UD 3. Interactividad mediante Hooks

1. Comunicación avanzada entre componentes

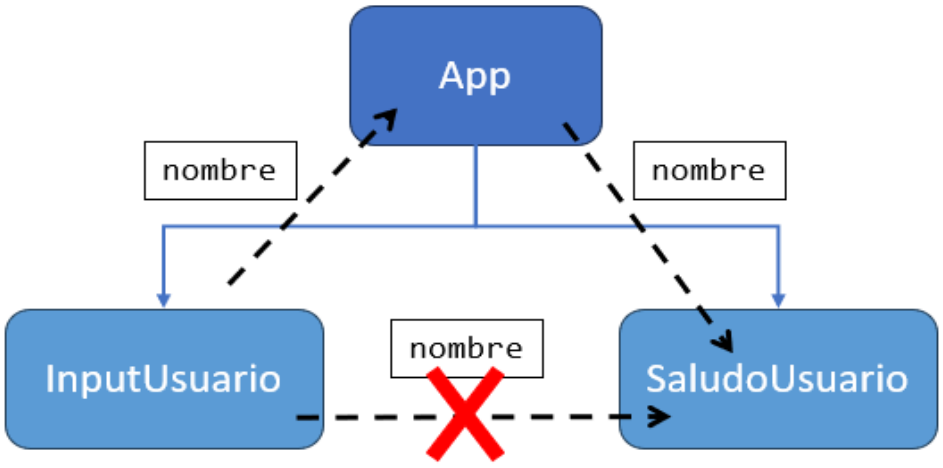
En React, hemos visto que la comunicación básica entre componentes se basa en el **flujo unidireccional de datos**: los datos fluyen del componente padre al hijo a través de props. Sin embargo, a medida que las aplicaciones crecen, surgen necesidades más complejas: componentes que no están directamente relacionados (hermanos o componentes en diferentes partes del árbol) necesitan compartir datos o coordinarse.

1.1 Patrón *Lifting State Up*

Cuando dos o más componentes necesitan compartir el mismo estado (por ejemplo, el valor de un input que afecta a varios componentes), el estado debe "elevarse" al ancestro común más cercano que englobe a todos los componentes que lo necesitan. Este ancestro actúa como "fuente de la verdad", gestionando el estado y pasándolo a los hijos mediante props. A este patrón de desarrollo se le conoce como *lifting state up*.

Ejemplo práctico: Sincronizar dos componentes con un input compartido

Imagina una aplicación donde un componente InputUsuario permite al usuario escribir su nombre, y otro componente SaludoUsuario muestra un saludo basado en ese nombre. Ambos componentes necesitan acceder al mismo valor del nombre, por lo que elevamos el estado al componente padre (App).



/src/App.js

```
// src/App.js

import { useState } from 'react';
import InputUsuario from './InputUsuario.js';
import SaludoUsuario from './SaludoUsuario.js';

function App() {
 const [nombre, setNombre] = useState('');
 return (
  <div>
   <h1>Aplicación para saludar</h1>
   <InputUsuario nombre={nombre}
setNombre={setNombre} />
   <SaludoUsuario nombre={nombre} />
  </div>
 );
}

export default App;
```

/src/InputUsuario.js

```
// src/InputUsuario.js

function InputUsuario({ nombre, setNombre }) {
 return (
  <div>
    <label>
     Nombre:
     <input
       type="text"
       value={nombre}
            onChange={(e) =>
setNombre(e.target.value)}
     />
    </label>
  </div>
 );
}

export default InputUsuario;
```

/src/ SaludoUsuario.js

```
// src/SaludoUsuario.js

function SaludoUsuario({ nombre }) {
  return (
    <p>{nombre ? `¡Hola, ${nombre}!` : 'Por
favor, introduce tu nombre'}</p>
  );
}
export default SaludoUsuario
```

Como vemos, no podemos comunicar directamente 2 componentes "hermanos" sino que han de hacerlo a través de su ancestro común (App en este caso, pero podría ser cualquier otro componente).

