React III

Controlled fields, `useEffect`

Javier Ribal del Río

2026-02-27

Table of contents

Contenido

- *Controlled Inputs*
- Concepto de Hook
- *Hook* `useEffect`
- Introducción a API para la descarga de datos

1 *Controlled Inputs*

1.1 Motivación

Hasta ahora se ha trabajado con eventos capturando acciones del usuario.

Sin embargo, al trabajar con formularios surge una cuestión fundamental:

> ¿Quién controla el valor del input?
> ¿El DOM o React?

Recordemos que en HTML puro o, el valor de un `<input>` está gestionado directamente por el DOM.

En React, podemos hacer que el valor del input esté controlado por el estado del componente.

A esto lo llamamos **controlled input**.

1.2 ¿Qué es un controlled input?

Un *controlled input* es un elemento de formulario cuyo valor:

- Está almacenado en el estado del componente
- Se actualiza mediante un evento (`onChange`)
- Se renderiza utilizando ese mismo estado

Única fuente de verdad (*single source of truth*).

1.3 Estructura del componente controlled

```
1  import { useState } from "react";
2  function FormularioNombre() {
3
4    const [nombre, setNombre] = useState("");
5
6    function manejarCambio(event) {
7      setNombre(event.target.value);
```

```
  }

  return (
    <div>
      <input type="text"
       value={nombre}
        onChange={manejarCambio}
      />
      <p>El nombre introducido es: {nombre}</p>
    </div>
  );
}
```

1.3.1 Análisis del ejemplo

El valor del input se sincroniza con el estado

1. Declaramos una variable de estado: nombre
2. Asociamos el atributo value del input a ese estado
3. Capturamos el evento onChange
4. Actualizamos el estado con setNombre

Cada vez que el estado cambia: El componente se vuelve a ejecutar El JSX se vuelve a generar El valor del input se sincroniza con el estado

1.4 Ejemplo Formulario

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

function FormularioSimple() {

  const [email, setEmail] = useState("");

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

  function handleSubmit(event) {
    event.preventDefault();   // Evita la recarga del navegador

    console.log("Formulario enviado");
    console.log("Email:", email);

    setEmail("");  // Limpieza opcional del campo
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
```

```
        value={email}
        onChange={handleChange}
      />
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Referencia: https://es.react.dev/reference/react-dom/components/input#controlling-an-input-with-a-state-variable

2 Hooks (Repaso)

- Función especial que permite modificar las propiedades internas de los componentes
- Características avanzadas
- Modificación del `life-cycle`
- Normalmente forma `use____`

3 `useEffect`

- `useEffect` es un Hook que permite ejecutar efectos secundarios dentro de un componente funcional.
- Un efecto secundario es cualquier operación que no sea simplemente devolver JSX (por ejemplo: peticiones a un servidor, temporizadores, suscripciones).
- Se ejecuta después de que el componente se renderiza en el DOM.

3.1 Métodos de uso de `useEffect`

3.1.1 1. Sin array de dependencias

```
useEffect(() => {
  console.log("Se ejecuta en cada render");
});
```

Comportamiento:

- Se ejecuta después de cada renderizado.
- No existe control sobre la frecuencia.
- Puede generar bucles si modifica estado sin control.

Interpretación conceptual:

El efecto acompaña siempre al render.

3.1.2 2. Con array vacío

```
useEffect(() => {
  console.log("Se ejecuta solo una vez");
}, []);
```

Comportamiento:

- Se ejecuta únicamente tras el primer render.
- Equivalente conceptual al "montaje" del componente.

Uso habitual:

- Peticiones a servidor
- Inicializaciones
- Suscripciones

Interpretación conceptual:

El efecto ocurre una única vez al crear el componente.

3.1.3 3. Con dependencias específicas

```
useEffect(() => {
  console.log("Se ejecuta cuando cambia contador");
}, [contador]);
```

Comportamiento:

- Se ejecuta tras el primer render.
- Se vuelve a ejecutar cuando cambia alguna variable del array.

Interpretación conceptual:

El efecto depende del estado indicado.

React compara el valor anterior con el nuevo y decide si debe ejecutar el efecto.

4 `fetch()`

Hasta ahora nuestros componentes eran completamente locales.

Sin embargo, en aplicaciones reales necesitamos:

- Descargar datos
- Conectarnos a servidores
- Trabajar con APIs externas

Para ello utilizamos `fetch()`.

