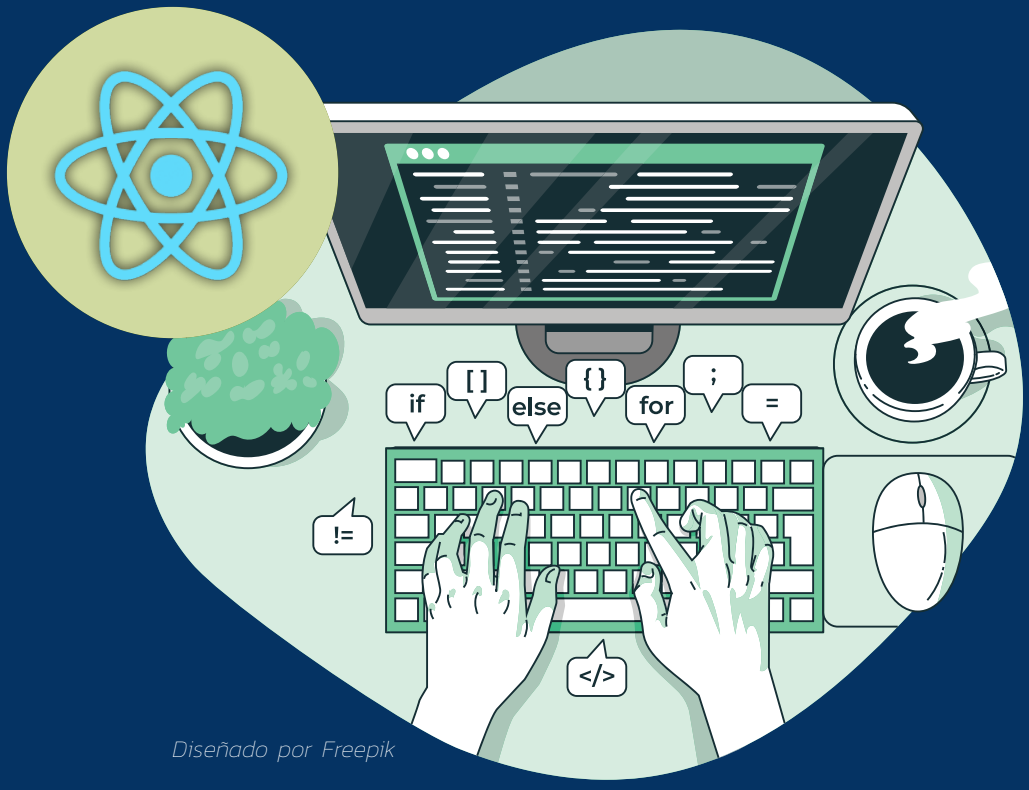
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

ETAPA 3: ESPECIALIZACIONES

REACT



Diseñado por Freepik

Apunte Nº 4

HERRAMIENTAS PARA CASOS COMPLEJOS

4

Año 2025

Reducers

***useReducer* es un Hook de React que proporciona una forma eficiente de manejar lógica compleja de estado en componentes funcionales. Es especialmente útil cuando el estado depende de acciones y reglas claras, y resulta ser una alternativa más escalable a *useState*.**

¿Qué es un Reducer?

Un reducer es una función pura que toma dos argumentos:

1. **El estado actual.**
2. **Una acción.**

Y devuelve un nuevo estado basado en la acción recibida. Este patrón es común en arquitecturas como Redux, pero también se aplica dentro de componentes React mediante el Hook useReducer.

Cuándo Usar useReducer

Considera useReducer cuando:

- Tienes lógica de estado compleja que involucra múltiples sub-valores.
- Las actualizaciones de estado dependen de acciones claras.
- Quieres organizar y consolidar la lógica del estado en un solo lugar.

Sintaxis de useReducer

const [state, dispatch] = useReducer(reducer, initialState);

- reducer: Una función que define cómo se actualiza el estado.
- initialState: El estado inicial del componente.
- state: El estado actual manejado por el reducer.
- dispatch: Una función para enviar acciones al reducer.

```
import { useReducer } from 'react';

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error('Acción desconocida');
  }
}
```

(Continuación de código de página anterior)

```
export default function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Contador: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Incrementar</button
>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrementar</button
>
    </div>
  );
}
```

Diseñando Reducers

Principios Clave:

1. **Pureza:**
   - Un reducer no debe tener efectos secundarios.
   - No debe modificar el estado actual; siempre devuelve un nuevo objeto.
2. **Acciones Claras:**
   - Usa constantes o strings descriptivos como type para las acciones.
   - Proporciona payloads ("datos") adicionales si es necesario.
3. **Escalabilidad:**
   - Divide reducers grandes en reducers más pequeños si es posible.

