INTRODUCCIÓN: POR QUÉ REACT EXIGE VOLVER A MIRAR JAVASCRIPT

React no es un lenguaje de programación. React es una biblioteca de JavaScript orientada a la construcción de interfaces de usuario. Esto significa que React se apoya completamente en JavaScript moderno para expresar la estructura, el comportamiento y la actualización de una interfaz. Por eso, cuando se trabaja con React, no alcanza con "saber HTML" o "saber algo de JavaScript": es necesario comprender ciertas características actuales de ECMAScript, especialmente funciones flecha, módulos, desestructuración, spread/rest, asincronía y manejo de arreglos.

Durante mucho tiempo, el desarrollo web se enseñó como una separación bastante directa entre HTML, CSS y JavaScript. HTML definía la estructura, CSS definía la apariencia y JavaScript manipulaba el DOM para modificar la página. Ese enfoque sigue siendo válido para comprender la base del desarrollo web, pero cuando una aplicación crece, la manipulación manual del DOM se vuelve difícil de mantener. Aparecen problemas como duplicación de código, dificultad para sincronizar datos e interfaz, y estructuras de archivos poco escalables.

React propone otra forma de pensar la interfaz: en lugar de decirle al navegador paso a paso cómo modificar el DOM, el programador describe cómo debe verse la interfaz en función del estado actual de la aplicación. Cuando ese estado cambia, React actualiza la vista. Esta idea se conoce como renderizado declarativo.

React recomienda actualmente definir componentes como funciones, mientras que los componentes de clase siguen soportados, pero no son la forma recomendada para código nuevo.

En React no programamos la interfaz como una sucesión de modificaciones manuales del DOM. Programamos componentes que describen cómo debe verse la interfaz según los datos actuales.

JAVASCRIPT MODERNO COMO BASE DE REACT

Las últimas versiones de React se apoyan en características de JavaScript que forman parte del ecosistema ECMAScript moderno. Estas características no son accesorios: son la base para escribir componentes claros, reutilizables y mantenibles.

Funciones flecha

Una función tradicional en JavaScript puede escribirse así:

```
function sumar(a, b) {
    return a + b;
}
```

Con funciones flecha, puede escribirse de esta forma:

```
const sumar = (a, b) => {
    return a + b;
};
```

Y si la función devuelve directamente una expresión, puede escribirse con retorno implícito:

```
const sumar = (a, b) => a + b;
```

Una función flecha es una alternativa compacta a una expresión de función tradicional, aunque tiene diferencias semánticas y limitaciones que deben conocerse.

En esta materia vamos a utilizar funciones flecha porque se ajustan al enfoque de componentes funcionales. En React, un componente funcional no es otra cosa que una función que devuelve una descripción de interfaz. Ejemplo:

```
const Saludo = () => {
    return <h1>Hola estudiantes</h1>;
};
```

Este componente es una función. Su responsabilidad es devolver una porción de interfaz.

En React, una función puede representar comportamiento, cálculo o una parte reutilizable de la interfaz.

Desestructuración

La desestructuración permite extraer valores desde objetos o arreglos y asignarlos a variables de manera clara. MDN[1] define la desestructuración como una sintaxis que permite "desempaquetar" valores de arreglos o propiedades de objetos en variables distintas

Ejemplo con el uso de un objeto:

```
const usuario = {
    nombre: "Ariel",
    edad: 40,
    rol: "docente"
};

const { nombre, edad } = usuario;

console.log(nombre);
console.log(edad);
```

Sin desestructuración, deberíamos escribir:

```
const nombre = usuario.nombre;
const edad = usuario.edad;
```

En React se utiliza permanentemente para recibir `props`

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

[1] **MDN** significa **Mozilla Developer Network**. Es la documentación oficial más completa, clara y confiable sobre HTML, CSS, JavaScript y APIs web.

Aquí el parámetro del componente no se llama simplemente `props`, sino que se desestructura directamente. Esto significa que desde el objeto de propiedades se extraen `titulo` y `descripcion`.

La desestructuración permite escribir componentes más limpios porque evita acceder repetidamente a propiedades mediante notación punto.

Spread operator

El operador spread se escribe con tres puntos:

...

