

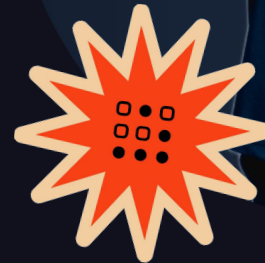
Ciclo de vida (Componentes)

Módulo 6



Elaborado por: Jesua Luján

Jesua Hadai Luján



DEV.F.:



● Módulo 3: Ciclo de Vida de los Componentes en React



🎯 **Objetivo:** Comprender y aplicar el ciclo de vida de los componentes en React



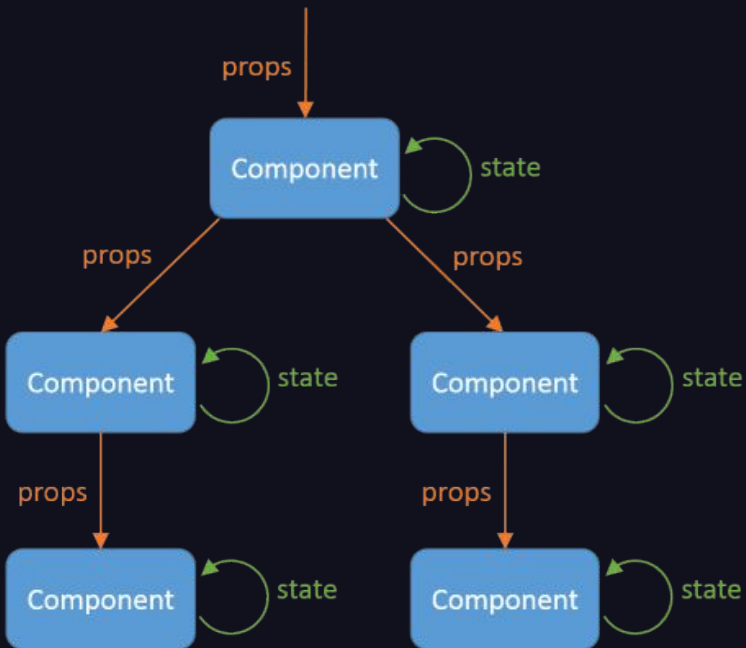
Contenido:

✓ Comprender **las fases del ciclo de vida** de un componente en React. 🤔

✓ Utilizar **useEffect()** para gestionar efectos en **montaje**, actualización y desmontaje. 🧑

✓ Identificar cuándo y cómo limpiar efectos secundarios. 🧑

✓ Aplicar conocimientos del ciclo de vida para mejorar el **performance y organización del código**. 😈



¿Qué es el ciclo de vida de un componente?

Es el conjunto de fases por las que pasa un componente de React **desde que se crea hasta que desaparece del DOM**. Cada una de estas etapas permite que ejecutes código en momentos clave: como **al montar el componente, al actualizarlo o al destruirlo**.





```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      data: null  
    };  
  }  
  
  componentDidMount() {  
    fetch('https://api.example.com/data')  
      .then(response => response.json())  
      .then(data => this.setState({ data }));  
  }  
  
  render() {  
    // ...  
  }  
}
```



¿Qué es el ciclo de vida de un componente?

Antes en las clases se usaban métodos como **componentDidMount()**, pero ahora con los **componentes funcionales y React Hooks**, el protagonista principal es el **hook useEffect()**. 🔥

```
class MiComponente extends React.Component
{
  componentDidMount() {
    console.log("Componente montado");
    // Aquí puedes hacer fetch de datos o
    manipular el DOM
  }

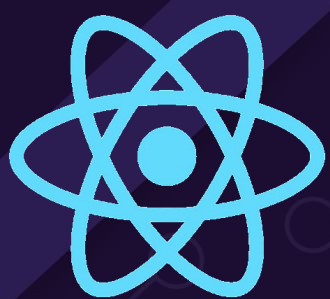
  render() {
    return (
      <div>
        <h1>Hola, soy un componente!</h1>
      </div>
    );
  }
}
```

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

const MiComponente = () => {
  useEffect(() => {
    console.log("Componente montado");
    // Aquí puedes hacer fetch de datos o
    manipular el DOM
  }, []); // El arreglo vacío asegura que
solo se ejecute al montar

  return (
    <div>
      <h1>Hola, soy un componente!</h1>
    </div>
  );
};
```





Las 3 fases del ciclo de vida (versión moderna con hooks)

DEV.F.
DESARROLLAMOS(PERSONAS);





I ● 1. Mounting (Montaje)

El componente se crea e inserta en el DOM.

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

El array vacío [] indica que este efecto solo se ejecuta una vez, al montarse.

🧠 Útil para:

- Fetching de datos iniciales.
- Suscripciones.
- Animaciones de entrada.





I 2. Updating (Actualización)

El componente se vuelve a renderizar por un **cambio de props o estado**.

Hook clave:

```
useEffect(() => {  
  console.log("El componente se actualizó 🔄");  
}, [estado, props]);
```

Se ejecuta cada vez que cambie alguna dependencia listada.

🧠 Útil para:

- Reaccionar a cambios.
- Validaciones.
- Actualizaciones condicionales.





I ● 3. Unmounting (Desmontaje)

El componente se elimina del DOM. Es el adiós definitivo 🙋

Hook clave (cleanup):

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

Ese return es la función de limpieza, como cuando sales de una casa y apagas las luces.

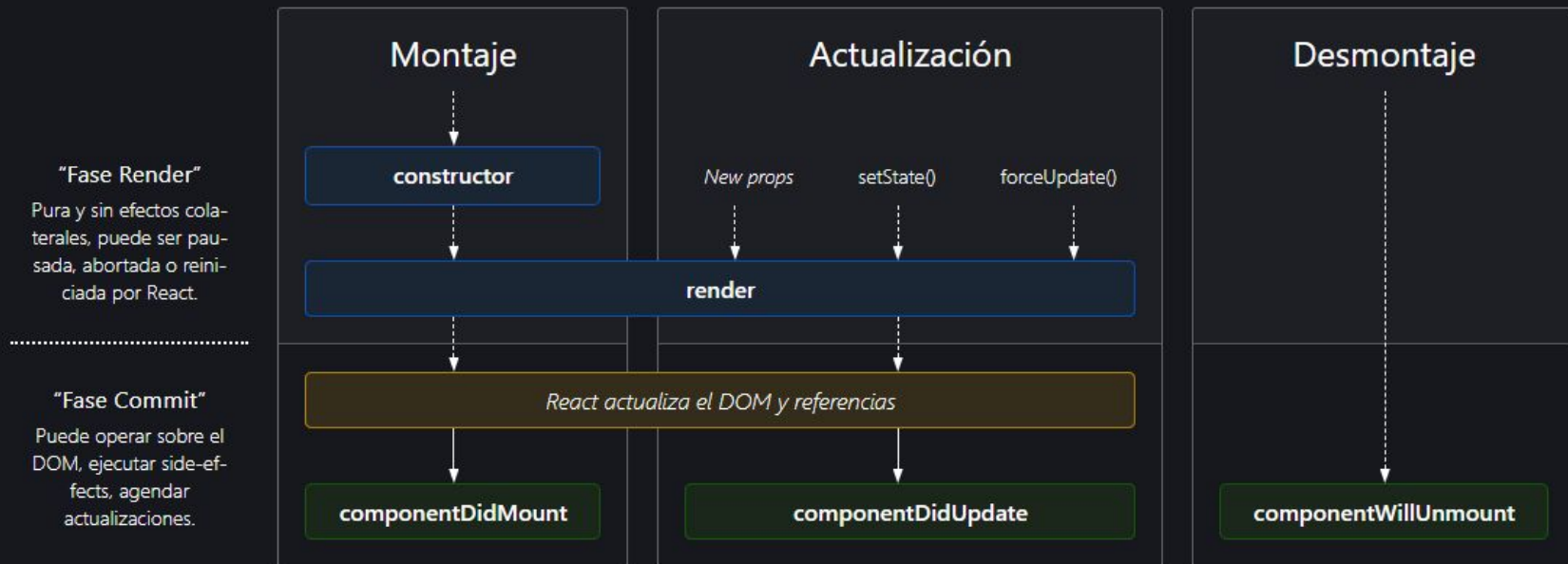
🧠 Útil para:

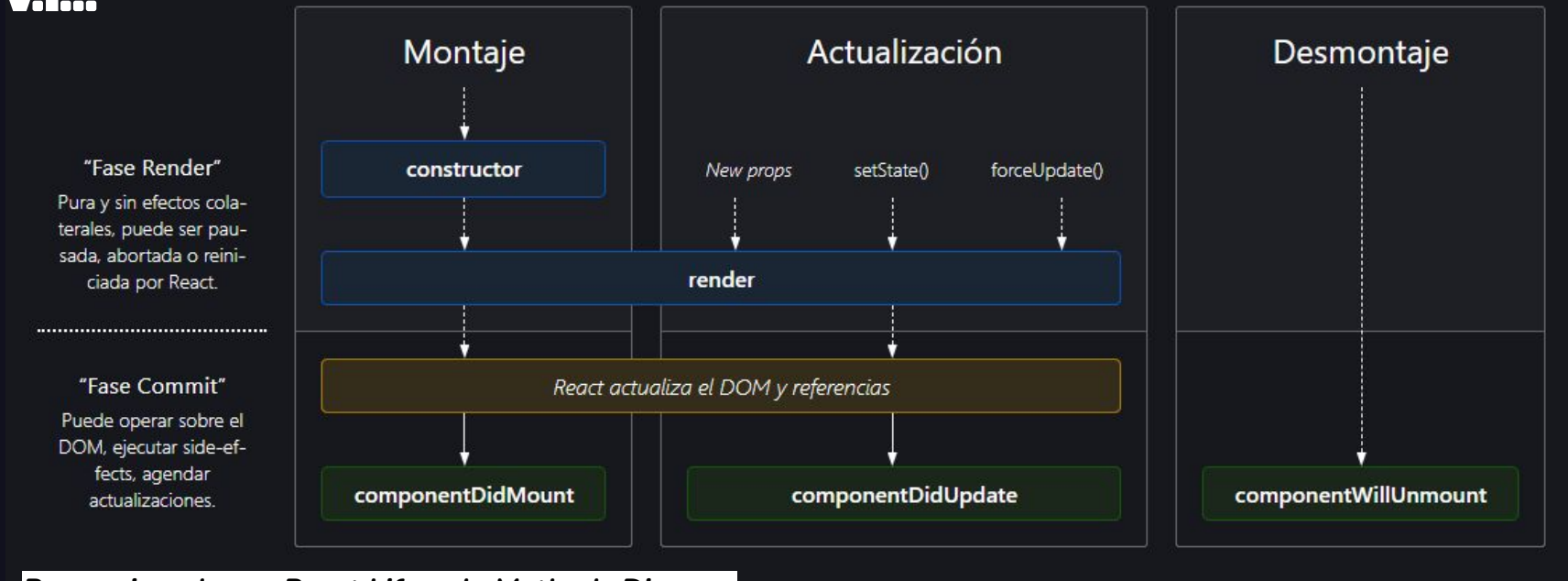
- Cancelar timers o listeners
- Cerrar conexiones
- Limpiar efectos secundarios





Antes de la introducción de **React Hooks (versión 16.8)**, el ciclo de vida era muy distinto a como lo conocemos ahora y se contaba con más métodos en el ciclo de vida, a pesar de ya estar más reducido que en sus primeras versiones, **con la introducción de los hooks esto supuso traer nuevas mejoras.**



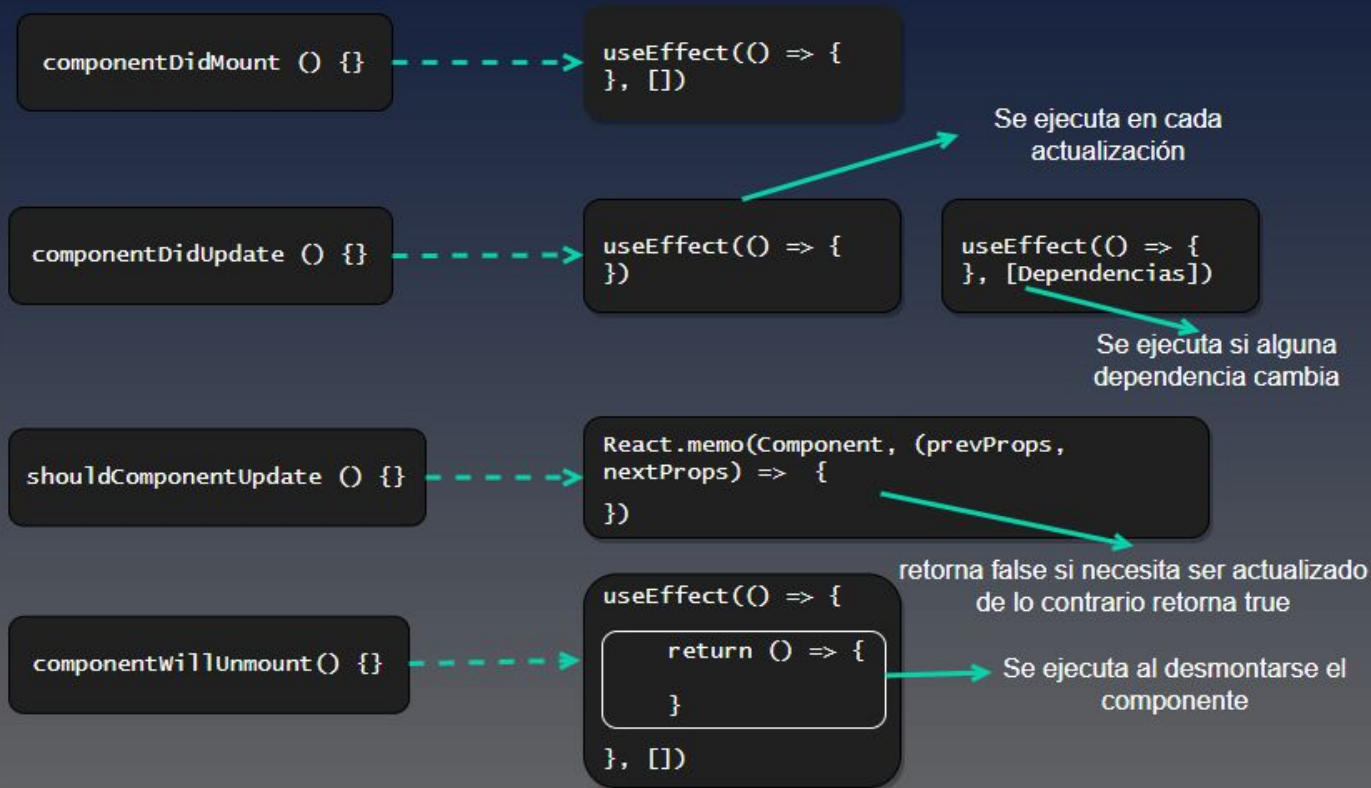


Luego de la introducción de los Hooks esto cambió, para hacer uso de este nuevo ciclo de vida **se tiene que hacer uso de los function components y no de class components**. Además esto se realiza principalmente **con el hook de useEffect**.

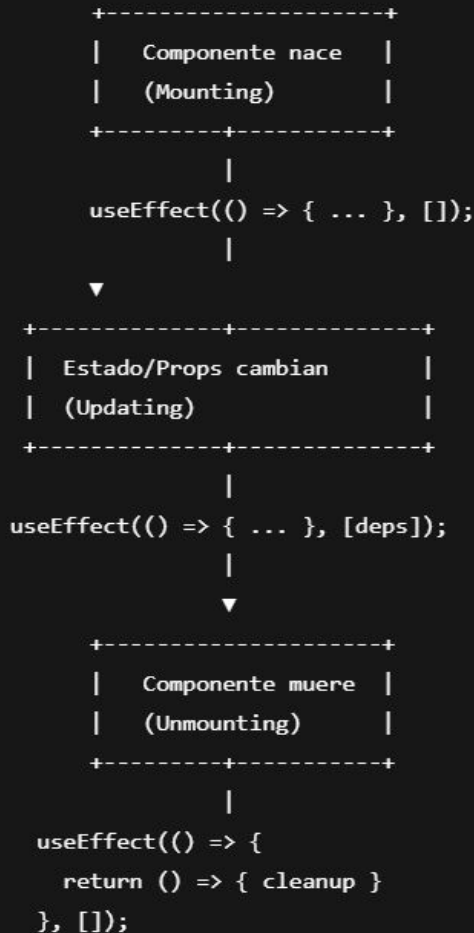




Métodos del ciclo de vida de React en Hooks






Mapa Visual del Ciclo de Vida (con Hooks)



¿Y si quiero una analogía geek

Imagina que tu componente es un Jedi:

-  **Mounting:** El joven padawan nace, recibe su primer sable láser.
-  **Updating:** El Jedi entrena, mejora, reacciona a la Fuerza (props/estado).
-  **Unmounting:** El Jedi se une a la Fuerza, pero deja su legado (cleanup).



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

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

  // Mounting
  useEffect(() => {
    console.log("🟢 Jedi nacido (mount)");

    return () => {
      console.log("🔴 Jedi desaparece en la Fuerza (unmount)");
    };
  }, []);

  // Updating
  useEffect(() => {
    console.log(`📧 La Fuerza se siente diferente: ${count}`);
  }, [count]);

  return (
    <div>
      <h1>Midi-chlorians: {count}</h1>
      <button onClick={() => setCount(count + 1)}>
        ¡Más poder!
      </button>
    </div>
  );
}
```



🧪 Ejemplo Real en Código

✅ ¿Qué significa esto?

🎬 `useEffect(() => { ... }, [])`

Este **useEffect** se ejecuta una sola vez, justo después de que el componente se monta por primera vez en el DOM, es decir, cuando aparece en pantalla.



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

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

  // Mounting
  useEffect(() => {
    console.log("🟢 Jedi nacido (mount)");

    return () => {
      console.log("🔴 Jedi desaparece en la Fuerza (unmount)");
    };
  }, []);

  // Updating
  useEffect(() => {
    console.log(`📧 La Fuerza se siente diferente: ${count}`);
  }, [count]);

  return (
    <div>
      <h1>Midi-chlorians: {count}</h1>
      <button onClick={() => setCount(count + 1)}>
        ¡Más poder!
      </button>
    </div>
  );
}
```



🔍 Explicación paso a paso

1. 🟢 Montaje del componente

Cuando React renderiza **JediCounter** por primera vez:

- Ejecuta todo el contenido dentro del **useEffect**.
- En este caso, muestra por consola:
🟢 Jedi nacido (mount)



👉 Es como si Yoda dijera: “¡Un nuevo aprendiz, tú eres!”


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

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

  // Mounting
  useEffect(() => {
    console.log("🟢 Jedi nacido (mount)");

    return () => {
      console.log("🔴 Jedi desaparece en la Fuerza (unmount)");
    };
  }, []);

  // Updating
  useEffect(() => {
    console.log(`📧 La Fuerza se siente diferente: ${count}`);
  }, [count]);

  return (
    <div>
      <h1>Midi-chlorians: {count}</h1>
      <button onClick={() => setCount(count + 1)}>
        ¡Más poder!
      </button>
    </div>
  );
}
```



🔍 Explicación paso a paso

2. 🔴 Retorno de **useEffect**: limpieza al desmontar

La función que **retorna** desde el **useEffect** (el return **() => { ... }**) no se ejecuta durante el montaje, sino cuando:

- El componente se va del DOM (se desmonta).
- Es decir, cuando deja de existir visualmente en la aplicación.
- En ese momento, aparece en consola:
🔴 Jedi desaparece en la Fuerza (unmount)
- 👉 Como si el componente “muriera” y tuviera que dejar todo limpio antes de fundirse con la Fuerza.

⚠️ ¿Qué tipo de cosas deberías hacer en esta fase de montaje?

```
// ✅ Llamar a una API externa
fetch("https://swapi.dev/api/people/1")
  .then(res => res.json())
  .then(data => setJedi(data));

// ✅ Suscribirse a un evento global
window.addEventListener("resize", handleResize);

// ✅ Iniciar un setInterval
const interval = setInterval(() => {
  console.log("👉 Midi-chlorianos subiendo...");
}, 1000);


// Limpieza en return
return () => {
  clearInterval(interval);
  window.removeEventListener("resize", handleResize);
};
```



⚠️ ¿Qué tipo de cosas deberías hacer en esta fase de montaje?



Recap rápido con analogía Star Wars

Fase	Qué pasa	Star Wars	
Mounting	Código dentro de <code>useEffect</code> sin dependencias (<code>[]</code>) se ejecuta	<i>Anakin entra al templo Jedi</i>	
Unmounting	El <code>return () => {}</code> se ejecuta cuando el componente se va	<i>Anakin se convierte en Vader y se retira al Lado Oscuro</i>	



```
// NO HAGAS ESTO  
useEffect(async () => {  
  // ✖  
}, []);
```



solución:

```
useEffect(() => {  
  const fetchData = async () => {  
    const res = await fetch('...');  
    // ...  
  };  
  fetchData();  
}, []);
```

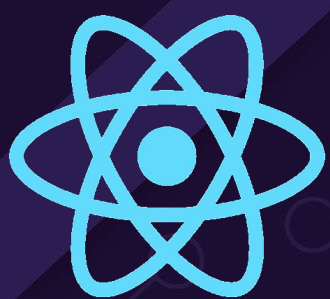


📦 Extra: si quieres ser más pro.

🧵 Separación por tipos de efectos

Puedes usar efectos *sincrónicos* vs *asincrónicos*, cómo usar *async* en *useEffect*, no debes usar *async* directamente en el hook.





Conclusiones

DEV.F
DESARROLLAMOS(PERSONAS);



✓ 1. Comprender las fases del ciclo de vida de un componente en React

🧐 Metáfora: Un Jedi en entrenamiento

Imagina que tu componente es un joven padawan.



Fase del Ciclo	Analogía Star Wars	Qué pasa en React
Montaje	Anakin entra al Templo Jedi	El componente se crea y aparece en el DOM
Actualización	Anakin entrena y cambia con el tiempo	El componente recibe nuevas props o cambia su estado
Desmontaje	Se convierte en Darth Vader y desaparece	El componente se elimina del DOM



✓ 2. Utilizar `useEffect()` para gestionar efectos en montaje, actualización y desmontaje

🔧 ¿Qué es `useEffect()`?

Es un hook que nos permite decirle a React:

"Ey, después de renderizar, haz esto... y si cambian estas cosas, vuelve a hacerlo."

⚙️ Sintaxis básica:

```
useEffect(() => {  
  // efecto principal  
  
  return () => {  
    // cleanup  
  };  
}, [dependencias]);
```





Ejemplo completo Star Wars



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

function JediStatus() {
  const [forceLevel, setForceLevel] = useState(0);

  // Montaje
  useEffect(() => {
    console.log('🟢 El joven padawan ha llegado (Mount)');

    return () => {
      console.log('🔴 Se ha unido a la Fuerza (Unmount)');
    };
  }, []);

  // Actualización
  useEffect(() => {
    if (forceLevel > 0) {
      console.log('📄 Midi-chlorianos actualizados: ${forceLevel}');
    }
  }, [forceLevel]);

  return (
    <div>
      <h2>Fuerza actual: {forceLevel}</h2>
      <button onClick={() => setForceLevel(forceLevel + 1)}>Entrenar 🧙</button>
    </div>
  );
}
```



✓ 3. Identificar cuándo y cómo limpiar efectos secundarios

🧹 ¿Qué es el "cleanup"?

Es una forma de evitar efectos fantasma: **conexiones abiertas, timers sin cerrar, eventos duplicados... caos absoluto en la galaxia** 🪐

🧠 Se usa para:

- **setInterval / setTimeout**
- **Listeners (addEventListener)**
- **Conexiones como WebSockets**

💡 Ejemplo con evento del teclado:

```
useEffect(() => {  
  const handleKey = (e) => {  
    console.log(`Tecla presionada: ${e.key}`);  
  };  
  
  window.addEventListener('keydown', handleKey);  
  
  return () => {  
    window.removeEventListener('keydown', handleKey);  
    console.log("🗑️ Listener eliminado");  
  };  
}, []);
```

Sin el cleanup, si el componente se monta/desmonta varias veces, ¡el evento se duplicaría y te convertirías en el Palpatine del rendimiento! ⚡



✓ 4. Aplicar conocimientos del ciclo de vida para mejorar la performance y organización del código



Buenas prácticas:

Caso	Qué hacer	Ejemplo
Efecto que solo se ejecuta 1 vez	Dejar el array de dependencias vacío (<code>[]</code>)	<code>fetch()</code> inicial
Efecto que depende de props/estado	Agregarlo en el array de dependencias	Validar formulario al escribir
Cleanup necesario	Usar <code>return () => { ... }</code> dentro del efecto	<code>clearInterval</code> , <code>removeEventListener</code>



★ Pro Tip: Separa responsabilidades

Evita mezclar muchos efectos en uno solo. Por ejemplo:

```
useEffect(() => {  
  // ✅ solo esto maneja eventos  
}, []);  
  
useEffect(() => {  
  // ✅ este solo reacciona al estado  
}, [state]);
```

⚡ Mejora de performance:
evitar renders innecesarios

```
useEffect(() => {  
  console.log("¡Me actualicé!");  
}, [importantProp]); // ✅ Solo se ejecuta si "importantProp" cambia
```



| Ejemplo de clase 🔥

🚀 simular el juego “¿Quién es ese Pokémon?” con:

- 🎮 Adivinanza basada en el nombre del Pokémon
- 🎨 Estilos bien chingones usando Bootstrap (bg-dark, btn-warning, form-control, sombras y más)
- 🔦 Imagen en modo silueta usando **filter: brightness(0)** hasta adivinar
- 🎉 Feedback visual y botón para pasar al siguiente Pokémon

¿Quién es ese Pokémon? 🤔



arbok

Adivinar

Siguiente Pokémon



¡Correcto! Eres un maestro Pokémon.



REACT JS

