({name, price}) {
  return (
    <article className="card">
      <h3>{name}</h3>
      <p>Precio: ${price}</p>
    </article>
  );
}

export default ProductCard;
```

return
Lo que devuelve la función es la interfaz que React dibuja; un solo elemento raíz.

JSX
Etiquetas como en HTML, pero son JavaScript. Toda etiqueta se cierra.

className, no class
`class` es una palabra reservada de JavaScript.

Las llaves abren JavaScript dentro del JSX

ProductCard.jsx

```jsx
function ProductCard({name, price}) {
  return (
    <article className="card">
      <h3>{name}</h3>
      <p>Precio: ${price}</p>
    </article>
  );
}

export default ProductCard;
```

Una expresión

Dentro de `{ }` se evalúa y React dibuja su resultado: `{name}` es Teclado.

El $ es texto

Lo dinámico es `{price}`. También puedes calcular: `{price * 2}`.

Sin llaves

El texto es solo texto: `name` se dibujaría tal cual.

export: para poder usarlo

ProductCard.jsx

```jsx
function ProductCard({name, price}) {
  return (
    <article className="card">
      <h3>{name}</h3>
      <p>Precio: ${price}</p>
    </article>
  );
}

export default ProductCard;
```

export default
Permite que otros archivos importen el componente.

En otro archivo

```jsx
import ProductCard from "./ProductCard.jsx"
```

Universidad ICESI

Qué permiten las llaves en JSX

Dónde	Ejemplo	Qué hace
En el contenido	`<h3>{name}</h3>`	Dibuja el valor de la variable
Con cálculos	`<p>{price * 2}</p>`	Dibuja el resultado de una operación
Con funciones	`<p>{formatPrice(price)}</p>`	Dibuja lo que devuelve la función
Con condiciones	`{stock > 0 ? "Disponible" : "Agotado"}`	Elige qué dibujar (ternario)
En un atributo	`<ProductCard price={20} />`	Pasa un valor que no es texto

Dentro de las llaves caben **expresiones**, no sentencias como `if` o `for`.

Universidad ICESI

8. Props

Pasar datos de padre a hijo

Las props son los datos que recibe un componente

Cómo se usa

```
<ProductCard name="Teclado" price={20} />
```

Lo que recibe la función

```
{ name: "Teclado", price: 20 }
```

Se escriben como atributos
Texto entre comillas; cualquier otro valor entre llaves.

Un mismo componente
Dibuja resultados distintos según sus props.

Solo lectura
El hijo las usa, no las modifica.

Una lista: el padre crea un hijo por producto

ProductList.jsx

```jsx
function ProductList() {
  const products = [
    { id: 1, name: "Teclado", price: 20 },
    { id: 2, name: "Mouse", price: 10 },
    { id: 3, name: "Monitor", price: 15 },
  ];

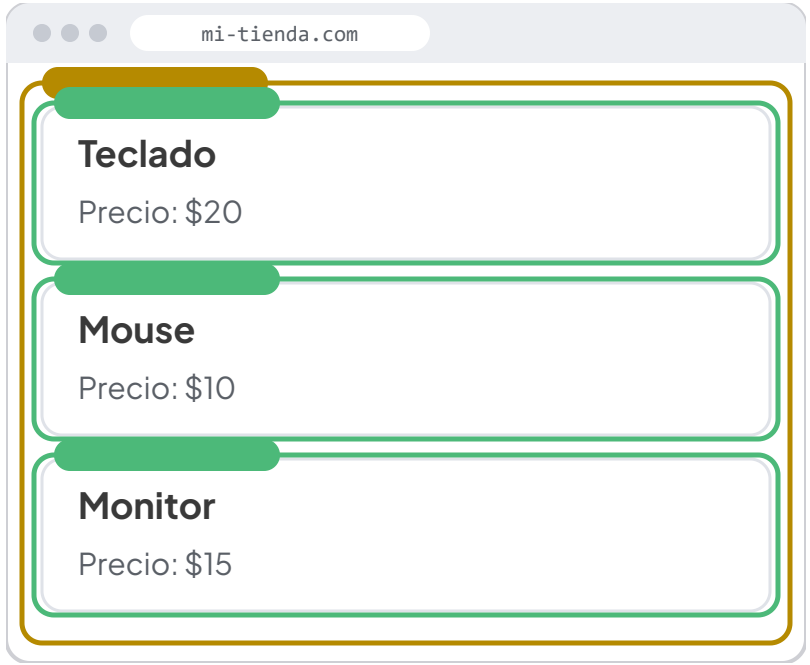
  return (
    <div>
      {products.map((p) => (
        <ProductCard key={p.id} name={p.name} price={p.price} />
      ))}
    </div>
  );
}
```

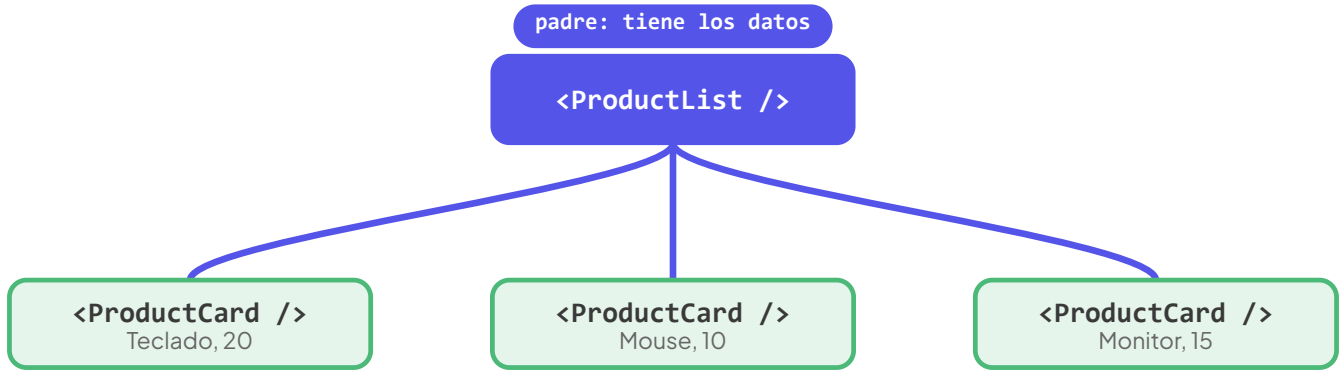
map recorre la lista
Devuelve un ProductCard por producto.

name y price
Son las props de cada tarjeta.

key
Ayuda a React a identificar cada elemento.

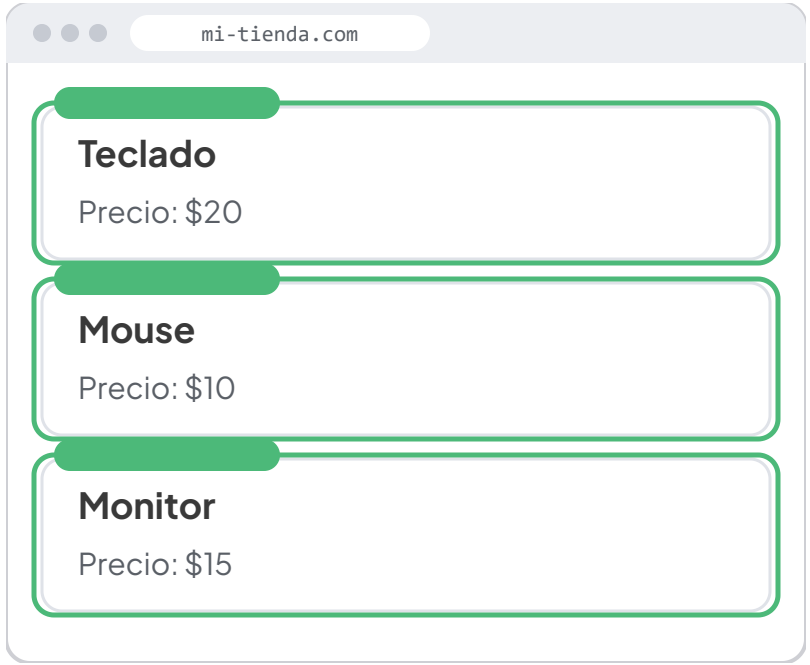
ProductList le pasa a cada ProductCard sus props

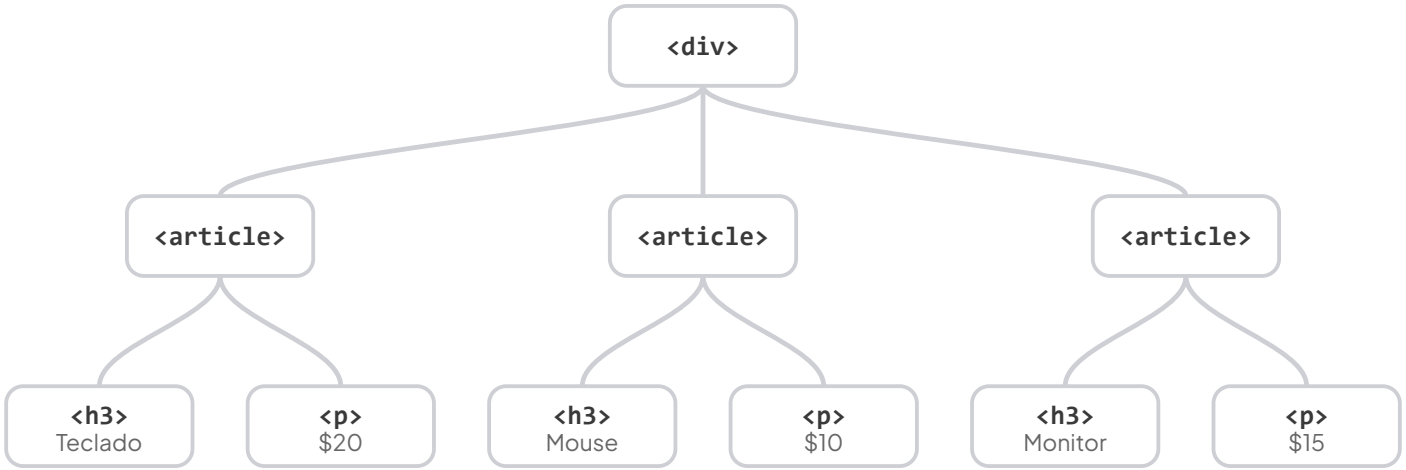




El **padre** tiene los datos y renderiza un ProductCard por producto. Cada hijo recibe **sus propias props**: los datos viajan hacia abajo.

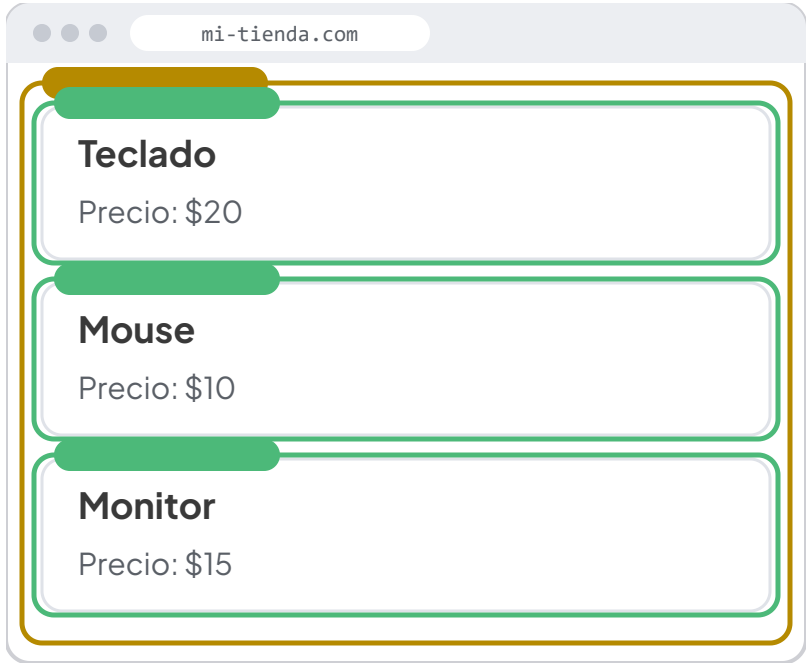
Las props se convierten en pantalla

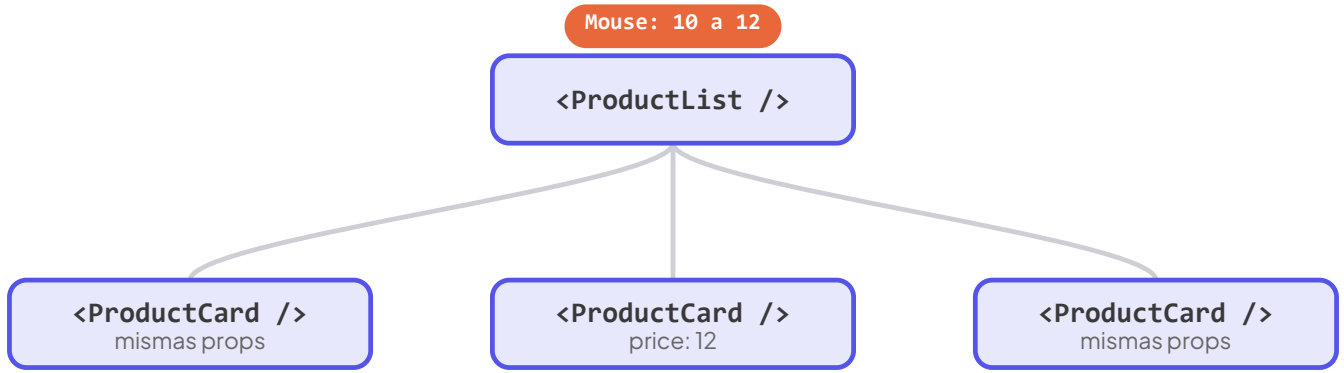




Cada ProductCard usa sus props para dibujar su `h3` y su `p`: el mismo componente, **tres resultados distintos**.

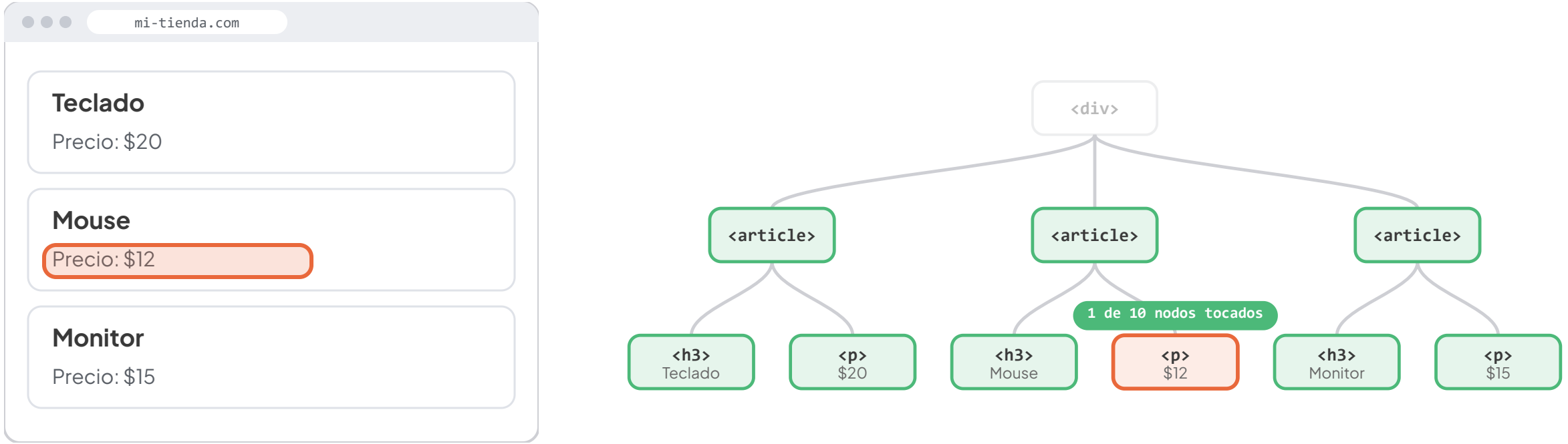
El padre cambia un dato y React vuelve a ejecutar





El precio del Mouse pasa de 10 a 12. **Los hijos no pueden cambiar sus props**: solo el padre decide. Si el padre se renderiza, sus hijos también.

Solo cambia el DOM de lo que recibió props distintas



React compara el resultado y solo actualiza el DOM donde algo cambió; los otros dos ProductCard recibieron las mismas props, **su DOM no se toca**. Los datos viajan en un solo sentido: del padre al hijo.

Resumen del capítulo

El DOM

Un árbol de nodos que JavaScript modifica; hacerlo a mano no escala.

CSR

El servidor entrega ingredientes crudos y el navegador cocina la página.

Virtual DOM

Disparo, render, diff y commit: solo se toca lo necesario.

Componentes

Funciones que devuelven JSX; se reutilizan y se componen.

Vite

Transforma, resuelve imports, HMR y build; el proyecto se organiza en src/.

Props

Datos de padre a hijo, de solo lectura: si cambian, solo cambia lo afectado.