Permite expandir elementos de un arreglo u objeto. Por ejemplo, en el caso de un arreglo:

```
const numeros = [1, 2, 3];

const copia = [...numeros];
```

El término "expandir elementos de un arreglo u objeto" significa que en la creación de arreglos u objetos a partir de otros primero, se crea el arreglo u objeto, luego se "expande, es decir abre el arreglo u objeto indicado y extrae una copia de cada uno de los elementos". Con esto se lograr crear un nuevo arreglo u objeto con los mismos valores" evitando la copia de la referencia.

En el ejemplo, `numeros` es un arreglo, entonces:

1. …`numeros` toma cada elemento de ese arreglo y lo expande así: 1, 2, 3
2. Luego crea `copia` (otro arreglo),
3. `copia` recibe un nuevo arreglo con los valores expandidos.

Todo lo anterior sería equivalente a escribir:

```
const copia = [1, 2, 3];
```

¿Cuál es la utilidad del spread operator?

Permite crear copias reales (denominadas shallow copy), no referencias. La siguiente instrucción:

```
const copia = numeros;
```

Provoca que ambas variables apuntan al mismo arreglo. Modificar uno modifica el otro. Entonces si agrego un elemento en `numeros`, este también se agrega en `copia`, porque ambas variables referencian al mismo arreglo. En cambio, si aplicara esto con spread:

```
const copia = [...numeros];
copia.push(4);

console.log(numeros); // [1, 2, 3]
console.log(copia);   // [1, 2, 3, 4]
```

Como era de esperarse, el elemento con valor 4 se agregó únicamente a `copia`.

Esto permite tener una amplia variedad de acciones interesantes:

1. Combinar arreglos fácilmente:
```
const a = [1, 2];
const b = [3, 4];

const combinado = [...a, ...b];  // [1, 2, 3, 4]
```
2. Permite agregar muchos elementos sin mutar el original:
```
const numeros2 = [...numeros, 4, 5]; // [1, 2, 3, 4, 5]
```
3. Combinar objetos:
```
const persona = { nombre: "Ana", edad: 30 };
const copiaPersona = { ...persona, ciudad: "Jujuy" };
```

En React, el uso del spread operator es fundamental porque el estado no debe modificarse directamente. Si tenemos un arreglo de tareas, no conviene alterarlo manualmente con operaciones mutables, sino crear una nueva versión del arreglo. Para lo cual se suele hacer algo como esto:

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

Esto significa:

1. tomar todas las tareas anteriores,
2. crear un nuevo arreglo,
3. agregar la nueva tarea al final,
4. actualizar el estado con ese nuevo arreglo.

En definitiva:

En React, muchas actualizaciones se realizan creando nuevas versiones de los datos, no modificando directamente los datos existentes.

Rest operator

El rest operator también usa tres puntos, pero cumple otra función: agrupar múltiples valores.

Observe el siguiente ejemplo:

```
const sumar = (...numeros) => {
    return numeros.reduce((acumulador, numero) => acumulador + numero,
0);
};

console.log(sumar(1, 2, 3, 4));
```

En este caso, `...numeros` agrupa todos los argumentos recibidos en un arreglo. Al invocar a *`sumar`*`(1,2,3,4)`, los cuatro valores se agrupan en un arreglo denominado `numeros`. Hasta allí el uso del operador rest operator.

Recordemos que la función *`reduce`*`()` recorre un arreglo y aplica una función reductora que acumula un resultado en un único valor. En este caso la variable `acumulador` es inicializada en 0, y cada vez que va avanzando en el recorrido del arreglo aplica `acumulador + numero`, donde `numero` almacena el valor del elemento del arreglo.

Este patrón tiene diferentes utilidades, ya que es una forma elegante de crear funciones que aceptan cantidad variable de parámetros con diferentes objetivos:

1. Sumar cualquier cantidad de números

```
sumar(5, 10);          // 15
sumar(1, 2, 3, 4, 5);  // 15
sumar();               // 0
```

2. Crear funciones matemáticas flexibles

```
const maximo = (...nums) => Math.max(...nums);
```

En este caso se usan tanto el spread operator como el rest operator.
Suponga que se ejecuta lo siguiente:

```
maximo(5, 10, 3);
```

El rest operator genera internamente el equivalente a

```
nums = [5, 10, 3];
```

