## VITE COMO ENTORNO MODERNO DE DESARROLLO

### ¿Por qué necesitamos una herramienta como Vite?

Antes de construir una aplicación React, necesitamos comprender que React no se trabaja normalmente como un archivo HTML con un `<script>` agregado al final. Si bien técnicamente es posible incorporar React en una página existente, el desarrollo moderno con React suele requerir un entorno que permita:

- organizar el código en múltiples archivos;
- utilizar módulos;
- trabajar con JSX;
- instalar dependencias;
- ejecutar un servidor local;
- recargar la aplicación automáticamente cuando cambia el código;
- preparar el proyecto para producción.

Aquí aparece Vite. Vite es una herramienta moderna de desarrollo frontend. Su función no es reemplazar a React, sino crear y ejecutar un entorno adecuado para desarrollar aplicaciones modernas. La documentación oficial de React recomienda usar una herramienta de construcción como Vite, Parcel o Rsbuild cuando se quiere construir una aplicación React desde cero, porque estas herramientas permiten empaquetar y ejecutar el código, brindar un servidor de desarrollo local y generar una versión lista para producción.

El siguiente cuadro permite establecer la función de diversas tecnologías a la hora de crear una aplicación React:

| Herramienta | Función |
|---|---|
| **React** | Biblioteca para construir interfaces |
| **Vite** | Herramienta para crear, ejecutar y preparar el proyecto |
| **Npm** | Gestor de paquetes |
| **Node.js** | Entorno necesario para ejecutar herramientas de desarrollo |

*Vite no es React. Vite es el entorno que nos permite trabajar con React de manera moderna, organizada y eficiente.*

### ¿Qué problema resuelve Vite?

Si escribimos una aplicación sencilla con HTML, CSS y JavaScript puro, podemos abrir el archivo HTML en el navegador y ver el resultado. Pero React introduce necesidades adicionales.

Por ejemplo:

```
const App = () => {
    return <h1>Hola React</h1>;
};
```

Ese código contiene JSX. El navegador no entiende JSX directamente como entiende HTML común. JSX debe ser transformado en JavaScript válido.

## ¿Qué es JSX?

JSX es una extensión de JavaScript que permite escribir estructuras parecidas a HTML dentro del código.

No es HTML real: es una forma más cómoda de describir la interfaz que React transformará en JavaScript puro.

Dado un simple ejemplo como el siguiente:

```
const elemento = <h1>Hola mundo</h1>;
```

Implica las siguientes consideraciones:

- El browser no entiende este código.
- React tampoco lo ejecuta directamente
- En consecuencia, primero debe ser transformado

JSX es la manera que propone React para escribir interfaces de usuarios aplicando el modelo mental declarativo. Los desarrolladores de React asumen que sería difícil e incómodo escribir y leer interfaces pensando únicamente en que se desea visualizar en pantalla y cuando debe renderizarse si únicamente se aplica Javascript puro.

En Javascript un ejemplo como este:

```
const elemento = React.createElement("h1", null, "Hola mundo");
```

Con JSX se escribe de esta manera:

```
const elemento = <h1>Hola mundo</h1>;
```

Lo cual, según los desarrolladores de React interpretan que es mucho más claro de interpretar, manipular e incluso enseñar.

Considerando estos aspectos, JSX está diseñado para:

- hacer el código más declarativo
- parecerse al HTML, que los desarrolladores ya conocen
- describir la UI como una función del estado
- evitar escribir `createElement()` a mano

En conclusión, JSX permite la aplicación del modelo mental declarativo que propone React para crear las interfaces de usuario de una aplicación web.

## ¿Cómo se ejecuta realmente JSX?

Como hemos mencionado anteriormente, el browser no interpreta JSX. Es necesario realizar una transformación previa del código JSX. Herramientas como Babel o Vite realizan esta transformación automáticamente. React utiliza ese JavaScript resultante para crear y actualizar la interfaz de usuario.

