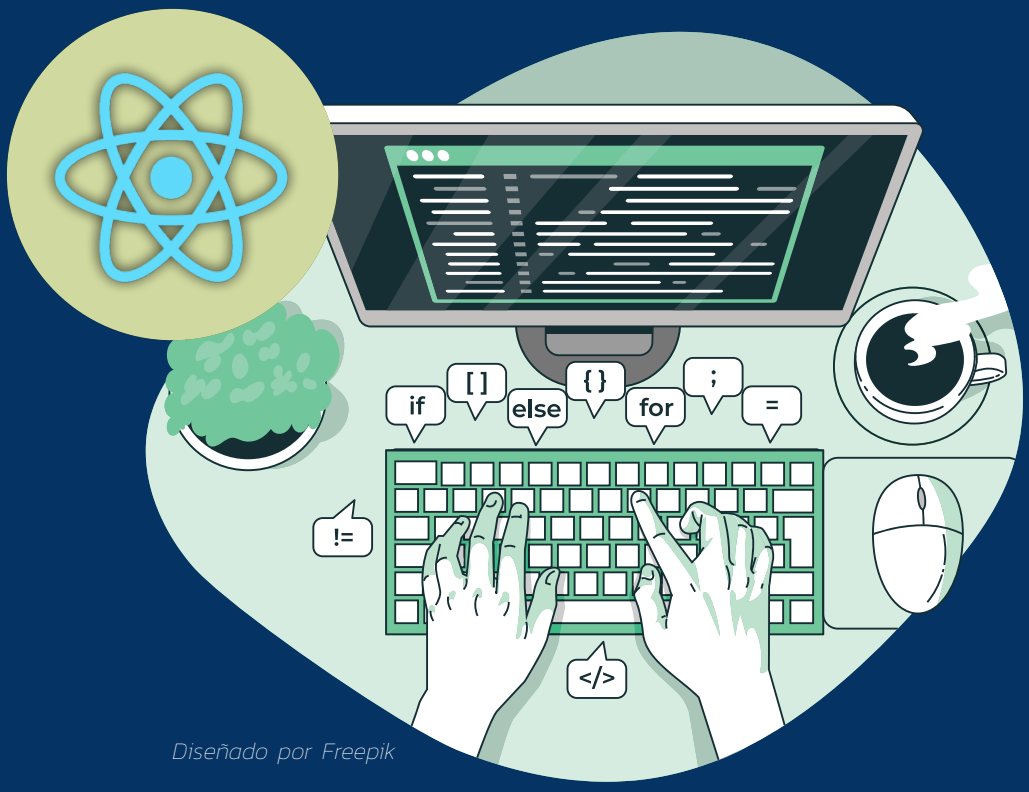
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

ETAPA 3: ESPECIALIZACIONES

REACT



Diseñado por Freepik

Apunte Nº 3

REACT HOOKS

3

Año 2025

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

INFORMATORIO

Eventos: interactuando con nuestros componentes

Los eventos en React son funciones que se ejecutan en respuesta a interacciones del usuario o del sistema.

React los maneja de manera similar a los eventos en HTML, pero con una sintaxis y comportamiento adaptados al entorno de componentes.

- **onClick:** Nombre del evento que responde al clic del usuario.
- **handleClick:** La función que se ejecuta cuando ocurre el evento.

La sintáxis básica es:

```
function MyComponent() {
  function handleClick() {
    alert('¡Botón presionado!');
  }

  return <button onClick={handleClick}>Presióname</button>;
}
```

Uso

Pasar argumentos a los manejadores

Cuando necesitas pasar argumentos a una función manejadora, puedes usar una arrow function

```
function List() {
  function handleItemClick(id) {
    console.log(`Item ${id} clickeado`);
  }

  return (
    <ul>
      {[1, 2, 3].map(id => (
        <li key={id} onClick={() => handleItemClick(id)}>
          Item {id}
        </li>
      ))}
    </ul>
  );
}
```

Eventos sintéticos

React utiliza un sistema de eventos sintéticos que encapsula eventos nativos del navegador, garantizando compatibilidad entre navegadores y optimización de rendimiento.

```
function Input() {
  function handleChange(event) {
    console.log(event.target.value);
  }

  return <input type="text" onChange={handleChange} />;
}
```