Manejo de Múltiples Acciones

Un reducer puede manejar cualquier cantidad de acciones.

```
function reducer(state, action) {
  switch (action.type) {
    case 'add':
      return { ...state, items: [...state.items, action.payload] };
    case 'remove':
      return { ...state, items: state.items.filter(item => item.id !== action.pay
load.id) };
    case 'reset':
      return { items: [] };
    default:
      return state;
  }
}
```

Ejemplo: Lista de Tareas

Definición del Reducer

```
function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'add':
      return [...tasks, { id: Date.now(), text: action.payload, completed: false
}];
    case 'toggle':
      return tasks.map(task =>
        task.id === action.payload
          ? { ...task, completed: !task.completed }
          : task
      );
    case 'remove':
      return tasks.filter(task => task.id !== action.payload);
    default:
      throw new Error('Acción desconocida');
  }
}
```

Componente Completo

```
export default function TodoList() {
  const [tasks, dispatch] = useReducer(tasksReducer, []);
  const [taskText, setTaskText] = useState('');

  function handleAdd() {
    if (taskText.trim()) {
      dispatch({ type: 'add', payload: taskText });
      setTaskText('');
    }
  }

  return (
    <div>
      <h1>Lista de Tareas</h1>
      <input
        type="text"
        value={taskText}
        onChange={e => setTaskText(e.target.value)}
      />
      <button onClick={handleAdd}>Agregar</button>
      <ul>
```

(Continuación de código de página anterior)

```
        {tasks.map(task => (
          <li key={task.id}>
            <span
              style={{ textDecoration: task.completed ? 'line-through' : 'none'
}}
              onClick={() => dispatch({ type: 'toggle', payload: task.id })}
            >
              {task.text}
            </span>
            <button onClick={() => dispatch({ type: 'remove', payload: task.id
})}>
              Eliminar
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Comparación con useState

Característica	useState	useReducer
Simplicidad	Ideal para estados simples.	Mejor para lógica de estado compleja.
Estructura	Se maneja con llamadas directas.	Requiere definir acciones y un reducer.
Escalabilidad	Puede complicarse con muchos estados distintos.	Organiza toda la lógica en una única función.

Buenas Prácticas

1. **Mantén los reducers puros:** No interactúes con APIs o realices operaciones de E/S dentro de un reducer.
2. **Agrupa la lógica:** Centraliza la lógica de estado en el reducer para mejorar la legibilidad.
3. **Combina reducers:** Si tu estado es complejo, considera dividir reducers y combinarlos manualmente.

Conclusión

useReducer es una herramienta poderosa para manejar estados complejos en React. Organiza la lógica de actualización, facilita el mantenimiento del código y mejora la claridad en aplicaciones grandes.

Custom hooks

Los Custom Hooks son una poderosa herramienta de React que te permite extraer y reutilizar lógica entre componentes, ayudando a mantener tu código limpio, reutilizable y más fácil de mantener.

¿Qué es un Custom Hook?

Un Custom Hook es simplemente una función que:

1. Comienza con el prefijo **use**.
2. Puede contener uno o más Hooks de React (como **useState**, **useEffect**, etc.).
3. Devuelve valores o funciones que encapsulan una lógica reutilizable.

Por ejemplo, si tienes una aplicación que necesita detectar si el usuario está en línea o no, podrías crear un Custom Hook **useOnlineStatus**:

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

function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(navigator.onLine);

  useEffect(() => {
    function updateStatus() {
      setIsOnline(navigator.onLine);
    }

    window.addEventListener('online', updateStatus);
    window.addEventListener('offline', updateStatus);

    return () => {
      window.removeEventListener('online', updateStatus);
      window.removeEventListener('offline', updateStatus);
    };
  }, []);

  return isOnline;
}

export default useOnlineStatus;
```

¿Por qué usar Custom Hooks?

