1. Cómo transformar un `fetch` anidado a `async/await`

Cuando usas `fetch` de manera anidada, generalmente tienes varias promesas y haces algo como esto:

```
fetch('url')
  .then(response => response.json())
  .then(data => {
    return fetch('otra-url');
  })
  .then(response2 => response2.json())
  .then(data2 => {
    console.log(data2);
  })
  .catch(error => console.error(error));
```

Este enfoque está bien, pero puede ser más legible si lo conviertes en `async/await`.

Transformación con `async/await`:

```
async function obtenerDatos() {
  try {
    // Espera la respuesta del primer fetch
    let response = await fetch('url');
    let data = await response.json();

    // Ahora, espera la respuesta del segundo fetch
    let response2 = await fetch('otra-url');
    let data2 = await response2.json();

    console.log(data2);
  } catch (error) {
    console.error(error);
  }
}

obtenerDatos();
```

Explicación:

- Usamos `await` para esperar la respuesta de cada `fetch`, lo que hace que el código se ejecute de manera secuencial y más fácil de leer.
- En caso de error, usamos `try/catch` para capturarlo de forma más sencilla.

---

2. Fetch Blob y Race

- **Blob** es un tipo de objeto que representa datos binarios, como imágenes o archivos, que puedes manejar con JavaScript. Si quieres obtener un Blob de una respuesta de `fetch`, simplemente haces lo siguiente:

```
fetch('url-de-imagen')
  .then(response => response.blob())
  .then(blob => {
    // Aquí tienes el Blob
    console.log(blob);
  })
  .catch(error => console.error(error));
```

Con `async/await` se vería así:

```
async function obtenerBlob() {
  try {
    let response = await fetch('url-de-imagen');
    let blob = await response.blob();
    console.log(blob);
  } catch (error) {
    console.error(error);
  }
}

obtenerBlob();
```

Explicación:

- Cuando usas `response.blob()`, obtienes los datos como un objeto Blob, que puedes luego utilizar (por ejemplo, mostrar una imagen en una página web).

- **`Promise.race()`** es útil cuando quieres ejecutar varias promesas y te interesa la que se resuelva primero. Por ejemplo, si tienes múltiples `fetch` y quieres que se resuelva el primero que conteste:

```
const fetch1 = fetch('url1');
const fetch2 = fetch('url2');

Promise.race([fetch1, fetch2])
  .then(response => response.json())
  .then(data => {
    console.log('Primera respuesta:', data);
  })
  .catch(error => console.error(error));
```

Explicación:

- `Promise.race([fetch1, fetch2])` hace que se resuelva tan pronto como una de las promesas (en este caso, uno de los `fetch`) se complete.
- El código usa `then` para manejar la respuesta de la promesa ganadora.

Resumen:

- **`async/await`** hace que tu código sea más legible y fácil de entender cuando trabajas con promesas.
- **`fetch`** y **`blob`** se usan para manejar datos binarios como imágenes.
- **`Promise.race`** te permite manejar múltiples promesas y actuar cuando la primera se resuelve.

1. Promesas

Una **promesa** (o `Promise`) es un objeto que representa la finalización (o el fallo) eventual de una operación asincrónica. En otras palabras, es una forma de manejar operaciones que no se completan inmediatamente, como cuando haces un `fetch` para obtener datos desde una API.

- Una promesa puede estar en tres estados:
  - **Pendiente (Pending):** La operación aún no ha terminado.
  - **Cumplida (Resolved/Fulfilled):** La operación se completó con éxito.
  - **Rechazada (Rejected):** Hubo un error en la operación.

Ejemplo de promesa básica:

```
let miPromesa = new Promise((resolve, reject) => {
  let exito = true; // Cambia esto para probar el rechazo

  if (exito) {
    resolve('Operación exitosa');
  } else {
    reject('Hubo un error');
  }
});

miPromesa
  .then((mensaje) => console.log(mensaje)) // Si la promesa se resuelve
  .catch((error) => console.log(error)); // Si la promesa se rechaza
```

---

2. `then` y `catch`

Cuando trabajas con promesas, usas `then` para manejar cuando una promesa **se resuelve** exitosamente y `catch` para manejar los **errores** cuando la promesa es rechazada.

- **`then`** se ejecuta cuando la promesa se resuelve correctamente.
- **`catch`** se ejecuta cuando la promesa es rechazada (cuando ocurre un error).