Con este arreglo la función flecha aplica el operador spread operator para expandirlo en valores individuales. Recordemos que `Math.max()` no acepta arreglos, sino valores sueltos. Gracias al operador logramos que esta función reciba todos los elementos del arreglo, pero en forma de valores sueltos.
Esto permite pasarle cualquier cantidad de valores:

```
console.log(maximo(1, 5, 3));      // 5
console.log(maximo(10, 2, 8, 7)); // 10
console.log(maximo(-1, -5, -3));  // -1
```

3. Capturar el resto de los parámetros

```
const mostrar = (primero, ...resto) => {
    console.log(primero);
    console.log(resto);
};

mostrar(10, 20, 30, 40);
// primero = 10
// resto = [20, 30, 40]
```

La diferencia entre spread y rest depende del contexto: Spread expande datos; rest agrupa datos.

Módulos

Los módulos permiten dividir el código en archivos independientes. Esta es una práctica central en aplicaciones modernas. MDN describe los módulos de JavaScript como la sintaxis necesaria para organizar y reutilizar código entre archivos.

Ejemplo de exportación:

```
export const saludar = () => {
    console.log("Hola");
};
```

Ejemplo de importación:

```
import { saludar } from "./utils.js";

saludar();
```

Suponga el siguiente ejemplo:

Tiene el archivo math.js

```
// Exportación individual
export const sumar = (a, b) => a + b;

// Exportación de varias funciones
export const restar = (a, b) => a - b;

// Exportación por defecto (solo puede haber una por archivo)
export default function multiplicar(a, b) {
    return a * b;
}
```

Y usamos este otro archivo (app.js) para invocar las funciones definidas en math.js mediante importación:

```
// Importación de funciones con nombre
import { sumar, restar } from "./math.js";

// Importación de la exportación por defecto
import multiplicar from "./math.js";

console.log(sumar(2, 3));         // 5
console.log(restar(10, 4));       // 6
console.log(multiplicar(3, 5));   // 15
```

En este simpe ejemplo se evidencia la importancia de los módulos porque organizan el código en piezas pequeñas y claras. En vez de tener un archivo gigante, separamos:

- funciones matemáticas
- validaciones
- manejo de datos
- lógica de negocio
- utilidades

Esto significa aplicar principios de buenos diseños respecto de la distribución de la sintaxis, también denominado arquitectura.

Otras ventajas relevantes del uso de módulos son:

- Evitan colisiones de nombres. Sin el uso de módulos esto es imposible
  ```
  function validar() {}
  function validar() {} // colisión de nombres  de funciones
  ```
  En cambio, mediante módulos podrían colocarse en diferentes archivos y exportarlos con diferentes nombres para importarlos desde el archivo donde es necesario usar ambas versiones de la función *validar()*. Esto significa que no es necesario cambiar el nombre de la función original con el objetivo de evitar la colisión
  ```
  import { validarUsuario } from "./usuarios.js";
  import { validarProducto } from "./productos.js";
  ```
- Permiten reutilizar código en distintos proyectos. Un módulo bien hecho puede copiarse o publicarse como paquete.

- Permiten desarrollar software de manera moderna. Webpack, Vite, Node.js, React, Vue, Svelte, entre otros dependen de módulos ES6. Pero los módulos NO son de estas tecnologías, sino que las usan porque JavaScript moderno lo permite.

A continuación, se presenta un ejemplo común de uso de los módulos. Imagine un pequeño sistema de usuarios que consta de los siguientes archivos de Javascript:

- usuarios.js

```
export const usuarios = [
    { id: 1, nombre: "Ana" },
    { id: 2, nombre: "Luis" }
];

export function buscarUsuario(id) {
    return usuarios.find(u => u.id === id);
}
```

- app.js

```
import { usuarios, buscarUsuario } from "./usuarios.js";

console.log("Usuarios:", usuarios);
console.log("Buscando ID 2:", buscarUsuario(2));
```

En este caso, desde `app.js` se importa tanto el arreglo como la función *`buscarUsuario`*`()` ubicados en `usuarios.js`. De esta manera el contenido de `app.js` es más reducido, fácil de seguir y controlado. En el ejemplo se visualizan en consola tanto el contenido del arreglo como el objeto dentro de ese arreglo que cumpla con la condición establecida en la función *`buscarUsuario`*`()`.