Un Custom Hook es simplemente una función que:

1. **Evitan la duplicación de código:** Puedes encapsular lógicas comunes para reutilizarlas en varios componentes.
2. **Hacen el código declarativo: Los componentes que usan Custom Hooks** son más legibles porque el código se enfoca en qué hace y no en cómo lo hace.
3. **Son independientes:** Cada vez que llamas a un Custom Hook, crea su propio estado y efectos, lo que permite que varios componentes lo usen simultáneamente sin interferencias.

Creando y utilizando Custom Hooks

Ejemplo: Detectar si un usuario está online

Imagina que quieres mostrar un mensaje diferente dependiendo de si el usuario está conectado. Con **useOnlineStatus**, puedes reutilizar esta lógica:

Custom Hook: useOnlineStatus

```
function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(navigator.onLine);

  useEffect(() => {
    const updateStatus = () => setIsOnline(navigator.onLine);

    window.addEventListener('online', updateStatus);
    window.addEventListener('offline', updateStatus);

    return () => {
      window.removeEventListener('online', updateStatus);
      window.removeEventListener('offline', updateStatus);
    };
  }, []);

  return isOnline;
}
```

Usando el Hook en un componente

```
import useOnlineStatus from './useOnlineStatus';

function StatusBar() {
  const isOnline = useOnlineStatus();
```

(Continuación de código de página anterior)

```
    return (
      <div>
        <p>{isOnline ? 'Estás conectado ✅' : 'Estás desconectado ❌'}</p>
      </div>
    );
  }
```

Extrayendo lógica de formularios

Supongamos que tienes un formulario con varios campos y quieres manejar sus valores y cambios de manera eficiente.

Custom Hook: useForm

```
function useForm(initialValues) {
  const [values, setValues] = useState(initialValues);

  const handleChange = (e) => {
    const { name, value } = e.target;
    setValues({ ...values, [name]: value });
  };

  return [values, handleChange];
}
```

Usando el Hook en un formulario

```
import useForm from './useForm';

function LoginForm() {
  const [values, handleChange] = useForm({ username: '', password: '' });

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Valores enviados:', values);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Usuario:
```

(Continuación de código de página anterior)

```
          Usuario:
          <input
            type="text"
            name="username"
            value={values.username}
            onChange={handleChange}
          />
        </label>
        <label>
          Contraseña:
          <input
            type="password"
            name="password"
            value={values.password}
            onChange={handleChange}
          />
        </label>
        <button type="submit">Iniciar sesión</button>
      </form>
    );
}
```

Buenas prácticas para Custom Hooks

1. **Nombra tus Hooks claramente:** El nombre debe comenzar con use y describir qué hace. Ejemplo: useAuth, useLocalStorage.
2. **Mantén el foco:** Un Custom Hook debe enfocarse en una tarea específica. Si tiene muchas responsabilidades, considera dividirlo.
3. **Evita los efectos secundarios:** Los Hooks deben ser predecibles y solo gestionar lógica relacionada con React.

Conclusión

- Los Custom Hooks permiten encapsular y reutilizar lógica.
- Son fáciles de crear: combina otros Hooks de React para construirlos.
- Usa nombres claros y mantenlos enfocados en una tarea.
- Implementa Hooks para mejorar la modularidad y la legibilidad de tus componentes.

Problemas de referencia de funciones (useCallback)

El Hook useCallback en React no solo es útil para optimizar código, sino también para garantizar que las referencias de las funciones se mantengan estables. Esto ayuda a evitar renderizados innecesarios.

¿Qué es useCallback?

useCallback es un Hook que memoiza una función, devolviendo la misma referencia entre renderizados si sus dependencias no cambian. Esto es particularmente útil para:

1. Evitar la recreación de funciones en cada renderizado.
2. Garantizar que los Custom Hooks que dependen de referencias estables funcionen correctamente.

```
const memoizedCallback = useCallback(() => {
  // Lógica de la función
}, [dependencies]);
```

Caso de Uso: Evitar Re-ejecuciones en useEffect

Problema

Cuando un Custom Hook expone una función que se pasa como dependencia a useEffect, React al re-renderizar el componente crea la función nuevamente, esto sucede porque es la forma en la JavaScript funciona, incluso si su lógica interna no se modifica. Esto puede causar que el efecto se vuelva a ejecutar innecesariamente. Por ejemplo:

```
function useCounter() {
  const [count, setCount] = useState(0);

  function increment() {
    setCount(prev => prev + 1);
  }

  return { count, increment };
}

function Counter() {
  const { count, increment } = useCounter();

  useEffect(() => {
    console.log('Efecto ejecutado');
  }, [increment]); // El efecto se ejecuta en cada renderizado

  return (
    <div>
      <p>Count: {count}</p>
```