Prevenir el comportamiento por defecto

Para evitar que un evento realice su acción predeterminada, utiliza **event.preventDefault()**.

```
function Link() {
  function handleClick(event) {
    event.preventDefault();
    console.log('Enlace bloqueado');
  }

  return <a href="https://example.com" onClick={handleClick}>Haz clic aquí</a>;
}
```

Detener la propagación

Puedes detener la propagación de un evento utilizando **event.stopPropagation()**.

```
function Container() {
  function handleContainerClick() {
    console.log('Contenedor clickeado');
  }

  function handleButtonClick(event) {
    event.stopPropagation();
    console.log('Botón clickeado');
  }
```

(Continuación de código de página anterior)

```
  return (
    <div onClick={handleContainerClick}>
      <button onClick={handleButtonClick}>Botón</button>
    </div>
  );
}
```

Resolución de problemas comunes

El manejador de eventos no funciona:

1. ¿Asignaste el evento correctamente? Asegúrate de usar **camelCase**: **onClick**, no **onclick**.

2. ¿La función está definida? Verifica que el manejador está correctamente definido.

```
// Correcto
function MyComponent() {
  function handleClick() {
    console.log('Clickeado');
  }

  return <button onClick={handleClick}>Click</button>;
}

// Incorrecto
function MyComponent() {
  return <button onClick={handleClick}>Click</button>; // handleClick no está def
inido
}
```

Referencias incorrectas:

No llames directamente al manejador en la definición del evento.

```
function MyComponent() {
  function handleClick() {
    console.log('Clickeado');
  }

  return <button onClick={handleClick}>Click</button>; // Correcto

  // return <button onClick={handleClick()}>Click</button>; // Incorrecto
}
```

Conclusión

El manejo de eventos en React es poderoso y flexible. Entender cómo funcionan los eventos sintéticos, prevenir comportamientos predeterminados y pasar argumentos a manejadores te permitirá crear interfaces más interactivas y controladas.

Estados: la memoria de nuestros componentes

Los componentes a menudo necesitan cambiar lo que se muestra en pantalla como resultado de una interacción.

Escribir dentro de un formulario debería actualizar el campo de texto, hacer clic en «siguiente» en un carrusel de imágenes debería cambiar la imagen que es mostrada; hacer clic en un botón para comprar un producto debería actualizar el carrito de compras. En los ejemplos anteriores los componentes deben «recordar» cosas: el campo de texto, la imagen actual, el carrito de compras. En React a este tipo de memoria de los componentes se le conoce como estado.

useState es un Hook en React que permite agregar una variable de estado a un componente funcional. Es una de las herramientas más fundamentales y esenciales para manejar el estado en React.

Sintaxis básica:

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

function MiComponente() {
  const [estado, setEstado] = useState(valorInicial);
}
```

- **estado**: la variable que contiene el estado actual.
- **setEstado**: la función que actualiza el estado.
- **valorInicial**: el valor inicial del estado.

Uso

Agregar estado a un componente

El estado se declara en el nivel superior del componente. *(ver el ejemplo a continuación).*

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

  return (
    <button onClick={() => setContador(contador + 1)}>
      Has presionado el botón {contador} veces
    </button>
  );
}
```

Actualizar el estado basado en el estado previo

Cuando necesitas usar el valor actual del estado para calcular el siguiente, pasa una función al **setter**.

```
function Incrementar() {
  const [edad, setEdad] = useState(42);

  function manejarClick() {
    setEdad(prevEdad => prevEdad + 1);
  }

  return (
    <button onClick={manejarClick}>Incrementar Edad</button>
  );
}
```

Actualizar objetos y arreglos en el estado

En React, el estado es inmutable, por lo que no debes modificar objetos o arreglos directamente.

```
function Formulario() {
  const [form, setForm] = useState({
    nombre: '',
    apellido: ''
  });
```

(Este código continúa en la página siguiente)

```
  function manejarCambio(e) {
    setForm({
      ...form,
      [e.target.name]: e.target.value
    });
  }

  return (
    <>
      <input name="nombre" value={form.nombre} onChange={manejarCambio} />
      <input name="apellido" value={form.apellido} onChange={manejarCambio} />
    </>
  );
}
```