## Node.js, npm y package.json

Para usar Vite necesitamos tener instalado Node.js. Node.js permite ejecutar herramientas JavaScript fuera del navegador. Junto con Node.js se instala normalmente npm, que es el gestor de paquetes que utilizaremos para instalar dependencias.

Una dependencia es una biblioteca o herramienta que el proyecto necesita para funcionar. En este caso, algunas dependencias importantes serán React, React DOM y Vite.

Cuando se crea un proyecto con Vite aparece un archivo llamado:

`package.json`

Este archivo describe el proyecto. Allí se registran:

- nombre del proyecto;
- scripts disponibles;
- dependencias;
- dependencias de desarrollo.

Y su estructura interna será similar a la siguiente:

```
{
  "scripts": {
    "dev": "vite",
    "build": "vite build"
  },
  "dependencies": {
    "react": "...",
    "react-dom": "..."
  },
  "devDependencies": {
    "vite": "..."
  }
}
```

Donde:

- La sección `scripts` define comandos que podemos ejecutar con `npm`. Por ejemplo:
  `npm run dev`
  ejecuta el script llamado `dev`, que en un proyecto Vite inicia el servidor de desarrollo.
- La sección `dependencies` contiene bibliotecas necesarias para que la aplicación funcione.
- La sección `devDependencies` contiene herramientas necesarias durante el desarrollo.

Existen otras secciones útiles que podrían ser abordadas más adelante.

### Creación del proyecto con Vite

Para crear el proyecto se utiliza:

```
npm create vite@latest
```

Este comando despliega un conjunto de opciones para crear proyectos. En nuestro caso debemos seleccionar las opciones Framework: React y Variant: Javascript

Posteriormente debe ingresarse a la carpeta creada. Desde la línea de comandos es posible utilizar

```
cd nombre-del-proyecto
```

En este punto utilizamos `npm` para instalar las dependencias del proyecto React. Para ello utiliza la información que creó el generador de proyectos en el archivo `package.json`. El comando por utilizar es

```
npm install
```

Y finalmente, para ejecutar el proyecto dentro de un servidor local se utiliza

```
npm run dev
```

### Desglosando la función de los comandos para generar proyectos React con Vite

Cuando se crea un proyecto con Vite, la estructura es generada a partir de plantillas básicas. Vite ofrece varias plantillas disponibles de ser usadas cuando se utiliza el comando `create-vite`.

Una vez creado el proyecto, es posible verificar que se han generados los archivos de configuración, pero las dependencias aún no están descargadas. Por ese motivo es que se utiliza

```
npm install
```

Este comando lee el archivo `package.json`, descarga los paquetes necesarios y crea la carpeta `node_modules/`. Esa carpeta contiene todas las dependencias instaladas.

---

**Nota importante**: La carpeta `node_modules` no debe subirse normalmente al repositorio GitHub. Es una carpeta pesada y puede reconstruirse ejecutando nuevamente `npm install`. Por eso el proyecto debe incluir un archivo `.gitignore` que ignore `node_modules`.

Una vez creado el proyecto, conviene registrar el primer estado con Git

```
git init
git add .
git commit -m "feat: crea proyecto react con vite"
```

Luego, si el repositorio remoto ya existe:

```
git remote add origin URL_DEL_REPOSITORIO
git push -u origin main
```

Este primer commit representa el punto inicial del proyecto. No contiene todavía lógica propia de la aplicación, pero sí la estructura base sobre la cual se trabajará.

---

El comando:

```
npm run dev
```

inicia el servidor local de desarrollo. Esto permite abrir la aplicación en una dirección similar a:

```
http://localhost:5173
```

Ese servidor no es el servidor definitivo de producción. Es un servidor para desarrollo local. Permite probar la aplicación mientras se escribe código.

Vite inicia un servidor de desarrollo usando el directorio actual como raíz del proyecto.

### Estructura inicial de React con Vite

Una vez creado el proyecto, aparece una estructura similar a esta:

```
mi-proyecto/
│
├── index.html
├── package.json
├── node_modules/
├── public/
└── src/
    ├── App.jsx
    ├── main.jsx
    └── assets/
```

No todos estos archivos tienen la misma función. Para comprender React, es fundamental entender qué papel cumple cada uno.