(Continuación de código de página anterior)

```
        <button onClick={increment}>Incrementar</button>
      </div>
    );
  }
```

Solución con useCallback

Podemos usar useCallback para garantizar que la referencia de increment se mantenga estable entre renderizados, evitando que el efecto se re-ejecute innecesariamente.

```
function useCounter() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount(prev => prev + 1);
  }, []);

  return { count, increment };
}

function Counter() {
  const { count, increment } = useCounter();

  useEffect(() => {
    console.log('Efecto ejecutado');
  }, [increment]); // Ahora el efecto solo se ejecuta cuando increment cambia

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Incrementar</button>
    </div>
  );
}
```

Buenas prácticas

1. **Usa useCallback solo cuando sea necesario:** Si la función no se pasa como dependencia de un efecto o prop, no es necesario memoizarla.

2. Mantén las dependencias claras: Asegúrate de incluir todas las variables necesarias en el array de dependencias del efecto.
3. Combina con Custom Hooks: Usa useCallback dentro de Custom Hooks para evitar que las dependencias cambien innecesariamente, si es necesario, siempre recordá el punto 1.

Conclusión

useCallback es clave para garantizar referencias estables en situaciones donde los efectos o Custom Hooks dependen de funciones memoizadas. Aplicar este Hook correctamente puede evitar re-ejecuciones innecesarias y mejorar el rendimiento de tus aplicaciones React.

Estados globales (Context)

Context en React te permite compartir información entre componentes sin necesidad de pasar props manualmente a través de cada nivel de la jerarquía. Esto es particularmente útil para evitar el "prop drilling" y manejar datos que necesitan ser accesibles globalmente o por varias partes de tu aplicación.

¿Qué es Context?

El Context es una API de React que permite que un componente padre proporcione datos a cualquier componente dentro de su árbol de hijos, sin importar cuán profundo esté. Esto elimina la necesidad de pasar props a través de todos los niveles intermedios.

Ejemplo de problema:

Cuando necesitas que un dato esté disponible para muchos componentes, podrías terminar pasando props de forma repetitiva (En el caso a continuación, el prop user se pasa manualmente desde App hasta Profile. Esto puede complicarse si hay muchos niveles o si el dato es usado por varios componentes) :
:

```
function App() {
  const user = { name: 'John Doe' };

  return (
    <Header user={user} />
  );
}

function Header({ user }) {
  return (
```

(Continuación de código de página anterior)

```
    <Nav user={user} />
  );
}

function Nav({ user }) {
  return (
    <Profile user={user} />
  );
}

function Profile({ user }) {
  return <p>Hola, {user.name}</p>;
}
```

Solución con Context:

El Context permite "teletransportar" datos directamente al componente que los necesita, evitando props innecesarias:

En el siguiente ejemplo, el componente **Profile** puede acceder al dato **user** directamente desde el contexto, sin que los componentes intermedios (como **Header** o **Nav**) necesiten pasarlo.

```
import React, { createContext, useContext } from 'react';

const UserContext = createContext();

function App() {
  const user = { name: 'John Doe' };

  return (
    <UserContext.Provider value={user}>
      <Header />
    </UserContext.Provider>
  );
}

function Header() {
  return <Nav />;
}

function Nav() {
  return <Profile />;
}
```

(Continuación de código de página anterior)

```
function Profile() {
  const user = useContext(UserContext);
  return <p>Hola, {user.name}</p>;
}
```

Uso de Context paso a paso

1. Crear el Context

Usa **createContext** para crear un nuevo Context. Esto se hace normalmente en un archivo separado *(ver siguiente ejemplo)*:

El valor inicial que pasas a **createContext** es el valor predeterminado, que se usa si ningún proveedor envuelve a los consumidores del Context.

```
import { createContext } from 'react';

export const UserContext = createContext();
```

2. Proveer el Context

Envuelve los componentes que necesitan acceder al Context dentro de un **Provider**. Este **Provider** está disponible en la propiedad **Provider** del objeto creado con **createContext**:

El prop **value** del **Provider** es el dato que estará disponible para todos los consumidores dentro del árbol.

```
function App() {
  const user = { name: 'John Doe' };

  return (
    <UserContext.Provider value={user}>
      <Header />
    </UserContext.Provider>
  );
}
```