Entendamos el ejemplo de código anterior:

- **Estado en el padre**: el estado nombre y la función setNombre se declaran en App usando useState, porque App es el ancestro común más cercano a los otros 2 componentes.
- **Paso de props**: App pasa nombre y setNombre a InputUsuario, también pasa nombre a SaludoUsuario para mostrar el saludo.
- **Sincronización**: Cuando el usuario escribe en el input, setNombre actualiza el estado en App, y React re-renderiza ambos componentes con el nuevo valor de nombre.
- **Ventajas**: Ambos componentes están sincronizados porque comparten el mismo estado, y la lógica está centralizada en App.

1.2. Props y funciones de callback

Los **hijos no pueden cambiar su estado padre directamente**, es decir, no pueden modificar directamente las props que reciben (son de sólo lectura) Sin embargo, un componente hijo puede comunicarse con su padre pasando datos a través de funciones callback. Estas funciones se pasan como props desde el padre, y el hijo las ejecuta para actualizar el estado del padre. Esto permite invertir el flujo de datos de abajo hacia arriba de forma controlada.

Este patrón es clave para manejar formularios, diálogos, componentes modales, etc.

Ejemplo práctico: Botón hijo que actualiza un contador en el padre

Crearemos un componente BotonIncrementar que, al hacer clic, incrementa un contador gestionado por el componente padre App. El padre pasa una función callback para actualizar el estado.

/src/App.js

```
// src/App.js

import { useState } from 'react';
import BotonIncrementar from
'./BotonIncrementar.js';
function App() {
 const [contador, setContador] = useState(0);
 const manejarIncremento = () => {
  setContador(contador + 1);
 };
  return (
  <div>
  <h1>Contador: {contador}</h1>
  <BotonIncrementar
onIncrementar={manejarIncremento} />
  </div>
 );
}
export default App;
```

/src/BotonIncrementar.js

```
// src/BotonIncrementar.js
function BotonIncrementar({ onIncrementar }) {
 return (
  <button onClick={onIncrementar}>
    Incrementar
  </button>
 );
}
export default BotonIncrementar;
```

1.3. Comunicación entre hermanos, prop drilling y límites

Los **componentes hermanos** son aquellos que comparten el mismo componente padre en el árbol de componentes, pero no tienen una relación directa padre-hijo entre sí. Como el flujo de datos en React es unidireccional (de padres a hijos a través de props), los hermanos no pueden comunicarse directamente. Para que un componente hermano afecte o comparta datos con otro, el estado debe gestionarse en el componente padre (usando Lifting State Up), y las acciones se coordinan mediante funciones callback.

Este patrón asegura que ambos hermanos estén sincronizados con el mismo estado y puedan desencadenar cambios que afecten al otro. La forma correcta de hacerlo es hacer *lifting* del estado al componente padre común y propagarlo a ambos hermanos vía props.

Ejemplo: un hermano modifica, otro responde

/src/App.js

```
// src/App.js

import React, { useState } from 'react';

function Panel() {
  const [contador, setContador] = useState(0);

  return (
    <>
      <Boton onClick={() => setContador(contador
+ 1)} />
      <Visor valor={contador} />
    </>
  );
}

function Boton({ onClick }) {
  return <button
onClick={onClick}>Incrementar</button>;
}

function Visor({ valor }) {
  return <p>Contador: {valor}</p>;
}

function App() {
  return (
    <div>
      <h1>Contador</h1>
      <Panel />
    </div>
  );
}

export default App;
```

1.4. Composición frente a herencia

React favorece la **composición de componentes** frente a la herencia. Esto implica **construir interfaces a base de componentes anidados y reusables**, donde los componentes padres deciden qué se renderiza, sin extender clases. La herencia requiere crear jerarquías de clases donde los componentes derivan propiedades y métodos de un componente base, lo que puede llevar a estructuras rígidas y difíciles de mantener. Debido a que ese concepto está ligado a los antiguos componentes de Clase, no lo vamos a utilizar.