- **index.html y el div root**. En un proyecto React con Vite, el archivo `index.html` sigue existiendo. Sin embargo, no contiene toda la interfaz de la aplicación. Contiene principalmente un punto de montaje. Por ejemplo:

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

  La primera línea define el contenedor donde React insertará la aplicación, mientras que la segunda línea indica que el punto de entrada de Javascript será `main.jsx`. Esto da la idea de que la interfaz se construye desde Javascript, para lo cual utilizará componentes.
- **main.jsx**. Suele tener una estructura como esta:

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

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

  La segunda línea importa la función que permite crear la raíz de React dentro del DOM.

La tercera línea importa el componente principal de la aplicación

`document.getElementById("root")`

Busca en el HTML el contenedor donde se montará React.

Finalmente `<App />`, permite observar que React permite insertar los componentes funcionales mediante una notación de marcado, esto es, expresarlos como etiquetas. Recuerda que este es el componente principal.

- **App.jsx**. Aloja al componente principal. Suelte tener la siguiente estructura

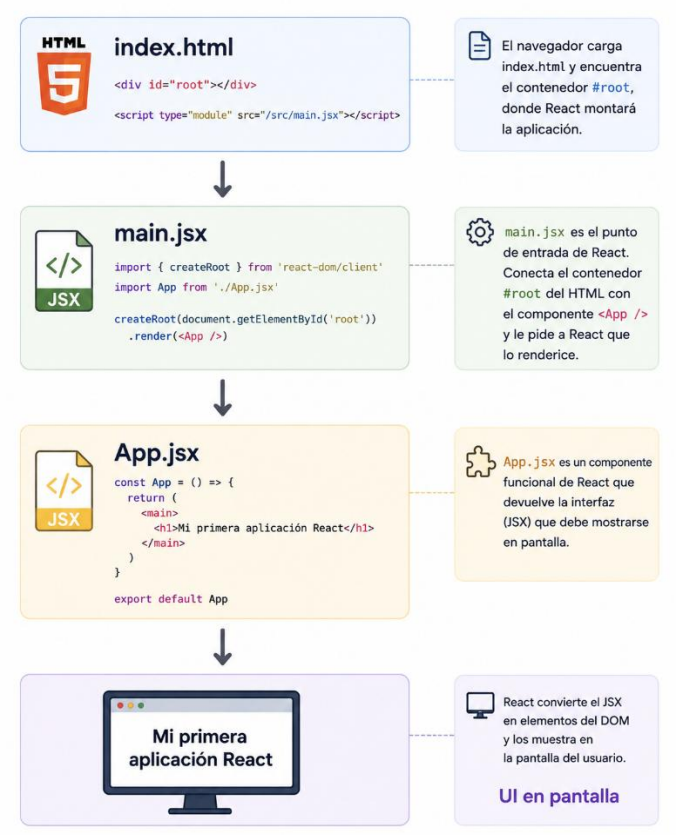
```
const App = () => {
    return (
        <main>
            <h1>Mi primera aplicación React</h1>
        </main>
    );
};

export default App;
```

Esto define un componente funcional que mediante la instrucción `return` devuelve la interfaz del componente.

La última línea exporta el componente, por lo que puede ser importado desde otro archivo.

## El flujo inicial de una aplicación React

Puede expresarse gráficamente de la siguiente manera:




El navegador carga `index.html`. Ese archivo carga `main.jsx`. React toma el componente `App` y lo renderiza dentro del `div` con id root.

## JSX

En las secciones anteriores hemos introducido los conceptos y funcionamiento esenciales de JSX. En esta sección lo ampliaremos. JSX es una sintaxis que permite escribir estructuras similares a HTML dentro de JavaScript.