En React, los módulos se utilizan todo el tiempo. Cada componente suele vivir en su propio archivo y luego se importa donde se necesita.

Async/Await

JavaScript trabaja frecuentemente con operaciones asincrónicas: pedir datos a una API, esperar una respuesta de red, leer información externa, etc. async/await permite escribir código asincrónico con una estructura más clara, evitando cadenas complejas de promesas (concepto que se abordará más adelante). MDN indica que await permite trabajar con comportamiento basado en promesas dentro de funciones async.

Por ejemplo, un código como este

```
const obtenerUsuarios = async () => {
    const respuesta = await
fetch("https://jsonplaceholder.typicode.com/users");
    const datos = await respuesta.json();

    console.log(datos);
};
```

Debe interpretarse de la siguiente manera:

- `async` indica que la función trabajará con operaciones asincrónicas.
- `await` espera la resolución de una promesa.
- `fetch` realiza una petición HTTP.

- `respuesta.json()` transforma la respuesta en datos utilizables.

¿Qué son las operaciones asincrónicas?

Una operación asincrónica es una tarea que tarda un tiempo (por ejemplo: pedir datos a un servidor, leer un archivo, esperar una respuesta), y JavaScript no se queda bloqueado esperando a que termine de ejecutarse esa tarea. Javascript sigue ejecutando el resto del programa y cuando la tarea termina, vuelve con el resultado.

Una operación asincrónica es algo que no se resuelve al instante. JavaScript la inicia, sigue con otras cosas, y cuando llega la respuesta, la procesa

Esta capacidad de Javascript es importante, principalmente porque es muy probable que la aplicación se comunique con servidores externos solicitando información, realización de cálculos, u otras acciones que pueden demorar en ejecutarse. Sin esta capacidad la página o aplicación se congelaría.

Ejemplos típicos de operaciones asincrónicas que surgen de usar Javascript o React son:

- pedir datos a una API (fetch)
- leer archivos
- esperar un temporizador (setTimeout)
- comunicarse con un servidor

En React, `async/await` se utilizará especialmente al trabajar con APIs dentro de `useEffect` (otro concepto muy importante que se tratará más adelante).

`async/await` permite escribir operaciones asincrónicas de forma más legible y cercana al flujo natural de lectura del código.

FUNDAMENTOS DE REACT

React es una biblioteca de JavaScript para construir interfaces de usuario. Su propuesta central consiste en dividir la interfaz en componentes. Un componente es una parte de la interfaz que puede tener estructura visual, datos, comportamiento y, en algunos casos, estado propio.

Un componente puede representar elementos muy pequeños, como un botón, o estructuras más complejas, como una tarjeta, un formulario, una lista o incluso una pantalla completa. Esta idea permite construir aplicaciones a partir de piezas más pequeñas, comprensibles y reutilizables.

Por ejemplo, un botón puede representarse como un componente:

```
const Boton = () => {
    return <button>Guardar</button>;
};
```

A simple vista parece un ejemplo mínimo, pero encierra una idea fundamental: en React, una parte de la interfaz puede expresarse como una función. Esa función devuelve lo que debe mostrarse en pantalla. Por eso, una aplicación React no se construye como un único bloque grande de HTML y JavaScript, sino como una composición de componentes. Observe en siguiente código:

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

En este ejemplo, `App` es un componente principal que utiliza otro componente llamado `Boton`. Esta composición es uno de los pilares de React: una aplicación se organiza mediante componentes que se combinan entre sí.

La documentación oficial de React presenta esta idea indicando que las aplicaciones React se construyen a partir de piezas aisladas de interfaz llamadas componentes, y que un componente de React es una función de JavaScript que puede contener marcado (es decir escribir etiquetas de html dentro de la función de Javascript).

React como modelo Mental

Para comprender React, no alcanza con aprender una sintaxis nueva. El punto central es adoptar otro modelo mental para construir interfaces.