Ejemplo con `fetch`:

```
fetch('https://api.example.com/data')  // Realizas un fetch a una URL
  .then(response => response.json())  // Convierte la respuesta a JSON
si es exitosa
  .then(data => console.log(data))    // Aquí manejas los datos
recibidos
  .catch(error => console.error('Hubo un error:', error));  // Si
ocurre un error en cualquier parte
```

Explicación:

- `then(response => response.json())` convierte la respuesta de `fetch` a formato JSON.

- `then(data => console.log(data))` muestra los datos recibidos.
- Si hay un error en cualquier parte del proceso (como si la URL no existe), el `catch` lo captura.

---

3. Encadenamiento de Promesas

El **encadenamiento de promesas** te permite ejecutar varias promesas en secuencia, una después de la otra. Cada `then` retorna una nueva promesa, por lo que puedes continuar encadenando más promesas.

Ejemplo:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data); // Manejas el primer set de datos
    return fetch('https://api.example.com/another-data');  // Retornas
una nueva promesa
  })
  .then(response2 => response2.json())  // Manejas la segunda promesa
  .then(data2 => console.log(data2))    // Manejas los datos del
segundo fetch
  .catch(error => console.error('Hubo un error:', error));
```

Explicación:

- El primer `fetch` obtiene datos y, en su `then`, retornamos una nueva promesa (otro `fetch`).
- Cada `then` se ejecuta de forma secuencial, esperando que cada promesa anterior se resuelva antes de pasar a la siguiente.
- Si alguna promesa falla, el `catch` captura el error.

---

4. `async/await`

`async/await` es una forma más moderna y más legible de trabajar con promesas en JavaScript. En lugar de usar `.then` y `.catch`, puedes escribir código más parecido al sincrónico, lo que mejora la claridad y legibilidad.

- **`async`** se usa para declarar una función que maneja promesas de forma asincrónica.
- **`await`** se usa dentro de una función `async` para esperar que una promesa se resuelva antes de continuar con el siguiente paso.

Ejemplo con `async/await`:

```
async function obtenerDatos() {
  try {
    let response = await fetch('https://api.example.com/data');
```

```
    let data = await response.json();
    console.log(data);  // Manejas los datos
  } catch (error) {
    console.error('Hubo un error:', error);  // Manejas los errores
  }
}

obtenerDatos();  // Llamas a la función
```

Explicación:

- **`async function`**: Declara que la función manejará operaciones asincrónicas.
- **`await`**: Pausa la ejecución hasta que la promesa de `fetch` se resuelva, y luego continúa con el siguiente paso (procesar los datos).
- **`try/catch`**: Captura los errores que puedan ocurrir, como en el caso de un `fetch` que no se pueda completar.

Resumen de conceptos:

- **Promesas**: Son objetos que representan la eventual resolución o fallo de una operación asincrónica.
- **`then` y `catch`**: Usados para manejar el resultado de una promesa. `then` maneja la resolución, y `catch` maneja los errores.
- **Encadenamiento de promesas**: Permite ejecutar varias promesas en secuencia, donde cada `then` maneja una parte del flujo y el siguiente `then` depende del resultado anterior.
- **`async/await`**: Una forma más moderna y limpia de manejar promesas, que permite escribir código asincrónico de manera más fácil y legible.

Convertimos el JSON a un objeto de JavaScript

const jsonData = [{blabla}, {blabla}, …];

const recetas = JSON.parse(jsonData);

Exportamos la variable si queremos usarla en otro archivo

export default recetas;

Para guardar un objeto JSON como una **cadena de texto** y luego convertirlo nuevamente en un **objeto JSON**, puedes usar dos funciones muy útiles de JavaScript:

1. **`JSON.stringify()`**: Convierte un objeto JavaScript en una cadena JSON.
2. **`JSON.parse()`**: Convierte una cadena JSON de vuelta a un objeto JavaScript.

Paso 1: Guardar el JSON como una cadena de texto

Para convertir un objeto JSON en una cadena de texto, usas `JSON.stringify()`. Esto es útil si quieres almacenar o transmitir el JSON como una cadena.

Ejemplo de cómo hacerlo:

javascript

Copiar

```
// Un objeto JSON
const objeto = {
  nombre: "Juan",
  edad: 25,
  ciudad: "Madrid"
};

// Convertir el objeto en una cadena JSON
const cadenaJSON = JSON.stringify(objeto);

console.log(cadenaJSON);
// Salida: '{"nombre":"Juan","edad":25,"ciudad":"Madrid"}'
```

Explicación:

- `JSON.stringify(objeto)` convierte el objeto JavaScript en una cadena de texto que representa ese objeto en formato JSON.

Paso 2: Convertir la cadena JSON de vuelta a un objeto JSON

Para convertir esa cadena de texto nuevamente a un objeto JSON (es decir, a un objeto JavaScript), usas `JSON.parse()`.

Ejemplo de cómo hacerlo:

javascript

Copiar

```
// La cadena JSON (la que obtuviste anteriormente)

const cadenaJSON = '{"nombre":"Juan","edad":25,"ciudad":"Madrid"}';


// Convertir la cadena JSON de vuelta a un objeto

const objetoParseado = JSON.parse(cadenaJSON);


console.log(objetoParseado);

// Salida: { nombre: 'Juan', edad: 25, ciudad: 'Madrid' }
```

Explicación:

- `JSON.parse(cadenaJSON)` convierte la cadena JSON de vuelta a un objeto JavaScript.

Resumen:

- **`JSON.stringify()`**: Convierte un objeto a una cadena JSON.
- **`JSON.parse()`**: Convierte una cadena JSON de vuelta a un objeto.

En JavaScript, puedes usar **`localStorage`** para almacenar datos de manera persistente en el navegador del usuario. Los datos guardados en `localStorage` permanecen allí incluso después de que el navegador se cierra y se vuelve a abrir. Sin embargo, **`localStorage`** solo almacena **cadenas de texto**, por lo que si deseas guardar objetos o arrays, primero deberás convertirlos en una cadena utilizando `JSON.stringify()`.

1. Guardar datos en `localStorage`

Para guardar datos en `localStorage`, usas el método `localStorage.setItem()`. Este método requiere dos parámetros:

- **La clave** (un nombre que se utilizará para identificar los datos).
- **El valor** (los datos que deseas almacenar como cadena de texto).

Si necesitas almacenar un objeto o un array, debes convertirlo primero a una cadena con `JSON.stringify()`.

Ejemplo de guardar un objeto en `localStorage`:

javascript

Copiar

```
const usuario = {
  nombre: "Juan",
  edad: 25,
  ciudad: "Madrid"
};

// Convertir el objeto a una cadena JSON y guardarlo
localStorage.setItem('usuario', JSON.stringify(usuario));

console.log("Datos guardados en localStorage");
```

2. Extraer datos de `localStorage`

Para obtener datos de `localStorage`, usas el método `localStorage.getItem()`, pasando la clave del dato que deseas recuperar. El valor que obtienes será una cadena de texto, por lo que si has almacenado un objeto, debes convertirlo nuevamente a un objeto con `JSON.parse()`.

Ejemplo de extraer un objeto de `localStorage`:

```
// Obtener la cadena JSON desde localStorage
const usuarioGuardado = localStorage.getItem('usuario');

// Convertir la cadena JSON de nuevo a un objeto
if (usuarioGuardado) {
  const usuarioObjeto = JSON.parse(usuarioGuardado);
  console.log(usuarioObjeto);
  // Salida: { nombre: 'Juan', edad: 25, ciudad: 'Madrid' }
} else {
  console.log("No se encontraron datos.");
}
```

3. Eliminar datos de `localStorage`

Si deseas eliminar un dato específico de `localStorage`, usas `localStorage.removeItem()`, pasando la clave del dato que deseas eliminar.

Ejemplo de eliminar datos:

```
localStorage.removeItem('usuario');
console.log("Datos eliminados de localStorage");
```

4. Limpiar todo el `localStorage`

Si deseas eliminar **todos** los datos almacenados en `localStorage`, usas `localStorage.clear()`.

Ejemplo de limpiar todo `localStorage`:

```
localStorage.clear();
console.log("Todos los datos de localStorage han sido eliminados");
```

Resumen:

- **Guardar datos**: `localStorage.setItem(clave, valor)`
    - Si el valor es un objeto, usa `JSON.stringify(valor)`.
- **Obtener datos**: `localStorage.getItem(clave)`
    - Si el valor es una cadena JSON, usa `JSON.parse(valor)` para convertirlo nuevamente a un objeto.
- **Eliminar un dato específico**: `localStorage.removeItem(clave)`
- **Eliminar todo el `localStorage`**: `localStorage.clear()`