Ejemplo:

```
const App = () => {
    return (
        <h1>Hola React</h1>
    );
};
export default App;
```

A primera vista parece HTML, pero no es HTML puro. JSX se transforma en JavaScript para que React pueda interpretarlo.

### ¿Qué problema resuelve JSX?

Sin JSX, construir interfaces sería más verboso. JSX permite expresar visualmente la estructura de la interfaz dentro del componente.

En lugar de separar completamente estructura y lógica, React propone una unidad llamada componente, donde conviven:

- estructura visual;
- datos;
- comportamiento;
- estado.
- Idea clave

JSX permite describir la UI desde JavaScript.

### Expresiones con llaves

En JSX se pueden insertar expresiones JavaScript usando llaves. Observe este ejemplo:

```
const nombre = "Ariel";

const App = () => {
    return (
        <h1>Hola {nombre}</h1>
    );
};
```

Lo que está dentro de {} se evalúa como JavaScript. Otro ejemplo:

```
const edad = 20;

<p>Edad: {edad}</p>
```

Pero debe considerar que dentro de las llaves debe colocarse una expresión, no una sentencia completa. Entonces podrías pensar en complementar el ejemplo anterior de la siguiente manera:

```
<p>{edad >= 18 ? "Mayor" : "Menor"}</p>
```

Lo cual sería correcto, pero no de esta otra forma:

```
<p>{if (edad >= 18) { "Mayor" }}</p>
```

Ya que sería incorrecto porque no puedes escribir entre llaves una sentencia completa.

### Diferencias con HTML

Aunque JSX se parece visualmente a HTML, tiene reglas propias, ya que en fondo es Javascript. Por eso, algunos atributos cambian de nombre y otros deben escribirse en camelCase. La siguiente tabla expresa una comparación entre HTML y JSX:

| Concepto | HTML | JSX | Explicación |
|---|---|---|---|
| **Atributo de clase** | class | className | En JSX, class es una palabra reservada de JS, por eso se usa className. |
| **Atributo de etiqueta <label>** | for | htmlFor | for también es palabra reservada en JS, por eso JSX usa htmlFor. |
| **Eventos** | onclick | onClick | JSX usa camelCase para eventos porque son propiedades JS. |
| **Estilos inline** | style="color: red" | style={{ color: "red" }} | En JSX, style recibe un **objeto JS**, no un string. |
| **Atributos booleanos** | disabled | disabled={true} o simplemente disabled | JSX permite expresiones JS dentro de {}. |
| **Comentarios** | <!— comentario --> | {/* comentario */> | JSX usa sintaxis de comentario de JS dentro de llaves |

Por ejemplo, en HTML tradicional para representar una un `label` y un `input` tendríamos:

```
<label for="nombre">Nombre</label>

<input id="nombre" class="campo" onclick="enviar()" />
```

El JSX equivalente es:

```
<label htmlFor="nombre">Nombre</label>
<input
  id="nombre"
  className="campo"
  onClick={enviar}
/>
```

Podemos observar las siguientes diferencias en este ejemplo comparativo:

- for → htmlFor
- class → className

- onclick="enviar()" → onClick={enviar} (sin comillas, porque es una función JS)
- JSX permite saltos de línea y formato más legible

### JSX debe devolver un único contenedor

Esto no es válido:

```
return (
    <h1>Título</h1>
    <p>Texto</p>
);
```

Debe envolverse mediante un contenedor, por ejemplo un `div`:

```
return (
    <div>
        <h1>Título</h1>
        <p>Texto</p>
    </div>
);
```

O usar un fragmento:

```
return (
    <>
        <h1>Título</h1>
        <p>Texto</p>
    </>
);
```

Esto se debe a que nn componente debe devolver una única estructura raíz.

## COMPONENTES FUNCIONALES

Formalmente, un componente funcional es una función que devuelve JSX. Observe el siguiente ejemplo:

```
const Encabezado = () => {
    return (
        <header>
            <h1>Programación Visual</h1>
        </header>
    );
};

export default Encabezado;
```

React permite construir interfaces a partir de piezas individuales llamadas componentes.