3. Consumir el Context

Dentro de los componentes que necesitan el dato, usa el Hook **useContext** para acceder al valor proporcionado:

```jsx
import { useContext } from 'react';
import { UserContext } from './UserContext';

function Profile() {
  const user = useContext(UserContext);
  return <p>Hola, {user.name}</p>;
}
```

Casos comunes de uso

- **Themes:** Pasar la configuración de tema (oscuro/claro) a través de la aplicación.
- **Usuario actual:** Compartir la información del usuario logueado entre componentes.
- **Localización:** Proveer configuraciones de idioma.
- **Estado global:** Manejar estados que afectan múltiples partes de la aplicación, como configuraciones o flags globales.

Conclusión

Context es una herramienta poderosa que simplifica el flujo de datos en React, eliminando la necesidad de prop drilling. Sin embargo, debe usarse con moderación y en los casos apropiados.

Portals

Un portal en React te permite renderizar componentes **fuera del árbol DOM de su componente padre**, manteniendo su comportamiento dentro del árbol de componentes de React (como acceso a props, contexto y eventos). Es útil para casos como modales, tooltips o diálogos que necesitan "escapar" de contenedores con estilos restrictivos (overflow: hidden, z-index, etc.).

Sintaxis Básica

```jsx
import { createPortal } from 'react-dom';

// Ejemplo básico:
function Component() {
  return createPortal(
    <div>Contenido del portal</div>,
    document.getElementById('portal-root') // Nodo DOM destino
  );
}
```

¿Cuándo Usar Portales?

- **Modales o Diálogos:** Para evitar problemas con z-index o estilos heredados.
- **Tooltips/Notificaciones:** Renderizar en la raíz del documento para posicionamiento absoluto.
- **Integrar con Código Legacy:** Insertar componentes React en nodos DOM fuera de la app principal.

Ejemplo Práctico: Modal con Portal

```
import { createPortal } from 'react-dom';
import { useState } from 'react';

function Modal({ children, onClose }) {
  return createPortal(
    <div className="modal-overlay">
      <div className="modal-content">
        <button onClick={onClose}>Cerrar</button>
        {children}
      </div>
    </div>,
    document.getElementById('modal-root') // <-- Nodo destino en el HTML
  );
}

function App() {
  const [showModal, setShowModal] = useState(false);

  return (
    <div>
      <button onClick={() => setShowModal(true)}>Abrir Modal</button>
      {showModal && (
        <Modal onClose={() => setShowModal(false)}>
          <h2>¡Hola desde el portal!</h2>
        </Modal>)}
    </div>);
}
```

HTML Requerido (para el portal):

```
<!-- Fuera del contenedor raíz de React -->
<div id="modal-root"></div>
```

Comportamiento de Eventos

Los eventos dentro de un portal **se propagan según el árbol de componentes de React**, no según el DOM físico. Por ejemplo, un clic en el botón Cerrar del modal anterior activará onClose aunque el modal esté en otro nodo DOM. *(Ver el siguiente ejemplo en código debajo).*

Consideraciones Clave

1. Accesibilidad: Los modales deben manejarse con prácticas ARIA (ej: role="dialog", aria-modal).
2. Nodo Destino: Asegúrate de que el nodo DOM destino (ej: modal-root) exista antes de usar el portal.
3. Ciclo de Vida: Si el nodo destino se crea dinámicamente, usa useEffect para limpiarlo al desmontar.

```
// Ejemplo: Event Bubbling en React
function ParentComponent() {
  const handleClick = () => {
    console.log('Evento llegó al componente padre de React');
  };

  return (
    <div onClick={handleClick}>
      <Modal> {/* Portal renderizado fuera de este div */}
        <button>Al hacer clic aquí, se activará handleClick</button>
      </Modal>
    </div>);
}
```

Errores Comunes

- **Olvidar el Nodo DOM Destino:** Si el nodo no existe, el portal fallará silenciosamente.
- **No Limpiar Portales:** Si creas nodos dinámicos, elimínalos para evitar fugas de memoria.

```
// Ejemplo: Crear nodo dinámico con useEffect
function Modal() {
  const portalRoot = document.createElement('div');

  useEffect(() => {
    document.body.appendChild(portalRoot);
    return () => document.body.removeChild(portalRoot); // Limpieza
  }, []);

  return createPortal(<div>...</div>, portalRoot);
}
```

Layout computation

Es el proceso de calcular la geometría de elementos en el DOM (posición, tamaño, etc.).