Ejemplo: Composición con children y slots personalizados

/src/App.js

```
// src/App.js

import React from 'react';

function Card({ titulo, children, footer }) {
  return (
    <div className="card">
      <h2>{titulo}</h2>
          <div>{children}</div>
      {footer && <div
className="footer">{footer}</div>}
    </div>
  );
}

function App() {
  return (
    <Card titulo="Producto"
footer={<button>Comprar</button>}>
      <p>Cacahuetes azucarados.</p>
    </Card>
  );
}

}

export default App;
```

Este patrón favorece la creación de **UI flexible**, sin necesidad de herencias rígidas como en OOP tradicional.

2. Referencias persistentes en componentes

En React, la mayoría del trabajo se realiza de forma declarativa: definimos **qué** queremos renderizar y React se encarga de actualizar el DOM según los cambios de estado o props. Sin embargo, hay casos donde necesitamos acceder directamente a **elementos del DOM o a valores que deben persistir entre renders** sin provocar un re-render. Para eso existe useRef.

¿Qué es useRef?

useRef es un **hook** que devuelve un objeto mutable cuya propiedad .current se puede usar para guardar cualquier valor persistente entre renders. Su uso más común es **referenciar un nodo del DOM**, aunque también se puede emplear para **guardar valores mutables que no deben provocar un nuevo render**.

¿Por qué no usar useState para todo?

Porque useState actualiza el estado y provoca un re-render. En cambio, useRef permite **almacenar información entre renders sin causar re-renderizados innecesarios**.

Acceder directamente al DOM: cuándo y por qué

Aunque React se basa en un modelo declarativo, hay momentos donde **acceder al DOM real es inevitable**:

- Enfocar un input manualmente.
- Medir dimensiones de un elemento.
- Reproducir un vídeo o audio.
- Gestionar integraciones con librerías externas (por ejemplo, Leaflet o Chart.js).
- Gestionar scroll, animaciones o transiciones personalizadas.

En estos casos, useRef permite acceder directamente al nodo DOM **después del renderizado**, de forma segura y sin romper el paradigma de React.

2.1 El hook useRef: referencias a elementos del DOM

La sintaxis básica es

```
const ref = useRef(null); // Crea la ref

<input ref={ref} /> // Se asigna al elemento

ref.current // Se accede al nodo DOM
```

Ejemplo: enfocar un input al pulsar un botón

/src/App.js

```
// src/App.js

import { useRef } from 'react';

function EnfocarInput() {
   const inputRef = useRef(null);

  const enfocar = () => {
    inputRef.current.focus();
  };

  return (
    <>
      <input ref={inputRef} type="text" />
      <button              onClick={enfocar}>Enfocar
campo</button>
    </>
  );
}

function App() {
  return <EnfocarInput />;
}

export default App;
```

2.2 useRef como almacén persistente de valores

Además de acceder al DOM, useRef es útil para almacenar **cualquier valor que deba sobrevivir a los renders** sin provocar un re-render.

Esto es útil, por ejemplo, para:

- Guardar valores anteriores.
- Contar cuántas veces se ha renderizado el componente.
- Almacenar temporizadores (setInterval, setTimeout).
- Flags de estado (yaSeEnvío, primeraCarga, etc.).

/src/App.js

```
// src/App.js

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

function ContadorDeRenders() {
  const [contador, setContador] = useState(0);
  const renders = useRef(1);

  useEffect(() => {
    renders.current++;
  });

  return (
    <>
      <p>Valor: {contador}</p>
      <p>El componente se ha renderizado
{renders.current} veces.</p>
      <button onClick={() => setContador(contador
+ 1)}>Incrementar</button>
    </>
  );
}

function App() {
  return <ContadorDeRenders />;
}

export default App;
```

Aquí, renders.current se actualiza sin afectar al render, actuando como una variable estática interna al componente.