### ¿Qué problema resuelven los componentes funcionales?

Si toda la aplicación se escribiera en un solo archivo, rápidamente sería difícil de leer, mantener y corregir. Los componentes permiten separar la interfaz en partes.

Por ejemplo:

```
App
├── Encabezado
├── Formulario
├── ListaTareas
└── PiePagina
```

Cada componente tiene una responsabilidad.

### Composición de componentes

Un componente puede usar otros componentes, por ejemplo:

```
const App = () => {
    return (
        <main>
            <Encabezado />
            <Formulario />
            <ListaTareas />
        </main>
    );
};
```

Esto es un ejemplo claro de composición, tenemos que el componente `App`, está compuesto por los componentes `Encabezado`, `Formulario` y `ListaTareas`, que antes habíamos organizado en la sección anterior.

La idea central es que una aplicación React se construye componiendo componentes pequeños.

### Responsabilidad única

Un componente debería tener una responsabilidad clara. Un mal enfoque consiste en tener un componente App que haga todo lo que corresponde a:

- formulario
- lista
- validaciones
- estilos
- cálculos
- navegación

Un mejor enfoque sería que App coordine otros componentes que realicen su tarea:

- Formulario captura datos
- ListaTareas muestra colección
- Tarea muestra un elemento

La separación de responsabilidades mejora la comprensión y el mantenimiento.

### Props

Los props (abreviatura de properties) son datos que un componente recibe desde otro componente.

Son la forma en que React permite configurar, personalizar y comunicar información entre componentes.

## ¿Qué problema resuelven las props?

Sin props, un componente sería estático, cerrado y no reutilizable. Observe el sigueite ejemplo de componente rígido (sin props)

```
const Tarjeta = () => {
  return (
    <article>
      <h2>React</h2>
      <p>Biblioteca frontend</p>
    </article>
  );
};
```

Este componente siempre muestra lo mismo. No sirve para mostrar otra cosa sin modificar su código. Con props, el componente se vuelve flexible:

```
const Tarjeta = ({ titulo, descripcion }) => {
  return (
    <article>
      <h2>{titulo}</h2>
      <p>{descripcion}</p>
    </article>
  );
};
```

Ahora el componente **se adapta** a los datos que recibe:

```
<Tarjeta titulo="HTML" descripcion="Estructura" />
<Tarjeta titulo="CSS" descripcion="Presentación" />
<Tarjeta titulo="JavaScript" descripcion="Comportamiento" />
```

Las props convierten un componente fijo en un componente configurable y reutilizable.

## Comunicación entre padre e hijo

Las props viajan siempre en un solo sentido: desde el componente padre hacia el componente hijo.

```
App
 ↓ props
Tarjeta
```

Este es un ejemplo visual de este concepto:

```
const App = () => {
  return (
    <div>
      <Tarjeta titulo="React" descripcion="UI declarativa" />
    </div>
  );
};
const Tarjeta = ({ titulo, descripcion }) => {
  return (
    <article>
      <h2>{titulo}</h2>
      <p>{descripcion}</p>
    </article>
  );
};
```

## ¿Cómo se leen las props?

Hay dos formas:

- Forma 1: Acceder desde props

```
const Tarjeta = (props) => {
  return (
    <article>
      <h2>{props.titulo}</h2>
      <p>{props.descripcion}</p>
    </article>
  );
};
```

- Forma 2: La desestructuración (más usado)

```
const Tarjeta = ({ titulo, descripcion }) => {
  return (
    <article>
      <h2>{titulo}</h2>
      <p>{descripcion}</p>
    </article>
  );
};
```

## Características importantes de los props

1. **Props son solo lectura**. Un componente no puede modificar sus props.
   `props.titulo = "Nuevo título"; // Error conceptual`
   Esto es intencional. React quiere que los componentes sean **predecibles** y que los datos fluyan en un solo sentido. Los props no se cambian, simplemente se utilizan.
2. **Props pueden ser cualquier tipo de dato**. React permite pasar: strings, números, booleanos, arrays, objetos, funciones, incluso otros componentes. Ejemplo pasando una función:
   `<Boton onClick={handleClick} />`

## Los Props.children

Cuando un componente envuelve contenido, ese no se pierde. React lo guarda automáticamente en una prop especial llamada `children`.