En JavaScript tradicional, cuando queremos modificar la interfaz, solemos pensar en términos de instrucciones sobre el DOM:

> *"Cuando el usuario haga clic, busco este elemento del DOM y cambio su texto."*

En React se piensa:

> *"Si el estado vale esto, la interfaz debe verse así."*

Por ejemplo, en un contador desarrollado con manipulación directa del DOM, podríamos pensar así:

```
let contador = 0;

const elemento = document.getElementById("contador");
const boton = document.getElementById("btnIncrementar");

boton.addEventListener("click", () => {
    contador++;
    elemento.innerText = contador;
});
```

Este código funciona. Sin embargo, expresa un enfoque **imperativo**. Es decir, le indicamos al programa paso a paso qué debe hacer:

1. crear una variable,
2. buscar un elemento del DOM,
3. escuchar un evento,
4. incrementar el valor,
5. modificar manualmente el contenido del elemento.

Este estilo responde a la pregunta:

¿Cómo modifico la interfaz?

En React, la pregunta cambia. Ya no pensamos primero en buscar elementos y modificarlos manualmente. Pensamos:

Si el estado vale esto, ¿cómo debería verse la interfaz?

Esta diferencia es profunda. React no elimina JavaScript ni HTML. Lo que hace es proponer una forma más ordenada de construir interfaces complejas, especialmente cuando los datos cambian con frecuencia.

La documentación oficial de React describe este enfoque como **declarativo**: el programador le dice a React qué debe renderizar desde la lógica del componente, y React decide cómo mostrarlo al usuario.

Del enfoque imperativo al enfoque declarativo

Para entender el salto conceptual, conviene comparar los dos modelos usando el mismo problema: un contador.

La solución desde el enfoque imperativo con manipulación del DOM

En Javascript tradicional, podríamos volver a escribir el código desarrollado en el apartado anterior:

```
let contador = 0;

const elemento = document.getElementById("contador");
const boton = document.getElementById("btnIncrementar");

boton.addEventListener("click", () => {
    contador++;
    elemento.innerText = contador;
});
```

Este enfoque dice explícitamente **cómo** actualizar la pantalla. El programador se hace responsable de modificar el DOM cada vez que cambia el dato.

La lógica mental sería:

"Cambio el valor y después actualizo manualmente el elemento visual."

El problema es que, cuando la interfaz crece, esta sincronización manual puede volverse difícil. Si muchos elementos dependen del mismo dato, el programador debe recordar actualizar cada parte de la interfaz que corresponda.

Enfoque declarativo con React

En React, se piensa de otra manera:

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

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

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

En este caso, no buscamos manualmente un elemento del DOM para cambiar su texto. La interfaz expresa directamente que debe mostrar el valor actual de `contador`:

```
<p>{contador}</p>
```

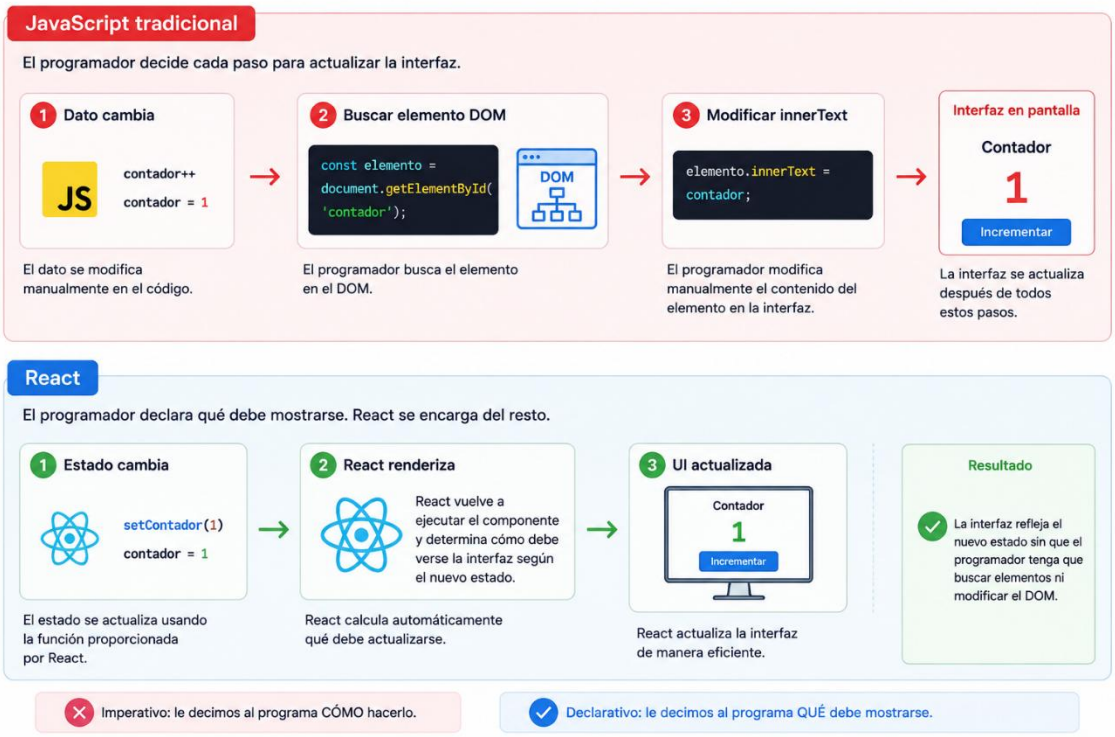
Y cuando queremos cambiar el estado, usamos:

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

A partir de ese cambio, React vuelve a renderizar el componente. Es decir, vuelve a calcular cómo debe verse la interfaz según el nuevo estado.

La documentación oficial explica que, al dispararse un render, React llama a los componentes para averiguar qué debe mostrarse en pantalla. En el render inicial llama al componente raíz y, en renders posteriores, llama al componente cuya actualización de estado disparó el render.

La siguiente imagen expresa la aplicación sobre este ejemplo de ambos enfoques



¿Qué es el estado?

El estado es la información que puede cambiar con el tiempo y que afecta lo que se muestra en pantalla.

En una aplicación, no todo dato es estado. Un dato se considera estado cuando:

- puede cambiar durante la ejecución,
- influye en la interfaz,
- debe ser recordado entre renderizados,
- y su cambio debe provocar una actualización visual.

Ejemplos típicos de estado son:

- el valor de un contador,
- el texto ingresado en un formulario,
- una lista de tareas,
- una lista de usuarios cargada desde una API,
- si un menú está abierto o cerrado,
- si un modal se muestra o no,
- si una operación está cargando,
- si ocurrió un error.

Por ejemplo:

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

Esta línea indica que el componente necesita recordar un valor llamado `contador`, cuyo valor inicial es 0. También indica que la forma correcta de cambiar ese valor es mediante `setContador`.

React describe el estado como “la memoria de un componente”. Además, señala que el estado es local a cada instancia del componente en pantalla. Si se renderiza el mismo componente dos veces, cada copia conserva su propio estado aislado.

El Estado no es una variable común

Este punto es muy importante para evitar errores. En JavaScript tradicional, podríamos escribir:

```
let contador = 0;

contador++;
```
(1)

Pero en React, una variable común no alcanza para provocar un cambio visual en la interfaz. React necesita saber que un dato cambió para volver a renderizar el componente.

Por eso se utiliza *`useState`*:

```
const [contador, setContador] = useState(0);
```
(2)

La diferencia es conceptual, (1) representa una variable común, en cambio (2) representa un dato administrador por React.

React conserva ese dato entre renderizados y vuelve a ejecutar el componente cuando el estado cambia.

React explica que el estado no se comporta como una variable común que desaparece cuando la función termina. El estado "vive" en React, fuera de la función, y cada render recibe una instantánea del estado correspondiente a ese momento.

La relación funcional entre el componente y el estado

Una forma muy adecuada de resumir el enfoque mental de React y la relación entre un componente y su estado es la siguiente:

$$UI = f(estado)$$

Esto significa que la interfaz de usuario es una función del estado. Dicho de otro modo:

- si el estado vale 0, la pantalla muestra 0;
- si el estado vale 1, la pantalla muestra 1;
- si el estado cambia, la interfaz debe cambiar.

En React no se piensa primero en "cómo modifico el DOM", sino en "qué interfaz corresponde a este estado".

Por ejemplo:

```
<p>{contador}</p>
```

Expresa que la interfaz depende del valor de `contador`. Si `contador` cambia, React actualiza la interfaz.

Las tres ideas centrales de React

El estado es la información que puede cambiar con el tiempo y que afecta lo que se muestra en pantalla.

