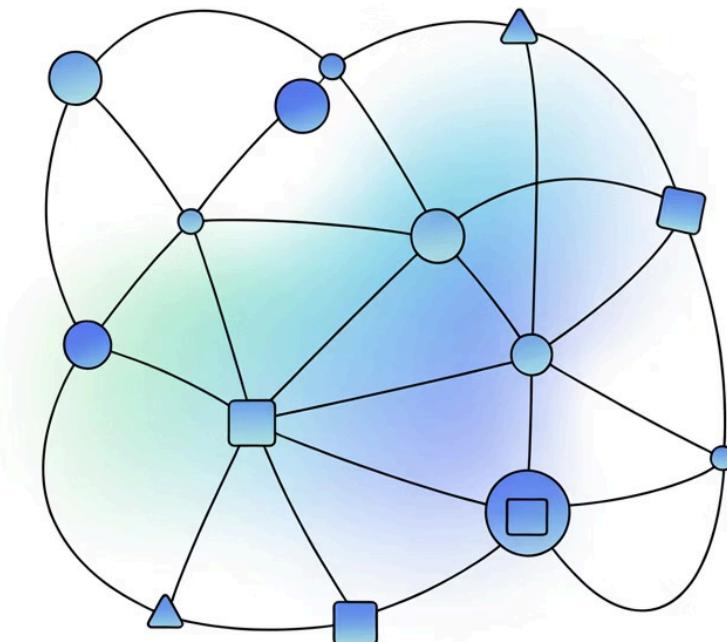


Curso de Desarrollo Frontend con React

Seamless API Integration

[Explore our solutions](#)

Consumo de API en React con **useEffect**

¿Qué aprenderemos hoy?

- 1 APIs y comunicación frontend-backend
- 2 Hook useEffect en profundidad
- 3 Fetch y solicitudes HTTP
- 4 Manejo de estado completo
- 5 Buenas prácticas

Fundamentos del intercambio de datos

Ciclo de vida y ejecución de efectos

Técnicas de consumo de datos

Loading, errores y datos

Código limpio y reutilizable

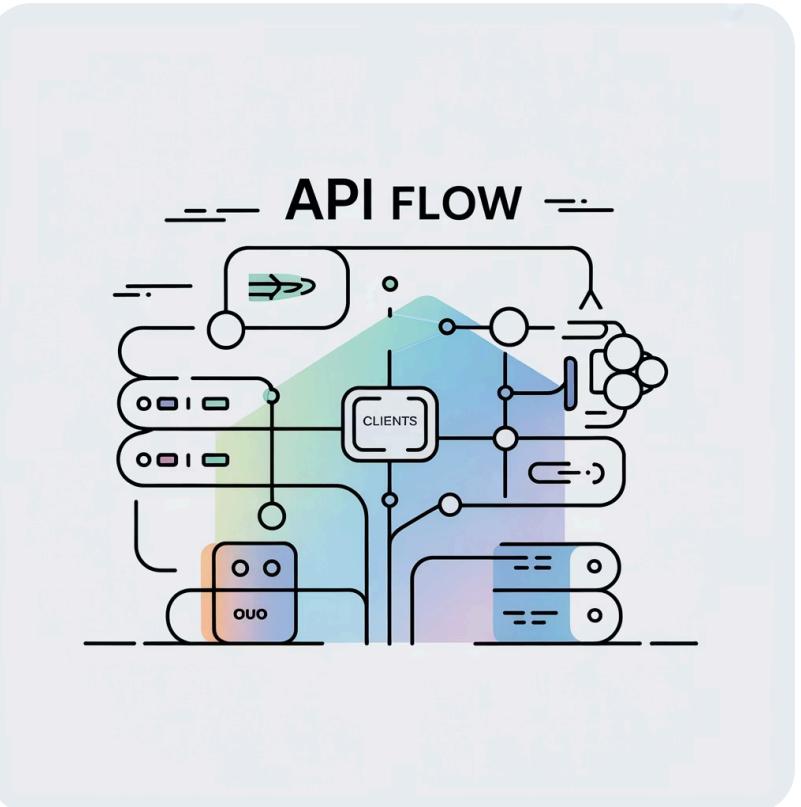
Introducción al Consumo de Datos

¿Qué es una API?