Es decir, en este ejemplo:

```
<Caja>
  <p>Hola mundo</p>
</Caja>
```

Todo lo que está entre `<Caja>` y `</Caja>` se convierte en el valor de `children`. El componente se convierte en una caja que recibe contenido. Observe este otro ejemplo clarificador:

```
const Caja = ({ children }) => {
  return <div className="caja">{children}</div>;
};
```

Aquí, `children` es lo que pongas adentro del componente. Por ejemplo:

```
<Caja>
  <p>Hola mundo</p>
</Caja>
```

React lo interpreta como:

```
<Caja children={<p>Hola mundo</p>} />
```

¿Por qué existe esto?

Porque te permite crear componentes contenedores, que no saben qué contenido van a mostrar, pero sí saben cómo mostrarlo.

Ejemplo clásico: un panel, una tarjeta, un modal, un layout.

Veamos un ejemplo un poco más avanzado y útil:

```
const Boton = ({ children }) => {
  return (
    <button style={{ padding: "10px", background: "blue", color:
"white" }}>
      {children}
    </button>
  );
};
```

El componente Boton se vuelve reutilizable y su uso es más flexible. Por ejemplo:

Por ejemplo, lo siguiente:

```
<Boton>Guardar</Boton>
<Boton><strong>Eliminar</strong></Boton>
<Boton><IconoCheck /> Aceptar</Boton>
```

Indica que hemos usado el mismo componente para cualquier contenido.

## ESTADO CON useState

El estado representa información que puede cambiar durante la vida del componente. React define `useState` como un Hook que permite agregar una variable de estado a un componente. El Hook devuelve una variable de estado y una función para actualizarla, lo que permite conservar datos entre renders y disparar nuevos renderizados. Por ejemplo:

```
import { useState } from "react";

const Contador = () => {
    const [contador, setContador] = useState(0);

    return (
        <div>
            <p>{contador}</p>

            <button onClick={() => setContador(contador + 1)}>
                Incrementar
            </button>
        </div>
    );
};
```

La línea

```
const [contador, setContador] = useState(0);
```

combina varias ideas:

- *useState*(0) crea un estado con valor inicial 0.
- contador es el valor actual.
- *setContador* es la función que actualiza el estado.
- La sintaxis [contador, setContador] usa desestructuración de arreglo.

La idea clave de esta instrucción es dar memoria a un componente. Esto se logra con useState.

Si se actualiza el estado React vuelve a renderizar el componente. En el ejemplo anterior esto se logra con la instrucción:

```
setContador(contador + 1);
```

Es decir, usamos la función indicada en la definición de la variable de estado para actualizarlo y provocar la renderización automática. No debemos utilizar:

```
contador = contador + 1;
```

porque eso no informa a React que debe actualizar la interfaz. En React no se modifica directamente el estado; se solicita su actualización mediante la función correspondiente.

### La inmutabilidad

Cuando el estado es un arreglo u objeto, se debe crear una nueva versión. Ejemplo con arreglo:

```
setTareas([...tareas, nuevaTarea]);
```

Ejemplo con objeto:

```
setUsuario({
    ...usuario,
    nombre: "Nuevo nombre"
});
```

Cuando un componente envuelve contenido, ese no se pierde. React lo guarda automáticamente en una prop especial llamada children.

Esto se relaciona directamente con el spread operator. Por este motivo, el enfoque mental de React establece que actualizar un estado suele significar crear una nueva versión del dato.

## EVENTOS EN REACT

Los eventos permiten que un componente reaccione a acciones del usuario: clics, teclas, cambios en inputs, etc.

En React:

- Los nombres de eventos usan camelCase → onClick, onChange, onSubmit

- Los eventos reciben funciones, no strings
- Las funciones se pasan sin ejecutarlas

La ultima característica mencionada es muy importante. Observe el siguiente ejemplo:

```
<button onClick={manejarClick}>
  Click
</button>
```

Aquí no se ejecuta `manejarClick`. Solo se pasa la referencia a la función. React la ejecutará cuando ocurra el evento.

Por tal motivo, la siguiente expresión es incorrecta:

```
<button onClick={manejarClick()}>
  Click
</button>
```

Porque lo que provoca es que se ejecuta *`manejarClick()`* inmediatamente, durante el render. El resultado de esa ejecución (probablemente undefined) es lo que se pasa al evento.

Por eso:

- Se ejecutará en el momento equivocado
- No espera al clic
- Y puede romper el componente

### Eventos con parámetros

A veces necesitarás pasar un dato al evento, por ejemplo, el id de una tarea. Dicho esto, no puedes hacer lo siguiente:

```
onClick={eliminarTarea(tarea.id)} // ejecuta la función
```

Entonces se debe usar una función flecha que envuelve la llamada:

```
<button onClick={() => eliminarTarea(tarea.id)}>
  Eliminar
</button>
```

¿Por qué funciona?

- La función flecha no se ejecuta
- React la ejecuta cuando ocurra el clic
- Dentro de esa función flecha, estarás invocando a `eliminarTarea`

React recibirá algo así:

```
onClick={() => eliminarTarea(3)}
```

Por lo que cuando el usuario hace clic:

1. React ejecuta la función flecha
2. La función flecha ejecuta *`eliminarTarea(3)`*

Veamos un ejemplo más integrador:

```
const BotonEliminar = ({ id, eliminar }) => {
  return (
    <button onClick={() => eliminar(id)}>
      Eliminar
    </button>
  );
};
```

Este componente es usado por un componente de la siguiente manera:

```
<BotonEliminar id={tarea.id} eliminar={eliminarTarea} />
```

¿Qué pasaría si quiero acceder al evento? React pasa el evento como primer parámetro:

```
<button onClick={(e) => console.log(e.target)}>
  Ver evento
</button>
```

Por lo que si se necesita enviar el evento y el parámetro debes realizarlo de la siguiente manera:

```
<button onClick={(e) => eliminarTarea(id, e)}>
  Eliminar
</button>
```

## LOS FORMULARIOS CONTROLADOS

Un formulario controlado es aquel cuyo valor depende del estado de React. Observa el siguiente ejemplo:

```
import { useState } from "react";

const Formulario = () => {
    const [texto, setTexto] = useState("");

    return (
        <input
            type="text"
            value={texto}
            onChange={(e) => setTexto(e.target.value)}
        />
    );
};
```

Como puedes notar el valor del input proviene de la variable de estado `texto`. Mientras que

```
onChange={(e) => setTexto(e.target.value)}
```

Provoca que cada vez que el usuario escriba en el `input`, se actualice el estado.

Por lo tanto, en un formulario controlado, React controla el valor del campo mediante estado.

## RENDERIZADO DE LISTAS

React permite renderizar listas usando `map`. Observa el siguiente ejemplo:

```
const tecnologias = ["HTML", "CSS", "JavaScript", "React"];

const Lista = () => {
    return (
        <ul>
            {tecnologias.map((tec) => (
                <li key={tec}>{tec}</li>
            ))}
        </ul>
    );
};
```

`map` transforma cada elemento del arreglo en otro valor. En React, se usa para transformar datos en componentes visuales.

### Uso de keys

Observe el siguiente ejemplo:

```
tareas.map((tarea) => (
    <Tarea key={tarea.id} tarea={tarea} />
));
```

React necesita una identificación única para cada elemento renderizado. Esto le permite saber qué elemento cambió, cuál se agregó y cuál se eliminó. Para ello utiliza la propiedad `key`.

En resumen, `map` transforma una colección de datos en una colección de elementos visuales. Y cada uno de esos elementos visuales tiene un identificador único.