Evitar recrear el estado inicial

Si el estado inicial requiere cálculos costosos, pasa una función como argumento a **useState**.

```
function ListaDeTareas() {
  function crearTareasIniciales() {
    return Array.from({ length: 50 }, (_, i) => ({ id: i, texto: `Tarea ${i + 1}`
}));
  }

  const [tareas, setTareas] = useState(crearTareasIniciales);
}
```

Resolución de problemas comunes

He actualizado el estado, pero el valor sigue siendo el anterior

React no actualiza el estado inmediatamente. Las lecturas del estado dentro del mismo ren-der mostrarán el valor previo.

```
function Ejemplo() {
  const [nombre, setNombre] = useState('Taylor');
```

(Este código continúa en la página siguiente)

```
  function manejarClick() {
    setNombre('Robin');
    console.log(nombre); // Aún muestra "Taylor"
  }

  return <button onClick={manejarClick}>Cambiar Nombre</button>;
}
```

La pantalla no se actualiza

Esto ocurre si el nuevo estado es igual al anterior (según la comparación **Object.is**).

"Too many re-renders"

Este error sucede si llamas a la función de actualización dentro del render.

Para evitarlo, usa efectos o manejadores de eventos.

```
function ComponenteInvalido() {
  const [contador, setContador] = useState(0);
  setContador(contador + 1); // ⚠️ Esto causa un loop infinito
}
```

Mi función de inicialización se ejecuta dos veces

En modo estricto, React ejecuta funciones de inicialización dos veces en desarrollo para detectar impurezas. Esto no ocurre en producción.

Conclusión

useState es una herramienta poderosa para manejar el estado local en React. Comprender su funcionamiento y las mejores prácticas para usarlo te ayudará a crear componentes más eficientes y mantenibles. Recuerda siempre trabajar con estados inmutables y evitar modificaciones directas para garantizar el comportamiento esperado.

> ***useState es una herramienta poderosa para manejar el estado local en React. (...) te ayudará a crear componentes más eficientes y mantenibles.***

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

INFORMATORIO

Controlando inputs

Los formularios son elementos fundamentales en las aplicaciones web para capturar datos del usuario.

React proporciona herramientas eficaces para controlar y manejar inputs, permitiendo crear formularios flexibles y robustos.

Inputs Controlados

Un input controlado es aquel cuyo valor es gestionado por el estado de React. Esto garantiza que el valor del input esté siempre sincronizado con el estado de tu componente.

Conceptos Clave:

- **value**: Propiedad que define el valor actual del input.
- **onChange**: Evento que se dispara cada vez que el usuario cambia el valor del input.

Ejemplo:

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

function SimpleForm() {
  const [inputValue, setInputValue] = useState('');

  function handleChange(event) {
    setInputValue(event.target.value);
  }

  return (
    <form>
      <label>
        Nombre:
        <input type="text" value={inputValue} onChange={handleChange} />
      </label>
      <p>El valor actual es: {inputValue}</p>
    </form>
  );
}
```

Múltiples Inputs

Cuando necesitas manejar varios inputs, puedes usar un solo manejador de eventos y actualizar el estado dinámicamente.

Claves:

- Usa el atributo **name** en los inputs para identificar el campo correspondiente.
- Actualiza el estado utilizando una copia del estado previo y sobrescribiendo

Ejemplo:

```
function MultiInputForm() {
  const [formData, setFormData] = useState({
    firstName: '',
    lastName: ''
  });

  function handleChange(event) {
    const { name, value } = event.target;
    setFormData({
      ...formData,
      [name]: value
    });
  }

  return (
    <form>
      <label>
        Nombre:
        <input
          type="text"
          name="firstName"
          value={formData.firstName}
          onChange={handleChange}
        />
      </label>
      <br />
      <label>
        Apellido:
        <input
          type="text"
          name="lastName"
          value={formData.lastName}
          onChange={handleChange}
        />
      </label>
      <p>Nombre completo: {formData.firstName} {formData.lastName}</p>
    </form>
  );
}
```

Manejar Formularios con Envío

El evento **onSubmit** se utiliza para capturar el envío del formulario. Puedes prevenir el comportamiento predeterminado utilizando **event.preventDefault()**.

Claves:

- Usa **onSubmit** en lugar de manejar el evento **onClick** directamente en el botón de envío.
- Verifica que todos los datos requeridos estén presentes antes de enviar.

Ejemplo:

```
function SubmitForm() {
  const [formData, setFormData] = useState({
    email: '',
    password: ''
  });

  function handleChange(event) {
    const { name, value } = event.target;
    setFormData({ ...formData, [name]: value });
  }

  function handleSubmit(event) {
    event.preventDefault();
    console.log('Datos enviados:', formData);
  }
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
      </label>
      <br />
      <label>
```

(Este código continúa en página siguiente)

```
        Contraseña:
        <input
          type="password"
          name="password"
          value={formData.password}
          onChange={handleChange}
        />
      </label>
      <br />
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Validación de Formularios

Para validar datos puedes utilizar reglas personalizadas o librerías como **Formik** o **React Hook Form**.

Ejemplo de validación simple:

```
function ValidationForm() {
  const [email, setEmail] = useState('');
  const [error, setError] = useState('');

  function handleChange(event) {
    const value = event.target.value;
    setEmail(value);
    if (!value.includes('@')) {
      setError('El email debe incluir un @.');
    } else {
      setError('');
    }
  }

  function handleSubmit(event) {
    event.preventDefault();
    if (!error) {
      console.log('Email enviado:', email);
    } else {
      console.log('Corrige los errores antes de enviar.');
    }
  }
```

(Continuación de código de página anterior)

```
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Email:
        <input type="email" value={email} onChange={handleChange} />
      </label>
      {error && <p style={{ color: 'red' }}>{error}</p>}
      <button type="submit" disabled={!!error}>Enviar</button>
    </form>
  );
}
```

Conclusión

El manejo de inputs y formularios en React es flexible y potente. Usar el estado para controlar los valores garantiza consistencia y facilita la validación. Una buena estructura de formulario debe incluir:

1. Inputs controlados.
2. Validación de datos.
3. Manejo correcto de envíos.

Side Effects

El hook **useEffect** de React permite a los componentes sincronizarse con sistemas externos, gestionar efectos secundarios y realizar tareas que van más allá del renderizado. A continuación, exploramos los conceptos clave, mejores prácticas y ejemplos para dominar su uso.

Introducción a los Efectos

Definición:

- Un efecto es código que se ejecuta después de cada renderizado para sincronizar el componente con sistemas externos como APIs o eventos del navegador.

Diferencia con los eventos:

- Los efectos no son causados por una interacción específica (como un clic). En su lugar, ocurren debido al proceso de renderizado.

Por ejemplo, considera un componente que debe conectarse a un servidor cuando aparece en pantalla. Esto no puede hacerse durante el renderizado porque no es una operación pura. Los efectos gestionan este tipo de necesidades.

Ejemplo:

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

function MyComponent() {
  useEffect(() => {
    console.log('Efecto ejecutado');
  });

  return <div>Componente listo</div>;
}
```

Conceptos Clave

1. Declarar un Efecto

Para usar un efecto, importa **useEffect** y declara una función en su interior. Este código se ejecutará después de cada render.

Por defecto, este efecto se ejecutará después de cada render, pero puedes controlar su comportamiento utilizando dependencias.

```
useEffect(() => {
  console.log('Efecto ejecutado');
});
```

2. Dependencias

Las dependencias determinan cuándo un efecto necesita volver a ejecutarse. Si no cambian las dependencias, el efecto no se re-ejecuta.

Advertencia: React verifica automáticamente las dependencias necesarias. No puedes ignorarlas si el código del efecto depende de ellas.

- **Sin dependencias:**

```
useEffect(() => {
  console.log('Solo al montar');
}, []); // Solo se ejecuta una vez
```

- **Con dependencias:**

```
useEffect(() => {
  console.log('Se ejecuta si `count` cambia');
}, [count]);
```

3. Limpieza

Algunos efectos necesitan deshacer cambios cuando se desmonta el componente o se actualiza el efecto. Esto se logra devolviendo una función de limpieza.

```
useEffect(() => {
  const interval = setInterval(() => {
    console.log('Intervalo activo');
  }, 1000);

  return () => clearInterval(interval); // Limpieza
}, []);
```

Casos de Uso Comunes

Sincronización con APIs externas

```
useEffect(() => {
  fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log(data));
}, []); // Solo al montar
```

Subscribirse a eventos

```
useEffect(() => {
  function handleScroll() {
    console.log(window.scrollY);
  }
  window.addEventListener('scroll', handleScroll);

  return () => window.removeEventListener('scroll', handleScroll);
}, []);
```

Controlar componentes que no son de React

```
useEffect(() => {
  const map = new SomeMapLibrary(mapRef.current);
  map.initialize();

  return () => map.destroy();
}, [mapRef]);
```

Buenas Prácticas

1. **Evita efectos innecesarios:** Si puedes calcular algo durante el renderizado, no uses useEffect.
2. **Gestiona las dependencias cuidadosamente:** Asegúrate de incluir todas las variables que tu efecto utiliza.
3. **Implementa la limpieza adecuadamente:** Siempre deshace conexiones, listeners u operaciones pendientes.
4. **Evita condiciones de carrera:** Maneja peticiones de red con abort controllers o banderas.

Modo Estricto

En modo desarrollo, React monta los componentes dos veces para probar la limpieza de los efectos. Implementa siempre una función de limpieza para evitar errores en producción.

Conclusión

- useEffect es un hook poderoso para sincronizar componentes con sistemas externos.
- **Define dependencias adecuadas** para evitar ejecuciones innecesarias.
- **Usa funciones de limpieza** para manejar desmontajes correctamente.
- **Asegúrate de que los efectos sean necesarios**; no los uses para lógica que puede resolverse durante el render. Probablemente, no necesites usar effects. Siempre que lo necesitas pensalo 2 o 3 veces si realmente es la solución más adecuada.

Con estas prácticas, puedes escribir componentes React robustos y eficientes.

Manejo de estado básico

Cuando necesitas que el estado de dos o más componentes cambie de forma coordinada, una solución común es elevar el estado al componente padre más cercano. Esta técnica permite compartir el estado y mantener una fuente única de verdad.

Introducción

1. ¿Qué es elevar el estado?

Consiste en mover el estado desde componentes individuales hacia su componente padre más cercano, para que este controle y coordine el estado compartido.

2. ¿Cuándo usarlo?

- Cuando necesitas que varios componentes compartan el mismo estado.
- Para coordinar cambios en componentes hijos desde un padre común.

3. Ventajas:

- Una fuente única de verdad.
- Simplifica la coordinación de componentes.

Ejemplo: Acordeón

- **Escenario inicial:**

Un componente padre **Accordion** renderiza dos componentes **Panel**, cada uno con su propio estado **isActive**.

- **Problema:** Actualmente, cada **Panel** maneja su estado de forma independiente. Si deseas que solo un Panel se expanda a la vez, necesitarás coordinar su estado desde el componente Accordion.

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

function Panel({ title, children }) {
  const [isActive, setIsActive] = useState(false);

  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={() => setIsActive(true)}>Show</button>
      )}
    </section>
  );
}

export default function Accordion() {
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel title="About">
        With a population of about 2 million, Almaty is Kazakhstan's largest cit
y.
      </Panel>
      <Panel title="Etymology">
        The name comes from the Kazakh word for "apple".
      </Panel>
    </>
  );
}
```

Proceso para Elevar el Estado

1. Remover el estado de los componentes hijos

Elimina el estado **isActive** de los componentes **Panel** y agrégalo como una prop.

```
function Panel({ title, children, isActive, onShow }) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={onShow}>Show</button>
      )}
    </section>
  );
}
```

2. Pasar datos desde el componente padre

Controla el estado desde el componente **Accordion** usando una variable de estado para rastrear qué panel está activo.