En React, cuando necesitas **medir o modificar el DOM antes de que el navegador pinte la pantalla**, useLayoutEffect es la herramienta clave.

useLayoutEffect vs useEffect

Característica	useEffect	useLayoutEffect
Momento de ejecución	Asíncrono, después de pintar la pantalla	Síncrono, antes de pintar la pantalla
Uso ideal	Fetch de datos, subscriptions	Medición / Ajuste del DOM
Impacto visual	Puede causar "flicker"	Evita flicker

Sintaxis Básica

```
import { useLayoutEffect } from 'react';

function Component() {
  useLayoutEffect(() => {
    // Lógica de medición o modificación del DOM aquí
    return () => { /* Limpieza (opcional) */ };
  }, [dependencies]);
}
```

Casos de Uso Comunes

1. Medir el Tamaño de un Elemento

```
function ResponsiveBox() {
  const [width, setWidth] = useState(0);
  const boxRef = useRef(null);
```

(Continúa en la siguiente página)

(Continuación de código de página anterior)

```
  useLayoutEffect(() => {
    const measureWidth = () => {
      if (boxRef.current) {
        setWidth(boxRef.current.offsetWidth);
      }
    };

    measureWidth();
    window.addEventListener('resize', measureWidth);

    return () => window.removeEventListener('resize', measureWidth);
  }, []);

  return <div ref={boxRef}>Ancho: {width}px</div>;
}
```

2. Ajustar el Scroll Automáticamente

```
function ChatList({ messages }) {
  const listRef = useRef(null);

  useLayoutEffect(() => {
    // Mantener el scroll al final de la lista
    listRef.current.scrollTop = listRef.current.scrollHeight;
  }, [messages]);

  return <div ref={listRef}>{messages.map(/* ... */)}</div>;
}
```

3. Animaciones sin "Flicker"

```
function FadeInBox() {
  const boxRef = useRef(null);

  useLayoutEffect(() => {
    const box = boxRef.current;
    // Configurar opacidad inicial ANTES de que el navegador pinte
    box.style.opacity = 0;
    // Animación suave
    requestAnimationFrame(() => {
      box.style.transition = 'opacity 0.5s';
      box.style.opacity = 1;
```

(Continuación de código de página anterior)

```
    });
  }, []);

  return <div ref={boxRef}>¡Aparece suavemente!</div>;
}
```

Comportamiento Detallado

1. Orden de Ejecución:
Los useLayoutEffect se ejecutan en el mismo orden que useEffect, pero **bloquean la pintura del navegador** hasta que terminan.

2. Server-Side Rendering (SSR):
useLayoutEffect **no funciona en SSR** (genera advertencias). Usa useEffect en esos casos.

Errores Comunes

X | Modificar el DOM sin useLayoutEffect (causa flicker) |:

```
function BadExample() {
  const ref = useRef(null);

  useEffect(() => { // ← Usa useEffect por error
    ref.current.style.padding = '20px'; // El usuario verá un cambio repentino
  }, []);

  return <div ref={ref}>Contenido</div>;
}
```

✔ | Solución con useLayoutEffect: |:

```
function GoodExample() {
  const ref = useRef(null);

  useLayoutEffect(() => {
    ref.current.style.padding = '20px'; // Ajuste antes de la pintura
  }, []);

  return <div ref={ref}>Contenido</div>;
}
```

Manejo imperativo

¿Qué es useImperativeHandle?

Este hook permite **personalizar la instancia que se expone cuando un componente padre usa ref** en un componente hijo. Es útil para controlar funcionalidades específicas del hijo (como enfocar un input, iniciar animaciones, etc.) de forma imperativa, sin pasar props.

Sintaxis Básica

```
import { useImperativeHandle, forwardRef } from 'react';

function Hijo({ ref }) {
  useImperativeHandle, () => ({
    // Métodos/propiedades expuestos al padre
    focus() { /* ... */ },
    scrollToTop() { /* ... */ },
  }));

  return <div>...</div>;
};

// Uso en el padre:
function Padre() {
  const hijoRef = useRef(null);
  return <Hijo ref={hijoRef} />;
}
```

Casos de Uso Comunes

1. Exponer Métodos de un Input Personalizado

```
function InputPersonalizado({ ref }) {
  const inputRef = useRef(null);

  useImperativeHandle(ref, () => ({
    focus: () => inputRef.current.focus(),
    clear: () => (inputRef.current.value = ''),
  }));

  return <input ref={inputRef} />;
};
```

(Continuación de código de página anterior)

```
// Padre:
function Formulario() {
  const inputRef = useRef(null);

  return (
    <div>
      <InputPersonalizado ref={inputRef} />
      <button onClick={() => inputRef.current.clear()}>Limpiar</button>
    </div>);
}
```

2. Controlar un Video o Animación

```
function Reproductor({ ref }) {
  const videoRef = useRef(null);

  useImperativeHandle(ref, () => ({
    play: () => videoRef.current.play(),
    pause: () => videoRef.current.pause(),
  }));

  return <video ref={videoRef} src="video.mp4" />;
});

// Padre:
function Controles() {
  const reproductorRef = useRef(null);
  return (
    <div>
      <Reproductor ref={reproductorRef} />
      <button onClick={() => reproductorRef.current.play()}>Reproducir</button>
    </div>);
}
```

Gestión del foco

Controlar el foco de manera accesible y predecible es crítico para la experiencia de usuario, especialmente en formularios, modales o componentes dinámicos. flushSync ayuda a forzar actualizaciones síncronas del DOM cuando es necesario.

¿Qué es flushSync?

Es una función de React DOM que **fuerza la aplicación inmediata de actualizaciones de estado y del DOM de forma síncrona**, evitando el comportamiento asíncrono por defecto de React. Se usa en casos excepcionales donde necesitas que el DOM esté actualizado antes de realizar una acción (como enfocar un elemento).

¿Por qué usar flushSync para el Foco?

1. **Problema común:** Si intentas enfocar un elemento después de un cambio de estado (ej: renderizar un input), React aún no ha actualizado el DOM, por lo que el elemento no existe.
2. **Solución:** flushSync asegura que el DOM se actualice antes de ejecutar la acción (como element.focus()).

Sintaxis Básica

```
import { flushSync } from 'react-dom';

// Ejemplo:
function handleAction() {
  flushSync(() => {
    setState(newValue); // Actualización de estado síncrona
  });
  // El DOM ya está actualizado aquí
}
```

Casos de Uso con Gestión de Foco

1. Enfocar un Input al Renderizar un Componente

```
function DynamicInput() {
  const [showInput, setShowInput] = useState(false);
  const inputRef = useRef(null);

  const handleClick = () => {
    flushSync(() => {
      setShowInput(true); // Actualización síncrona
    });
    // Ahora el input existe en el DOM
    inputRef.current.focus();
  };
  return (
    <div>
      <button onClick={handleClick}>Mostrar Input</button>
      {showInput && <input ref={inputRef} />}
    </div>);
}
```

2. Enfocar un Modal al Abrir

```
function Modal({ isOpen, onClose }) {
  const modalRef = useRef(null);

  useEffect(() => {
    if (isOpen) {
      flushSync(() => {}); // Fuerza actualización pendiente (si existe)
      modalRef.current.focus();
    }
  }, [isOpen]);

  return (
    isOpen && (
      <div ref={modalRef} tabIndex={-1} role="dialog">
        <button onClick={onClose}>Cerrar</button>
      </div>)
  );
}
```

3. Enfocar el Siguiente Campo en un Formulario

```
function Form() {
  const inputs = [useRef(null), useRef(null), useRef(null)];

  const handleKeyPress = (index, e) => {
    if (e.key === 'Enter') {
      flushSync(() => {});
      const nextIndex = index + 1;
      if (nextIndex < inputs.length) {
        inputs[nextIndex].current.focus();
      }
    }
  };

  return (
    <form>
      {inputs.map((ref, index) => (
        <input
          key={index}ref={ref}onKeyDown={(e) => handleKeyPress(index, e)}/>))}
    </form>);
}
```

Advertencias Importantes

1. Impacto en el Rendimiento:
flushSync puede degradar el rendimiento al evitar la agrupación de actualizaciones. Úsalo solo cuando sea estrictamente necesario.

2. Accesibilidad:
El foco debe moverse de manera lógica y predecible. Usa aria-live o role="alert" para notificar cambios a lectores de pantalla.