## LOS HOOKS

Un Hook es una función especial de React que permite "enganchar" (hook into) características internas del framework desde un componente funcional.

React trae varios Hooks, pero los dos más importantes para empezar son:

- useState → manejar estado
- useEffect → sincronizar con sistemas externos

Sin Hooks, los componentes funcionales serían estáticos. Con Hooks, pueden tener estado, efectos, lógica reutilizable y más.

### useEffect

Es el Hook que permite sincronizar un componente con sistemas externos. La documentación oficial lo define así:

*useEffect sincroniza tu componente con un sistema externo.*

Esto significa que useEffect sirve para todo lo que no ocurre dentro del renderizado puro.

El renderizado de React es puro: si le das las mismas props y estado, produce la misma UI.

Pero en la vida real, un componente necesita hacer cosas que no son parte del renderizado, por ejemplo:

- llamar a una API
- leer o escribir en localStorage
- configurar un temporizador
- escuchar un evento del navegador
- suscribirse a un WebSocket
- manipular el DOM (casos puntuales)

Todo eso es efecto secundario (side effect). Y React necesita un lugar seguro para ejecutarlos. Ese lugar es useEffect.

Veamos un ejemplo básico:

```
useEffect(() => {
  console.log("Componente montado");
}, []);
```

Este ejemplo indica que cuando el compoenente sea montado visualice en la consola un mensaje. La clave como verás en la siguiente sección es el segundo parámetro que recibe la función flecha.

**El arreglo de dependencias**

El segundo parámetro de useEffect controla cuándo se ejecuta el efecto. Existen diversos casos:

- Caso 1: arreglo vacío → se ejecuta una sola vez (montaje). Es el caso del ejemplo anterior. Útil para: cargar datos iniciales, configurar listeners, inicializar librerías externas.
- Caso 2: dependencias → se ejecuta cuando cambian. Por ejemplo:
  ```
  useEffect(() => {
    console.log("Cambió el contador");
  }, [contador]);
  ```
  Se ejecuta cada vez que contador cambia. Es Útil para: sincronizar datos, guardar en localStorage, actualizar el título del documento, reaccionar a cambios de props o estado.
- Caso 3: sin arreglo → se ejecuta en cada render. Por ejemplo:
  ```
  useEffect(() => {
    console.log("En cada render");
  });
  ```
  Casi nunca se usa. Puede generar problemas de rendimiento.

**Limpieza (cleanup)**

Cuando un componente:

- se desmonta (deja de existir en pantalla), o
- se vuelve a ejecutar el efecto porque cambió una dependencia

React te da la oportunidad de limpiar lo que dejaste abierto. El Cleanup es el proceso de deshacer lo que creaste dentro del efecto. Es como apagar la luz cuando salís de una habitación.

¿Por qué existe el cleanup?

Porque algunos efectos siguen funcionando, aunque el componente ya no esté:

- temporizadores (setInterval, setTimeout)
- listeners del navegador (addEventListener)
- suscripciones a APIs o sockets
- conexiones a bases de datos
- animaciones
- observadores (IntersectionObserver, MutationObserver)

Si no los limpiás, quedan "colgados" y siguen ejecutándose.

Ejemplo con un temporizador:

```
useEffect(() => {
  const id = setInterval(() => {
    console.log("tick");
  }, 1000);

  return () => clearInterval(id); // cleanup
}, []);
```

¿Qué pasa acá?

1. Cuando el componente se monta se crea un intervalo que hace "tick" cada segundo.
2. Si el componente se desmonta React ejecuta la función del return.
3. Esa función hace *clearInterval(id)* y el temporizador se detiene.

Esto es necesario porque sino el intervalo seguirá corriendo aunque el componente ya no exista generando:

- logs infinitos
- consumo de CPU
- errores difíciles de rastrear

Esto permite también realizar una advertencia sobre el uso de useEffect:

***No todo necesita useEffect.***

Si algo puede calcularse directamente desde:

- props
- estado
- funciones puras

entonces no requiere un efecto.