Clase 8 — React: Fetch y manejo de carga

Diego Muñoz

7 de octubre de 2025

Introducción

- Obtener datos reales con `fetch`.
- Controlar estados: `isLoading` y `error`.
- Usar `useState` + `useEffect` de forma metódica.
- Patrón básico reutilizable para consumir APIs.

`fetch` (idea general)

- API nativa para peticiones HTTP.
- Retorna *Promise* → `.then()`.
  Ojo: códigos 4xx/5xx no lanzan error por sí solos.

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(r => r.json())
  .then(data => console.log(data));
```

`useEffect` + `fetch` (mínimo viable)

```jsx
import { useState, useEffect } from "react";

function Users() {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(r => r.json()).then(setUsers);
  }, []);
  return (
    <ul> {users.map(u => <li key={u.id}>{u.name}</li>)} </ul>
  );
}
```

Agregar `isLoading`

```
function Users() {
  const [users, setUsers] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(r => r.json()).then(data => setUsers(data))
      .finally(() => setIsLoading(false));
  }, []);
  if (isLoading) return <p>Cargando...</p>;
  return <ul>{users.map(u => <li key={u.id}>{u.name}</li>)}</ul>;
}
```

Manejo de `error` (1/2)

```
function Users() {
  const [users, setUsers]   = useState([]);
  const [isLoading, setIsLoading]   = useState(true);
  const [error, setError]   = useState(null);
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(r => {
        if (!r.ok) throw new Error("Respuesta no OK");
        return r.json();
      })
      .then(setUsers).catch(err => setError(err.message))
      .finally(() => setIsLoading(false));
  }, []);
```

Manejo de `error` (2/2)

```
  if (isLoading) return <p>Cargando...</p>;
  if (error)     return <p>Error: {error}</p>;
  if (users.length === 0) return <p>Sin datos.</p>;

  return (
    <ul>
      {users.map(u => <li key={u.id}>{u.name}</li>)}
    </ul>
  );
}
```

Visualización condicional (patrón)

```
if (isLoading) return <p>Cargando...</p>;
if (error) return <p>Error: {error}</p>;
if (!data?.length) return <p>Sin resultados.</p>;

// Render “feliz”
return <List data={data} />;
```

- Separar fases evita *ifs* anidados en el `return` principal.

Patrón reutilizable (3 estados)

```
const [data, setData]       = useState([]);
const [isLoading, setLoad]  = useState(true);
const [error, setError]     = useState(null);
```

1. Cargar datos.
2. Actualizar estados.
3. Renderizar según estado.

Buenas prácticas

1. Siempre feedback: `Cargando` / `Error` / `Sin datos`.
2. Nunca `fetch` en el cuerpo del componente.
3. Validar `response.ok` antes de `r.json()`.
4. Preparar UI para datos vacíos.

Proyecto de ejemplo

Referir a ejemplo en GitHub.

Resumen

- `fetch`: peticiones HTTP asincrónicas.
- `useEffect`: decide **cuándo** cargar.
- `useState`: administra `data`, `isLoading`, `error`.
- Patrón “3 estados” listo para reutilizar en toda la app.