Alternativas a flushSync

autoFocus: Úsalo para elementos que deben enfocarse al montarse:

```
<input autoFocus /> // Enfoca automáticamente al renderizar
```

Efectos con Dependencias:

```
useEffect(() => {
  if (condition) elementRef.current.focus();
}, [condition]); // Se ejecuta después de que el DOM se actualiza
```

Errores Comunes

x | Enfocar sin Esperar la Actualización |:

```
function Error() {
  const [show, setShow] = useState(false);
  const ref = useRef(null);

  const handleClick = () => {
    setShow(true);
    ref.current.focus(); // ⚠ El input aún no existe en el DOM
  };

  return <>{show && <input ref={ref} />}</>;
}
```

✔ | Solución con flushSync |:

```
const handleClick = () => {
  flushSync(() => setShow(true)); // DOM actualizado aquí
  ref.current.focus();
};
```

Sincronización de datos externos

¿Qué es useSyncExternalStore?

Es un hook de React que permite **suscribirse a un almacenamiento externo** (como una API del navegador, una biblioteca de estado externa o un WebSocket) y sincronizar su estado con un componente React. Es ideal para integrar datos que React no controla directamente.

¿Por qué usarlo?

1. **Evitar "Tearing"**: Garantiza que todos los componentes muestren el mismo estado externo, incluso durante actualizaciones concurrentes.
2. **Reemplaza useEffect + useState**: Simplifica la suscripción a fuentes externas y evita fugas de memoria.

Sintaxis Básica

```
import { useSyncExternalStore } from 'react';

const state = useSyncExternalStore(
  subscribe, // Función para suscribirse al almacenamiento externo
  getSnapshot, // Función que devuelve el valor actual del estado
  getServerSnapshot? // (Opcional) Snapshot para Server-Side Rendering (SSR)
);
```

Casos de Uso Comunes

1. Suscribirse a una API del Navegador (Ej: Online Status)

```
function OnlineStatus() {
  const isOnline = useSyncExternalStore(
    (callback) => {
      window.addEventListener('online', callback);
      window.addEventListener('offline', callback);
      return () => {
        window.removeEventListener('online', callback);
        window.removeEventListener('offline', callback);
      };
    },
    () => navigator.onLine // Snapshot actual
  );

  return <div>Status: {isOnline ? 'Online' : 'Offline'}</div>;
}
```

2. Integrar con una Biblioteca de Estado Externa (Ej: Zustand, Redux)

```
// Store externo (simulado)
let externalStore = { count: 0 };
const listeners = new Set();

function subscribe(callback) {
  listeners.add(callback);
  return () => listeners.delete(callback);
}

function increment() {
  externalStore.count++;
  listeners.forEach((listener) => listener());
}

// Componente React
function Counter() {
  const count = useSyncExternalStore(
    subscribe,
    () => externalStore.count // Snapshot
  );

  return <button onClick={increment}>Clicks: {count}</button>;
}
```

3. Suscribirse a un WebSocket

```
function WebSocketData() {
  const message = useSyncExternalStore(
    (callback) => {
      const ws = new WebSocket('wss://api.example.com');
      ws.onmessage = (event) => callback(event.data);
      return () => ws.close();
    },
    () => 'Waiting for data...' // Valor inicial
  );

  return <div>Último mensaje: {message}</div>;
}
```

Comportamiento Clave

1. Suscripción Automática:
React gestiona la suscripción y desuscripción cuando el componente se monta/desmonta.

2. SSR Compatible:
Usa **getServerSnapshot** para definir un valor inicial durante el renderizado en el servidor:

```
const state = useSyncExternalStore(
  subscribe,
  getSnapshot,
  () => 'Valor inicial de servidor' // getServerSnapshot
);
```

Comparación: useSyncExternalStore vs useEffect

Aspecto	useSyncExternalStore	useEffect + useState
Gestión de Suscripción	Automática	Manual (addEventListener, cleanup)
Consistencia de Estado	Evita tearing (estado incoherente)	Puede mostrar estados intermedios
Complejidad	Menos código	Más propenso a errores y fugas

Errores Comunes

X | Olvidar el Snapshot Inicial en SSR |:

```
// ❌ Error: Sin snapshot para SSR
const data = useSyncExternalStore(subscribe, getSnapshot);

// ✅ Solución: Proporciona un valor inicial seguro
const data = useSyncExternalStore(subscribe, getSnapshot, () => initialValue);
```

X | Modificar el Store Externo Directamente |:

```
function Component() {
  const store = useSyncExternalStore(/* ... */);
  // ❌ Error: Mutar el store externo sin notificar a los listeners
  store.value = 10;
}
```