Una **API (Application Programming Interface)** es un conjunto de reglas y protocolos que permite la comunicación entre diferentes aplicaciones o servicios.

En el contexto web, las APIs actúan como un [puente de comunicación](#) entre tu aplicación React (frontend) y el servidor (backend).

Piensa en la API como un **mesero en un restaurante**: tú pides el plato (solicitud), el mesero lleva tu pedido a la cocina (servidor), y te trae la comida (respuesta).



El papel de las APIs en aplicaciones modernas



Acceso a datos

Obtener información almacenada en bases de datos remotas



Sincronización

Mantener datos actualizados entre múltiples dispositivos



Seguridad

Controlar acceso y permisos de forma centralizada

Datos estáticos vs datos dinámicos

Datos estáticos

- Definidos directamente en el código
- No cambian sin modificar el código fuente
- Ideales para contenido fijo como menús
- Carga instantánea

```
const usuarios = [  
  {  
    id: 1,  
    nombre: "Ana"  
  },  
  {  
    id: 2,  
    nombre: "Carlos"  
  }  
];
```

Datos desde API

- Se obtienen desde servicios externos
- Cambian dinámicamente
- Reflejan estado actual del sistema
- Requieren tiempo de carga

```
// Se obtienen en tiempo real  
fetch('/api/usuarios')  
  .then(res => res.json())  
  .then(datos => {  
    // ...  
  });
```



Casos de uso comunes en Colombia

E-commerce

Tiendas como Mercado Libre obtienen productos, precios y disponibilidad desde sus APIs internas

Fintech

Apps como Nequi consultan saldos bancarios y movimientos a través de APIs seguras

Delivery

Rappi y Domicilios.com obtienen menús de restaurantes y estado de pedidos en tiempo real

Hook useEffect

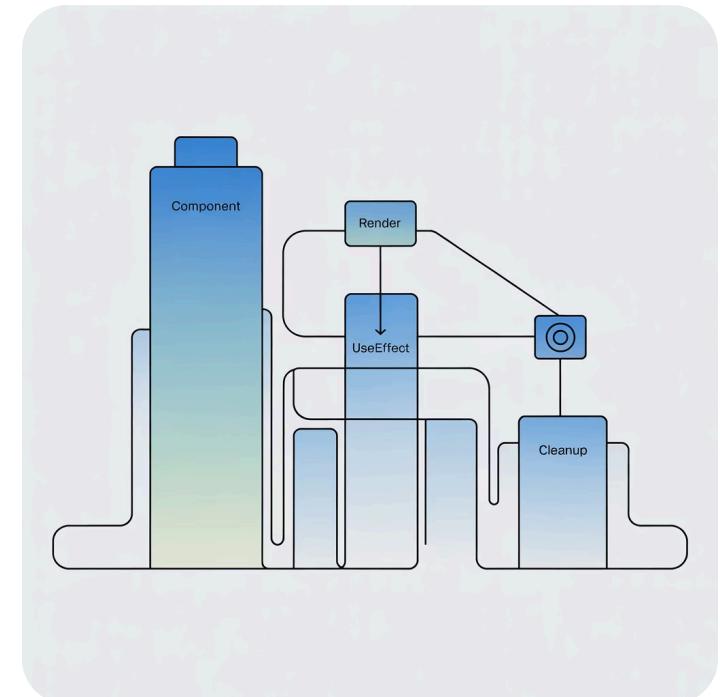
¿Qué es useEffect?

useEffect es un Hook de React que permite realizar **efectos secundarios** en componentes funcionales.

Los efectos secundarios incluyen:

- Llamadas a APIs
- Modificación del DOM
- Configuración de timers
- Suscripciones a eventos

Es el equivalente a `componentDidMount`, `componentDidUpdate` y `componentWillUnmount` en componentes de clase.



Sintaxis básica de useEffect

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

function MiComponente() {
  const [datos, setDatos] = useState([]);

  useEffect(() => {
    // Este código se ejecuta después del renderizado
    console.log('Componente montado o actualizado');
  });

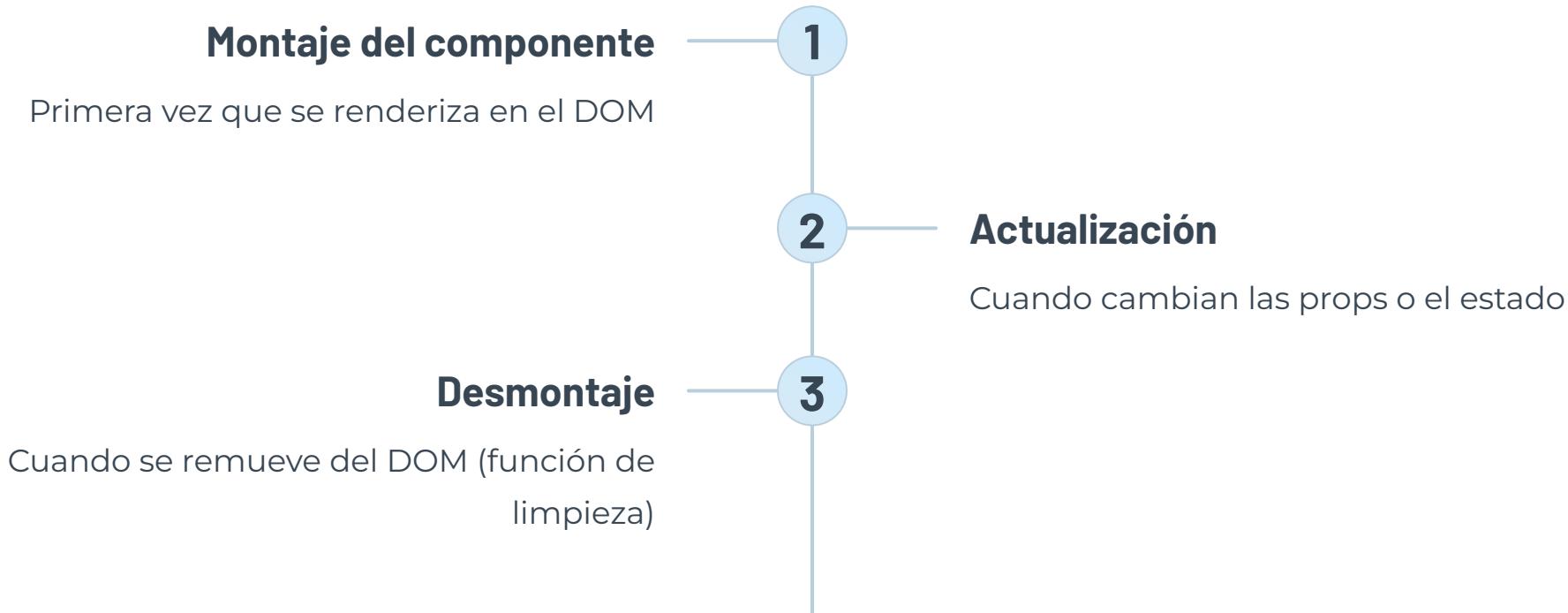
  return
}
```

Mi componente

; }

Importante: Sin array de dependencias, useEffect se ejecuta después de cada renderizado.

Cuándo se ejecuta useEffect



Array de dependencias: la clave del control

Sin dependencias

```
useEffect(() => {  
  // Se ejecuta siempre  
});
```

Cada renderizado

Array vacío

```
useEffect(() => {  
  // Se ejecuta una vez  
}, []);
```

Solo al montar

Con dependencias

```
useEffect(() => {  
  // Se ejecuta cuando  
  // cambia userId  
, [userId]);
```

Al cambiar valores

Ejemplo práctico: contador con efecto

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

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

  // Se ejecuta solo al montar el componente
  useEffect(() => {
    document.title = `Contador iniciado`;
  }, []);

  // Se ejecuta cada vez que count cambia
  useEffect(() => {
    document.title = `Contador: ${count}`;
  }, [count]);

  return (
    <div>
      <p>Has hecho clic {count} veces</p>
      <button onClick={() => setCount(count + 1)}>
        Incrementar
      </button>
    </div>
  );
}
```

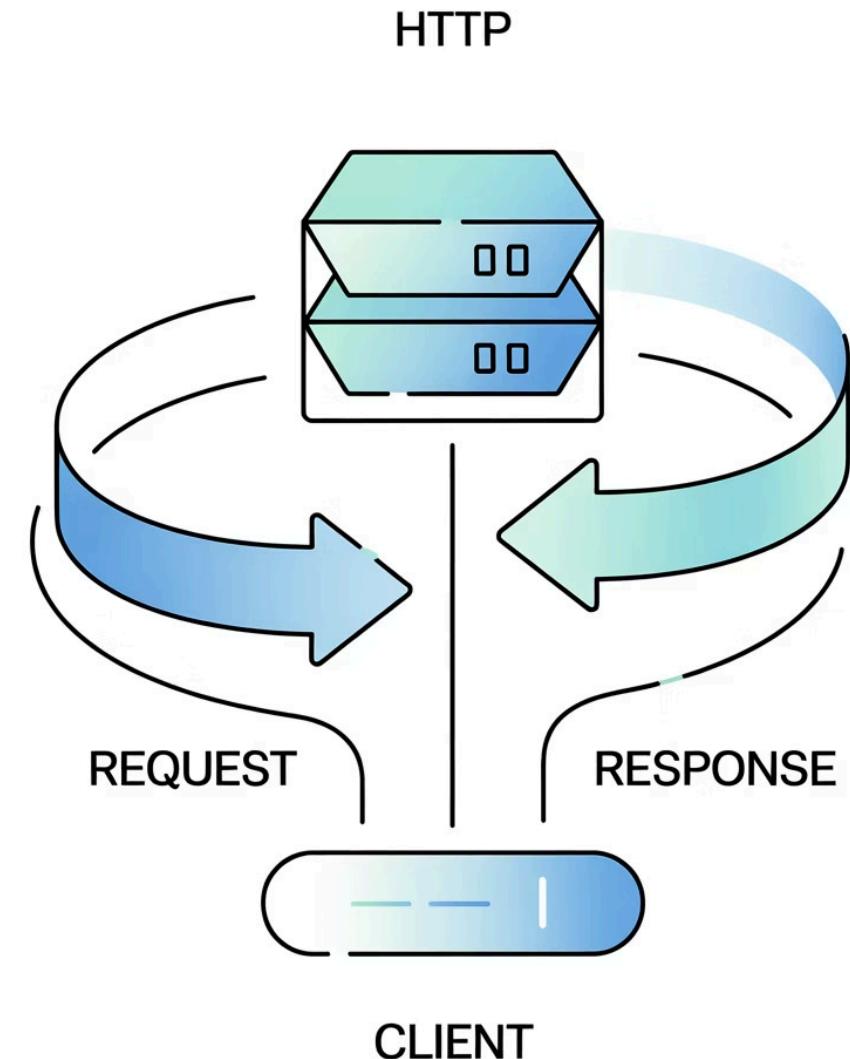
Fetch y HTTP

¿Qué es fetch()?

Fetch es una API nativa del navegador que permite realizar solicitudes HTTP de forma asíncrona. Es el método moderno para comunicarse con servidores.

Ventajas de fetch:

- Nativo en navegadores modernos
- Basado en Promises
- Sintaxis limpia y legible
- Soporte completo para HTTP/HTTPS



Sintaxis básica de fetch

```
// Solicitud GET básica
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => response.json())
  .then(data => {
    console.log('Datos recibidos:', data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
});
```

Flujo: solicitud → respuesta → conversión a JSON → manejo de datos

Manejo de promesas: `.then()` y `.catch()`



Las promesas permiten manejar operaciones asíncronas de forma elegante, evitando el "callback hell".

Async/await: sintaxis moderna

Con .then()

```
useEffect(() => {
  fetch('/api/users')
    .then(res => res.json())
    .then(data => {
      setUsuarios(data);
    })
    .catch(error => {
      setError(error.message);
    });
}, []);
```

Con async/await

```
useEffect(() => {
  const obtenerUsuarios = async () => {
    try {
      const response = await fetch('/api/users');
      const data = await response.json();
      setUsuarios(data);
    } catch (error) {
      setError(error.message);
    }
  };
  obtenerUsuarios();
}, []);
```

¿Por qué `async/await` es mejor?

Legibilidad

El código se lee de forma secuencial, como código síncrono

Manejo de errores

Un solo bloque `try/catch` para manejar todas las excepciones

Debugging

Más fácil de debuggear con breakpoints

Formato JSON: el lenguaje de las APIs

JSON (JavaScript Object Notation) es el formato estándar para intercambiar datos entre cliente y servidor.

Características:

- Fácil de leer y escribir
- Ligero y eficiente
- Compatible con JavaScript
- Soportado universalmente

```
{  
  "id": 1,  
  "nombre": "Juan Pérez",  
  "email": "juan@email.com",  
  "ciudad": "Bogotá",  
  "activo": true,  
  "hobbies": ["fútbol", "música"]  
}
```

Manejo de Estado

Los tres estados esenciales

Loading
Indica que la solicitud está en proceso



Error

Maneja fallos en la comunicación

Data

Almacena la información recibida

Configuración inicial del estado

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

function ListaUsuarios() {
  // Estado para los datos
  const [usuarios, setUsuarios] = useState([]);

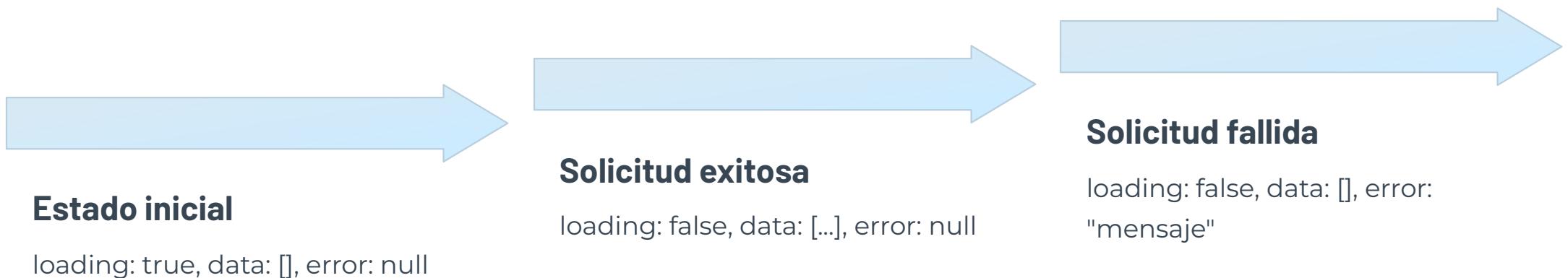
  // Estado para el loading
  const [loading, setLoading] = useState(true);

  // Estado para errores
  const [error, setError] = useState(null);

  return (
    <div>
      {/* Componente renderizado */}
    </div>
  );
}
```

Tip: Inicializa arrays vacíos [] para listas y null para objetos únicos.

Flujo completo de manejo de estado



Implementación práctica del estado

```
useEffect(() => {
  const obtenerUsuarios = async () => {
    try {
      setLoading(true);
      setError(null);

      const response = await fetch('/api/usuarios');

      if (!response.ok) {
        throw new Error(`Error ${response.status}: ${response.statusText}`);
      }

      const data = await response.json();
      setUsuarios(data);

    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  obtenerUsuarios();
}, []);
```

Renderizado condicional basado en estado

```
return (
  <div>
    <h1>Lista de Usuarios</h1>

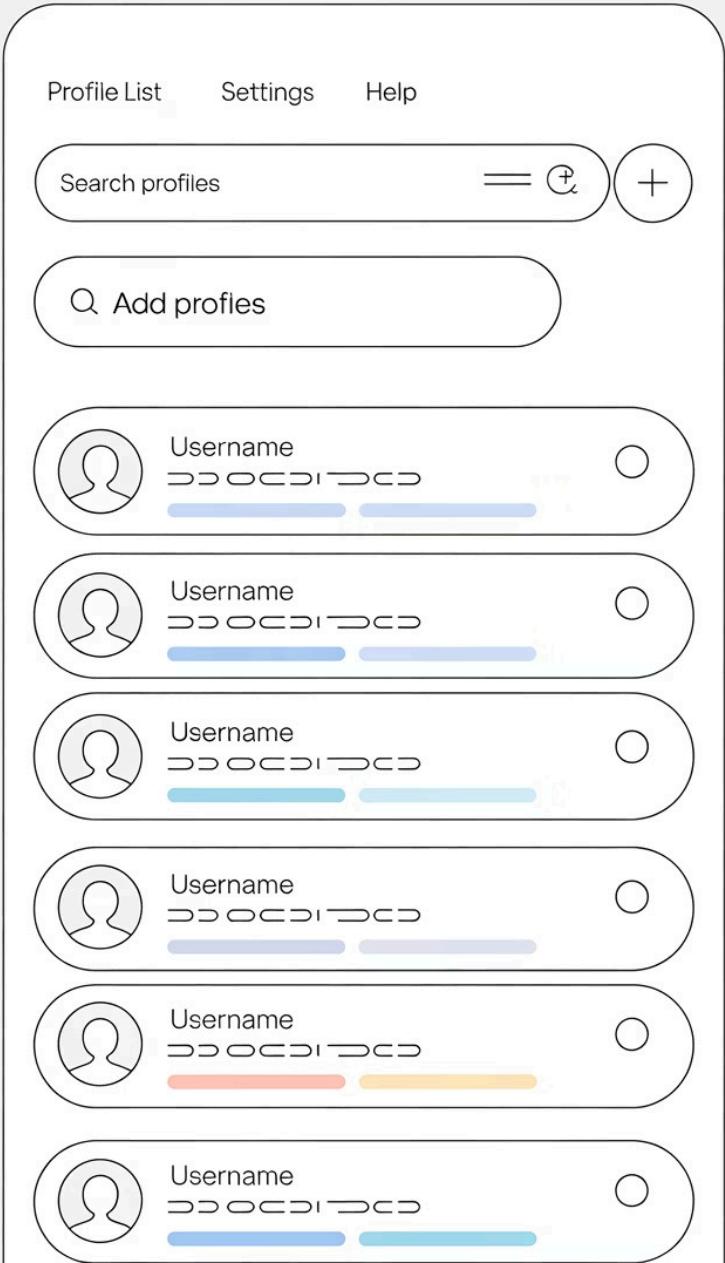
    {loading && (
      <div className="loading">
        <p>Cargando usuarios...</p>
      </div>
    )}

    {error && (
      <div className="error">
        <p>Error: {error}</p>
        <button onClick={reintentarCarga}>Reintentar</button>
      </div>
    )}

    {!loading && !error && usuarios.length === 0 && (
      <p>No se encontraron usuarios.</p>
    )}

    {!loading && !error && usuarios.length > 0 && (
      <ul>
        {usuarios.map(usuario => (
          <li key={usuario.id}>{usuario.nombre}</li>
        )));
      </ul>
    )}
  </div>
);
```

Ejemplo Práctico



Objetivo: Lista de usuarios desde API pública

Vamos a crear un componente que obtenga usuarios desde JSONPlaceholder, una API pública perfecta para pruebas y aprendizaje.

API endpoint: <https://jsonplaceholder.typicode.com/users>

Funcionalidades: carga de datos, indicador de loading, manejo de errores, y botón de recarga.

Paso 1: Estructura del componente

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

function ListaUsuarios() {
  // Estados necesarios
  const [usuarios, setUsuarios] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  // useEffect para cargar datos
  useEffect(() => {
    cargarUsuarios();
  }, []);

  const cargarUsuarios = async () => {
    // Lógica de carga aquí
  };

  return (
    // JSX del componente
  );
}

export default ListaUsuarios;
```

Paso 2: Función de carga de datos

```
const cargarUsuarios = async () => {
  try {
    setLoading(true);
    setError(null);

    const response = await fetch('https://jsonplaceholder.typicode.com/users');

    // Verificar si la respuesta es exitosa
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();
    setUsuarios(data);

  } catch (err) {
    setError('Error al cargar los usuarios: ' + err.message);
    console.error('Error detallado:', err);

  } finally {
    setLoading(false);
  }
};
```

Paso 3: Renderizado del componente

```
return (
  <div className="container">
    <div className="header">
      <h1>Usuarios Registrados</h1>
      <button
        onClick={cargarUsuarios}
        disabled={loading}
        className="btn-reload"
      >
        {loading ? 'Cargando...' : 'Recargar'}
      </button>
    </div>

    {loading && (
      <div className="loading-spinner">
        <p>Cargando datos, por favor espere...</p>
      </div>
    )}

    {error && !loading && (
      <div className="error-message">
        <p>{error}</p>
        <button onClick={cargarUsuarios} className="btn-retry">Reintentar</button>
      </div>
    )}

    {!loading && !error && (
      <ul className="user-list">
        {usuarios.map(usuario => (
          <li key={usuario.id}>
            <strong>{usuario.name}</strong> - {usuario.email}
          </li>
        )));
      </ul>
    )}
  </div>
);
```

Paso 4: Componente UserCard

```
function UserCard({ usuario }) {
  return (
    <div className="user-card">
      <div className="user-header">
        <h3>{usuario.name}</h3>
        <span className="username">@{usuario.username}</span>
      </div>

      <div className="user-info">
        <p>
          <strong>Email:</strong> {usuario.email}
        </p>
        <p>
          <strong>Ciudad:</strong> {usuario.address.city}
        </p>
        <p>
          <strong>Empresa:</strong> {usuario.company.name}
        </p>
      </div>

      <div className="user-actions">
        <a href={`mailto:${usuario.email}`} className="btn-contact">
          Contactar
        </a>
      </div>
    </div>
  );
}
```

Resultado visual esperado

La aplicación mostrará:

- Header con título y botón de recarga
- Spinner de carga durante las solicitudes
- Grid de tarjetas con información de usuarios
- Mensajes de error claros y accionables
- Diseño responsive y amigable

The screenshot shows a mobile application interface. At the top is a header bar with a search bar labeled "Search users" and a button labeled "+ Add user". Below the header is a navigation bar with tabs: "Dashboard" (selected), "Users" (highlighted in yellow), "Settings" (highlighted in orange), and "Help". The main content area is titled "React User List" and displays a grid of 12 user cards. Each card features a placeholder user icon and a "User name" label. The cards are arranged in four rows of three. The background has a gradient from light blue at the top to light green at the bottom.

Manejo avanzado de errores

1

Errores de red

Sin conexión a internet o servidor no disponible

2

Errores HTTP

Status 404, 500, 401 - verificar response.ok

3

Errores de formato

JSON malformado - validar antes de parsear

4

Timeout

Solicitudes que toman demasiado tiempo

Buenas Prácticas

Separación de responsabilidades

✗ Todo en el componente

```
function MiComponente() {  
  const [data, setData] = useState([]);  
  
  useEffect(() => {  
    fetch('/api/data')  
      .then(res => res.json())  
      .then(setData);  
  }, []);  
  
  return <div>...</div>;  
}
```

✓ Lógica separada

```
// services/api.js  
export const obtenerDatos = async () => {  
  const response = await fetch('/api/data');  
  return response.json();  
};  
  
// MiComponente.jsx  
function MiComponente() {  
  const [data, setData] = useState([]);  
  
  useEffect(() => {  
    obtenerDatos().then(setData);  
  }, []);  
  
  return <div>...</div>;  
}
```

Custom Hook para reutilización

```
// hooks/useApi.js
import { useState, useEffect } from 'react';

export function useApi(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const cargarDatos = async () => {
      try {
        setLoading(true);
        const response = await fetch(url);
        if (!response.ok) throw new Error('Error en la solicitud');
        const resultado = await response.json();
        setData(resultado);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };
    cargarDatos();
  }, [url]);

  return { data, loading, error };
}
```

Evitando llamadas infinitas

⚠ Problema común

```
useEffect(() => {  
  fetch('/api/data').then(setData);  
}); // Sin array de dependencias
```

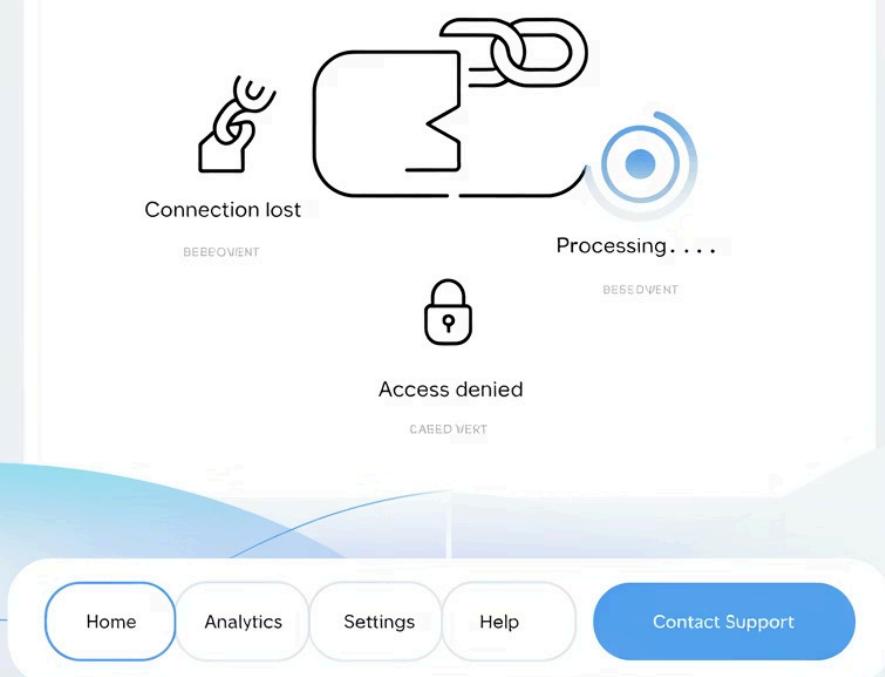
Se ejecuta en cada renderizado, creando un bucle infinito

✓ Solución

```
useEffect(() => {  
  fetch('/api/data').then(setData);  
}, []); // Array vacío
```

Se ejecuta solo una vez al montar el componente

Error Messaages



Mensajes claros para el usuario

Loading

"Cargando información..." -
con indicador visual

Error

"No pudimos cargar los datos.
Verifica tu conexión e intenta de nuevo."

Sin datos

"No hay información disponible en este momento."

Reflexión final

¿Por qué es importante manejar los estados de carga y error al consumir datos de una API en React?

El manejo adecuado de estados es crucial porque:

- **Experiencia del usuario:** Los usuarios saben qué está pasando en todo momento
- **Confiabilidad:** La app funciona correctamente incluso cuando hay problemas de red
- **Profesionalismo:** Demuestra atención al detalle y calidad en el desarrollo
- **Debugging:** Facilita identificar y resolver problemas durante el desarrollo

#EDCOUNIANDES

<https://educacioncontinua.uniandes.edu.co/>

Contacto: educacion.continua@uniandes.edu.co

© - Derechos Reservados: La presente obra, y en general todos sus contenidos, se encuentran protegidos por las normas internacionales y nacionales vigentes sobre propiedad Intelectual, por lo tanto su utilización parcial o total, reproducción, comunicación pública, transformación, distribución, alquiler, préstamo público e importación, total o parcial, en todo o en parte, en formato impreso o digital y en cualquier formato conocido o por conocer, se encuentran prohibidos, y solo serán lícitos en la medida en que se cuente con la autorización previa y expresa por escrito de la Universidad de los Andes.