En el marco de esta materia, podemos sintetizar el fundamento de React en tres ideas:

1. **Componentes.** La interfaz se divide en partes pequeñas, reutilizables y comprensibles. En nuestros ejemplos vimos que

   ```
   const Boton = () => {
       return <button>Guardar</button>;
   };
   ```

   Es un componente que puede usarse dentro de otro componente:

   ```
   const App = () => {
       return (
           <main>
               <Boton />
               <Boton />
               <Boton />
           </main>
       );
   };
   ```

   Esto permite construir aplicaciones por composición.

2. **Estado.** El estado representa la información que cambia y afecta la pantalla. En nuestro ejemplo vimos que React plantea una forma de establecer cuando una variable es un estado administrado por React:
   `const [contador, setContador] = useState(0);`
   Cuando cambia el estado, React vuelve a renderizar el componente. En secciones posteriores estudiaremos useState como mecanismo para crear estados y sus particularidades.
3. **Declarativiidad.** En lugar de manipular manualmente el DOM, describimos cómo debe verse la interfaz. React permite sentencias como la siguiente:
   `<p>{contador}</p>`
   Porque responde al esquema mental declarativo. En lugar de "buscar el elemento concreto para cambiar el valor que muestra" (enfoque imperativo) establecemos que "la pantalla debe mostrar el valor actual del contador" (enfoque declarativo).

Ejemplo introductorio integral

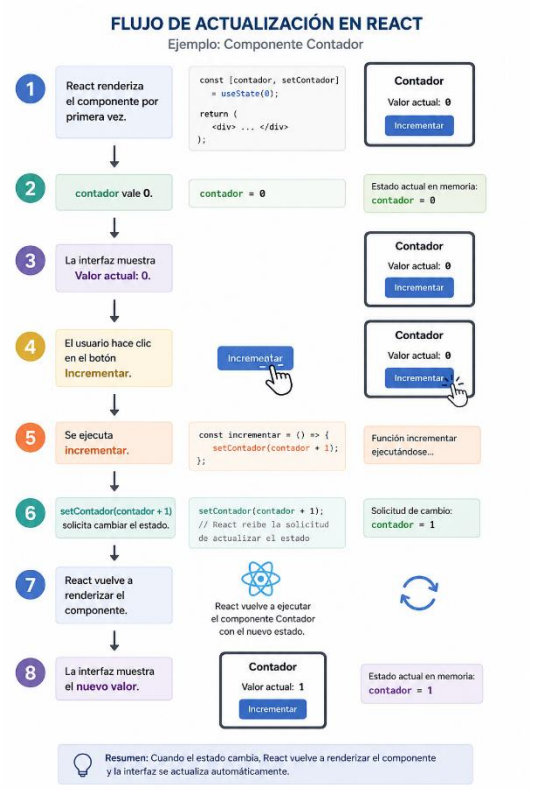
Para que los conceptos no queden aislados, se propone el siguiente ejemplo de una aplicación React que integra todo lo visto hasta ahora. El objetivo es que pueda reflexionar sobre el funcionamiento de la aplicación web, aunque existan detalles técnicos y conceptos que serán abordados con el detalle necesario más adelante (por ejemplo, como se llamaría el archivo que aloja este código, como se crean los proyectos React, cómo funciona la ejecución de eventos en esta tecnología o desde donde se invoca este componente)

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

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

    const incrementar = () => {
        setContador(contador + 1);
    };

    return (
        <section>
            <h2>Contador</h2>

            <p>Valor actual: {contador}</p>

            <button onClick={incrementar}>
                Incrementar
            </button>
        </section>
    );
};

export default Contador;
```

De manera resumida, el flujo completo del funcionamiento de este componente es el siguiente:

1. React renderiza el componente por primera vez.
2. `contador` vale 0.
3. La interfaz muestra Valor actual: 0.
4. El usuario hace clic.
5. Se ejecuta *`incrementar`*.
6. *`setContador`*`(contador + 1)` solicita cambiar el estado.
7. React vuelve a renderizar el componente.
8. La interfaz muestra el nuevo valor.

La siguiente figura expresa este flujo de manera gráfica:



Por otro lado, esta última imagen representa una síntesis de todos los conceptos destacados sobre los fundamentos de React: