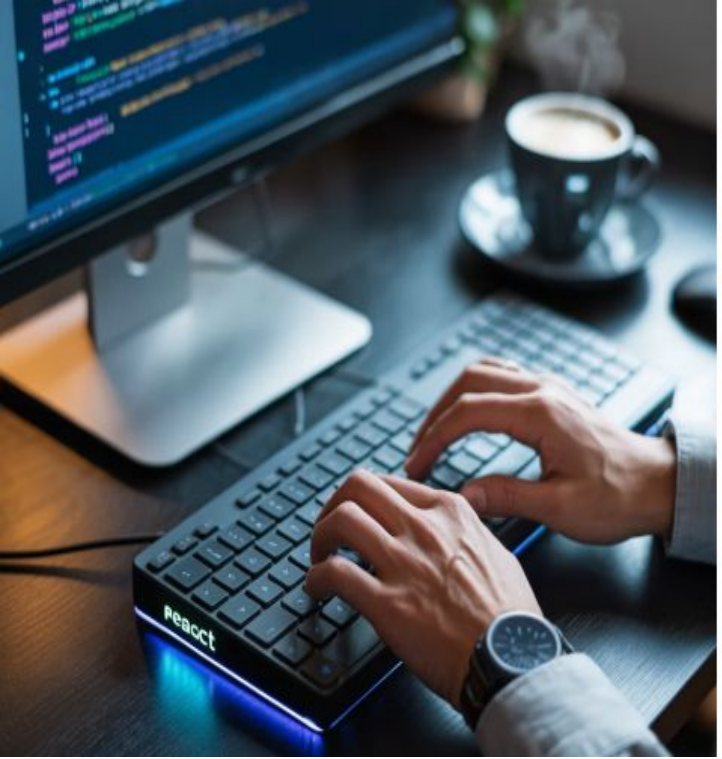
BOOTCAMP

# Generación T

streambe




# Dominando useEffect en React: Efectos Secundarios en la Práctica

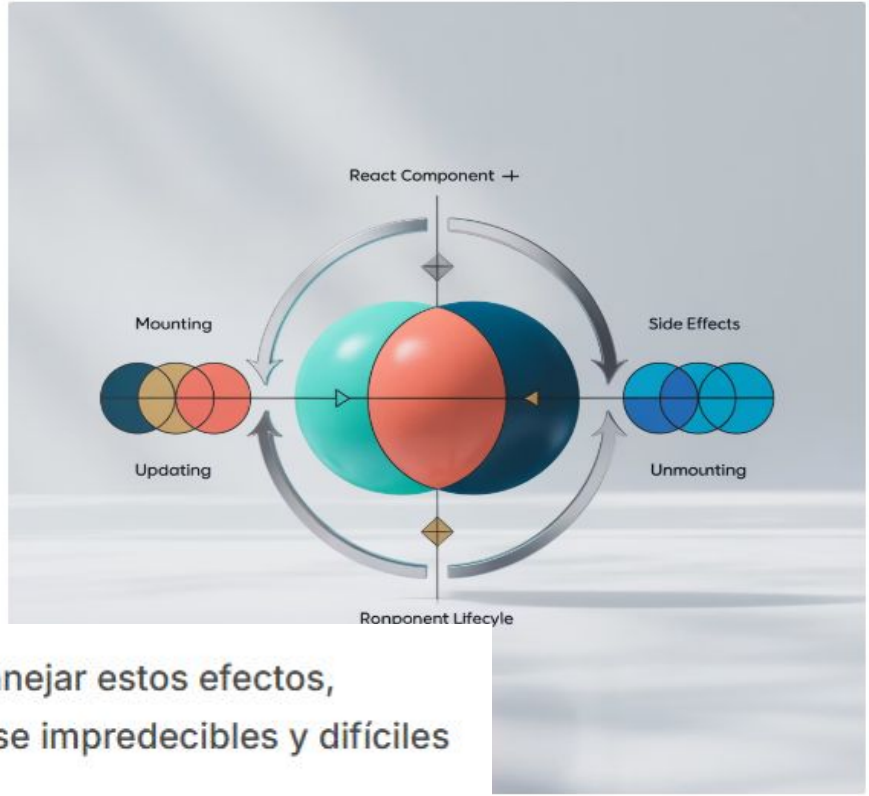
Este curso está diseñado para ayudarte a comprender y dominar uno de los hooks más importantes y complejos de React: useEffect. A lo largo de estas sesiones, exploraremos cómo manejar operaciones asíncronas, consumir APIs y gestionar recursos externos en tus aplicaciones de React.

# ¿Qué son los side effects en React?

Los **side effects** (efectos secundarios) son operaciones que ocurren fuera del flujo principal de renderizado de React y pueden afectar a otros componentes o interactuar con sistemas externos.

Mientras que el propósito principal de un componente es renderizar UI basada en props y estado, los efectos secundarios manejan todo lo demás:

- Peticiones a APIs (fetch)
- Manipulación directa del DOM
- Suscripciones a eventos
- Timers e intervalos
- Almacenamiento en localStorage




Sin un mecanismo adecuado para manejar estos efectos, nuestras aplicaciones podrían volverse impredecibles y difíciles de depurar.

# Introducción a useEffect: Sintaxis y funcionamiento

### Sintaxis Básica

```
useEffect(() => {
  // Código del efecto
  console.log('Componente renderizado');
});
```

Se ejecuta **después de cada renderizado** del componente.

### Con Dependencias Vacías

```
useEffect(() => {
  // Código del efecto
  console.log('Componente montado');
}, []);
```

Se ejecuta **solo una vez** al montar el componente.

### Con Dependencias

```
useEffect(() => {
  // Código del efecto
  console.log(`Valor actualizado: ${valor}`);
}, [valor]);
```

Se ejecuta cuando cambia la variable especificada en el array de dependencias.

El hook useEffect nos permite sincronizar nuestro componente con sistemas externos y ejecutar código en momentos específicos del ciclo de vida del componente.

# Caso práctico: Consumiendo APIs con useEffect

### Patrón común para fetch de datos

```
useEffect(() => {
  // Estado de carga
  setLoading(true);

  fetch('https://pokeapi.co/api/v2/pokemon/1')
    .then(response => response.json())
    .then(data => {
      setPokemon(data);
      setLoading(false);
    })
    .catch(error => {
      setError(error);
      setLoading(false);
    });
}, []);
```

Este patrón te permite manejar los tres estados principales de una petición: carga, éxito y error.


Puntos importantes:

- Usa el array de dependencias vacío `[]` para evitar bucles infinitos
- Considera usar `AbortController` para cancelar peticiones pendientes
- Maneja siempre los errores con try/catch o .catch()


# Gestión de timers e intervalos con useEffect

## Crear el Intervalo

```
useEffect(() => {
  const intervalo = setInterval(() => {
    setSegundos(prev => prev + 1);
  }, 1000);
}, []);
```

## Limpiar el Intervalo

```
useEffect(() => {
  const intervalo = setInterval(() => {
    setSegundos(prev => prev + 1);
  }, 1000);

  return () => clearInterval(intervalo);
}, []);
```

## Evitar Memory Leaks

La función de limpieza evita que el intervalo continúe ejecutándose después de que el componente se desmonte.

# La función de limpieza (cleanup)

## ¿Qué es la función de limpieza?

Es una función que se retorna dentro del callback de useEffect y se ejecuta:

- Antes de ejecutar el efecto nuevamente (si las dependencias cambian)
- Cuando el componente se desmonta

## ¿Cuándo es necesaria?

- Al crear suscripciones (event listeners)
- Al establecer timers o intervalos
- Al abrir conexiones (WebSockets)
- Cuando necesitamos cancelar peticiones fetch

```
useEffect(() => {
  // 1. Se ejecuta el efecto
  const handleResize = () => {
    setWindowSize(window.innerWidth);
  };

  window.addEventListener('resize', handleResize);

  // 2. Se retorna función de limpieza
  return () => {
    window.removeEventListener('resize',
      handleResize);
  };
}, []);
```

Sin esta limpieza, cada vez que el componente se renderice se añadiría un nuevo event listener, causando memory leaks y comportamientos inesperados.

# Orden de ejecución en useEffect

1. **Montaje del componente**
   React renderiza el componente por primera vez, mostrando la UI inicial.
2. **Ejecución del efecto**
   Después del renderizado, React ejecuta los efectos definidos con useEffect.
3. **Actualización del estado**
   Si el efecto actualiza algún estado, se programa un nuevo renderizado.
4. **Re-renderizado**
   React actualiza el DOM con los nuevos valores.
5. **Limpieza del efecto anterior**
   Antes de ejecutar el efecto nuevamente, se ejecuta la función de limpieza del efecto anterior.
6. **Ejecución del nuevo efecto**
   Se ejecuta el efecto con los valores actualizados.
7. **Desmontaje**
   Cuando el componente se elimina, se ejecuta la función de limpieza por última vez.

# Ejercicio Práctico: Carga de datos desde API

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

function PokemonViewer() {
  const [pokemon, setPokemon] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  const [id, setId] = useState(1);

  useEffect(() => {
    setLoading(true);

    fetch(`https://pokeapi.co/api/v2/pokemon/${id}`)
      .then(response => {
        if (!response.ok) throw new Error('No se pudo cargar el
Pokémon');
        return response.json();
      })
      .then(data => {
        setPokemon(data);
        setLoading(false);
      })
      .catch(error => {
        setError(error.message);
        setLoading(false);
      });
  }, [id]); // Se ejecuta cuando cambia el ID

  // Renderizado condicional según el estado
  if (loading) return
```

Cargando...

; if (error) return

Error: {error}

; if (!pokemon) return null; return (

# {pokemon.name}

setId(prev => prev + 1)}>Siguiente

); }

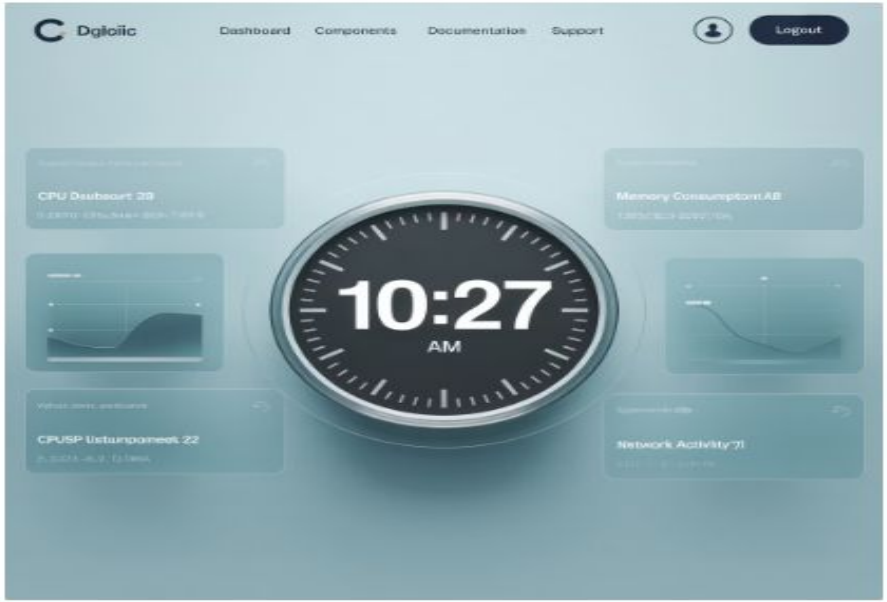
# Ejercicio Práctico: Reloj en tiempo real

### Implementación de un reloj digital

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

function Reloj() {
  const [tiempo, setTiempo] = useState(new Date());

  useEffect(() => {
    const intervalId = setInterval(() => {
      setTiempo(new Date());
    }, 1000);

    // Función de limpieza
    return () => {
      clearInterval(intervalId);
      console.log('Reloj detenido');
    };
  }, []); // Array vacío -> solo al montar

  return (
```

## Hora actual:

```
    {tiempo.toLocaleTimeString()}
  ); }
```




Este componente:

- Establece un intervalo que se ejecuta cada segundo
- Actualiza el estado con la hora actual
- Limpia correctamente el intervalo cuando el componente se desmonta
- Usa el formato local del navegador para mostrar la hora

Prueba a implementar un botón que monte/desmonte este componente para verificar que la limpieza funciona correctamente.

# Errores comunes y buenas prácticas con useEffect

## Bucles infinitos

Causados por:

- Olvidar el array de dependencias
- Actualizar estados en useEffect que están en sus dependencias

```
// MAL
```

# ¿Alguna duda?


streambe GENERACIÓN T

# Muchas gracias.

streambe GENERACIÓN T

# Nuestras Redes

www.generaciont.org
www.streambe.com
www.instagram.com/generaciont_ar
www.tiktok.com/@generaciont
generaciont@generaciont.org
Cel: 11 61331747

GENERACIÓN T

streambe