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

export default function Accordion() {
  const [activeIndex, setActiveIndex] = useState(0);

  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel
        title="About"
        isActive={activeIndex === 0}
        onShow={() => setActiveIndex(0)}
      >
        With a population of about 2 million, Almaty is Kazakhstan's largest cit
y.
      </Panel>
```

(Continuación de código de página anterior)

```
        <Panel
          title="Etymology"
          isActive={activeIndex === 1}
          onShow={() => setActiveIndex(1)}
        >
          The name comes from the Kazakh word for "apple".
        </Panel>
      </>
    );
}
```

3. Coordinar eventos en los hijos

Pasa manejadores de eventos desde el componente padre para que los hijos puedan cambiar el estado.

```
<Panel
  title="About"
  isActive={activeIndex === 0}
  onShow={() => setActiveIndex(0)}
/>
```

Componentes Controlados y No Controlados

- **Componentes no controlados:** Manejan su propio estado local (ejemplo inicial de Panel).
- **Componentes controlados:** Reciben su estado y manejadores desde props (ejemplo final de Panel).

Los componentes controlados son más flexibles cuando necesitas coordinarlos entre sí.

Buenas Prácticas

- **Evita duplicar estado:** Eleva el estado al padre más cercano en lugar de tener estados separados en cada hijo.
- **Una fuente única de verdad:** Mantén cada pieza de estado en un único lugar.
- **Refactoriza según sea necesario:** Es común mover el estado hacia arriba o hacia abajo mientras diseñas la arquitectura de tu aplicación.

Resumen

- Elevar el estado es una técnica fundamental para compartir estado entre componentes.
- Al mover el estado al padre, puedes coordinar cambios y mantener una fuente única de verdad.

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

INFORMATORIO

Refs

En React, las referencias (refs) permiten a los componentes "recordar" información que no desencadena renderizados. Son útiles para manejar interacciones directas con elementos del DOM o almacenar valores mutables sin afectar el flujo de datos de React.

Definición: Una referencia es un objeto mutable con una propiedad current que puedes leer y modificar sin que React vuelva a renderizar el componente.

Casos de uso:

- Almacenar identificadores de temporizadores (setTimeout, setInterval).
- Acceder a elementos del DOM para manipularlos directamente.
- Retener valores entre renderizados sin afectar la interfaz de usuario.

Diferencias con el estado

Aspecto	Refs	Estado
Renderizado	No desencadenan re-renderizados	Desencadenan re-renderizados al cambiar
Mutabilidad	Son mutables	Inmutables, solo cambiables con `setState`
Uso Ideal	Manejar el DOM o almacenar datos externos	Datos que afectan la interfaz de usuario (UI)

Crear una Ref

Puedes crear una ref usando el Hook useRef.

- **Propiedad current:** Contiene el valor almacenado.

- **Mutable:** Puedes actualizar myRef.current en cualquier momento sin afectar el renderizado.

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

function MyComponent() {
  const myRef = useRef(initialValue); // initialValue es opcional.

  console.log(myRef.current); // Accede al valor actual.

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

Ejemplo: Contador con Ref

- **Nota:** Aunque la ref se actualiza, el componente no se re-renderiza.

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

export default function Counter() {
  const counterRef = useRef(0);

  function handleClick() {
    counterRef.current += 1;
    alert(`Has hecho clic ${counterRef.current} veces.`);
  }

  return (
    <button onClick={handleClick}>
      Hacer clic
    </button>
  );
}
```

Combinar Refs y Estado

Refs y estado pueden trabajar juntos para manejar valores que afectan el renderizado y otros que no.

Detalles clave:

- **Estado:** Maneja valores que afectan la UI (como startTime y now).
- **Refs:** Almacena el ID del intervalo para evitar condiciones de carrera.

Ejemplo: Cronómetro

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

export default function Stopwatch() {
  const [startTime, setStartTime] = useState(null);
  const [now, setNow] = useState(null);
  const intervalRef = useRef(null);

  function handleStart() {
    setStartTime(Date.now());
    setNow(Date.now());
```

(Continuación de código de página anterior)

```
    clearInterval(intervalRef.current);
    intervalRef.current = setInterval(() => {
      setNow(Date.now());
    }, 10);
  }

  function handleStop() {
    clearInterval(intervalRef.current);
  }

  let secondsPassed = 0;
  if (startTime != null && now != null) {
    secondsPassed = (now - startTime) / 1000;
  }

  return (
    <>
      <h1>Tiempo transcurrido: {secondsPassed.toFixed(3)}s</h1>
      <button onClick={handleStart}>Iniciar</button>
      <button onClick={handleStop}>Detener</button>
    </>
  );
}
```

Buenas Prácticas

1. **No leas ni escribas en ref.current durante el renderizado**: Esto puede hacer que el componente sea impredecible.
2. **Usa refs como una "vía de escape"**: Solo para casos donde el estado o props no sean suficientes (e.g., manejar APIs externas o el DOM).
3. **Evita usar refs para datos que impactan la interfaz de usuario**: Usa el estado en su lugar.

Uso Común: Manejar el DOM

Las refs son frecuentemente usadas para interactuar con elementos del DOM.

Ejemplo: Enfocar un Input
(Ver código de página siguiente)

```jsx
import { useRef } from 'react';

export default function InputFocus() {
  const inputRef = useRef(null);

  function handleFocus() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} type="text" placeholder="Escribe algo" />
      <button onClick={handleFocus}>Enfocar</button>
    </>
  );
}
```

Resumen

- Las refs son útiles para almacenar valores mutables sin desencadenar renderizados.
- Ideales para manejar temporizadores, interactuar con el DOM o almacenar valores que no afectan la UI.
- Trabaja con estado para manejar lógica que afecta el renderizado.

Ids Únicos

useId es un Hook de React para generar identificadores únicos que pueden usarse en atributos relacionados con la accesibilidad y otros casos donde se requieran IDs exclusivos.

Introducción

¿Qué es useId?

- **Función principal:** Generar un ID único asociado a la llamada del Hook en un componente.
- **Uso principal:** Atributos de accesibilidad como aria-describedby o id.
- **Beneficio clave:** GGarantiza unicidad incluso si el componente es renderizado múltiples veces.

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

function PasswordField() {
  const passwordHintId = useId();
  // ...
}
```

Usos Comunes

Generar IDs para atributos de accesibilidad

El siguiente ejemplo muestra cómo usar **useId** para enlazar un **input** con un **p** mediante **aria-describedby**:

Este enfoque asegura que los IDs sean únicos incluso si el componente es renderizado varias veces.

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

function PasswordField() {
  const passwordHintId = useId();

  return (
    <>
      <label>
        Password:
        <input type="password" aria-describedby={passwordHintId} />
      </label>
      <p id={passwordHintId}>
        The password should contain at least 18 characters.
      </p>
    </>
  );
}
```

Generar IDs relacionados

Puedes usar **useId** para generar un prefijo compartido y asignarlo a varios elementos:

```jsx
import { useId } from 'react';

export default function Form() {
  const id = useId();

  return (
    <form>
      <label htmlFor={`${id}-firstName`}>First Name:</label>
      <input id={`${id}-firstName`} type="text" />
      <hr />
      <label htmlFor={`${id}-lastName`}>Last Name:</label>
      <input id={`${id}-lastName`} type="text" />
    </form>
  );
}
```

Prefijos Compartidos para IDs

Si tienes múltiples aplicaciones React en la misma página, puedes especificar un prefijo compartido para los IDs generados. Esto evita conflictos entre aplicaciones.
Ejemplo:

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

const root1 = createRoot(document.getElementById('root1'), {
  identifierPrefix: 'app-1-',
});
root1.render(<App />);

const root2 = createRoot(document.getElementById('root2'), {
  identifierPrefix: 'app-2-',
});
root2.render(<App />);
```

Buenas Prácticas

1. Evita hardcodear IDs: Usa useId para evitar colisiones entre componentes.
2. Evita usar useId para claves: Genera claves de lista a partir de los datos.
3. Uso en renderizado del servidor: Asegúrate de que el árbol de componentes sea idéntico entre el cliente y el servidor para evitar inconsistencias.

Comparación con Contadores Incrementales

Aunque podrías usar un contador global para generar IDs (“nextId++”), useId garantiza consistencia en escenarios como:

1. Renderizado del servidor: Los IDs generados coinciden entre el cliente y el servidor.

2. Hidratación: React asegura que los IDs sean consistentes durante la transición del servidor al cliente.