4.1 Ejemplo mínimo de `fetch`

```
fetch("https://pokeapi.co/api/v2/pokemon/1")
  .then(response => response.json())
  .then(data => console.log(data));
```

Utiliza las *Promises*

Interpretación simple:

1. Se hace una petición a una URL
2. Se transforma la respuesta a formato JSON
3. Se accede a los datos

4.2 `fetch()` dentro de `useEffect`

En React, lo habitual es realizar la descarga cuando el componente se monta.

Para ello:

- Utilizamos `useEffect`
- Añadimos array vacío `[]`
- Guardamos los datos en el estado

4.3 Ejemplo integrado

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

function Pokemon() {

  const [pokemon, setPokemon] = useState(null);

  useEffect(() => {

    fetch("https://pokeapi.co/api/v2/pokemon/1")
      .then(response => response.json())
      .then(data => setPokemon(data));

  }, []);

  if (!pokemon) return <p>Cargando...</p>;

  return (
    <div>
      <h2>{pokemon.name}</h2>
      <img
        src={pokemon.sprites.front_default}
        alt={pokemon.name}
```

```
      />
    </div>
  );
}
```

4.4 Flujo conceptual

1. Render inicial
2. Se ejecuta `useEffect`
3. Se lanza la petición
4. Se actualiza el estado
5. Nuevo render con los datos

4.5 Advertencia importante

Si eliminamos el array vacío:

```
useEffect(() => {
  fetch("https://pokeapi.co/api/v2/pokemon/1")
    .then(res => res.json())
    .then(data => setPokemon(data));
});
```

El efecto se ejecutará en cada render.

Como actualizar estado provoca render, se generará un bucle infinito.

`useEffect` conecta React con el exterior.
`fetch()` permite traer datos del exterior.
El estado vuelve a controlar el render.

En la siguiente sección formalizaremos esto utilizando `async / await`.

5 `async` y `await`

Hasta ahora hemos utilizado `fetch()` con encadenamiento.

Existe una forma más clara y estructurada de escribir código asíncrono:

- `async`
- `await`

Su objetivo es hacer que el código sea más legible.

5.1 ¿Qué significa `async`?

- Se coloca delante de una función
- Indica que la función trabajará con operaciones asíncronas

```
async function ejemplo() {
  console.log("Función asíncrona");
}
```

5.2 ¿Qué significa `await`?

- Solo puede utilizarse dentro de una función `async`
- Detiene la ejecución hasta que la operación finaliza
- Hace que el código se lea de forma secuencial

5.3 Ejemplo básico con `fetch`

```
async function obtenerPokemon() {

  const response = await fetch(
    "https://pokeapi.co/api/v2/pokemon/1"
  );

  const data = await response.json();

  console.log(data);
}
```

Interpretación conceptual:

1. Se realiza la petición
2. Se espera la respuesta
3. Se transforma a JSON
4. Se trabaja con los datos

El código se lee de arriba hacia abajo.

5.4 `async / await` dentro de `useEffect`

En React, lo habitual es declarar una función interna y ejecutarla.

5.5 Ejemplo completo

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

function Pokemon() {
```

```
  const [pokemon, setPokemon] = useState(null);

  useEffect(() => {

    async function fetchData() {

      const response = await fetch(
        "https://pokeapi.co/api/v2/pokemon/1"
      );

      const data = await response.json();

      setPokemon(data);
    }

    fetchData();

  }, []);

  if (!pokemon) return <p>Cargando...</p>;

  return (
    <div>
      <h2>{pokemon.name}</h2>
      <img
        src={pokemon.sprites.front_default}
        alt={pokemon.name}
      />
    </div>
  );
}
```

5.6 ¿Por qué no hacer directamente?

```
useEffect(async () => {
  ...
}, []);
```

Porque `useEffect` no debe recibir una función asíncrona directamente.

Por eso declaramos la función dentro y luego la ejecutamos.

5.7 Flujo mental con `async / await`

1. Render inicial
2. Se ejecuta `useEffect`
3. Se llama a la función asíncrona

4. `await` espera la respuesta
5. Se actualiza el estado
6. Nuevo render

`async / await` no cambia lo que hace el programa.

Solo cambia la forma de escribirlo:

Más claro.
Más estructurado.
Más fácil de mantener.

6 Ejemplo con la Pokeapi

Poke-API

Download Code