Async y await en JavaScript

En JavaScript, async y await son ***una sintaxis para escribir código asíncrono*** que es más sencillo **más fácil de leer y mantener** que las alternativas anteriores *(como los callbacks y las promesas)*

Introducido en ECMAScript 2017, async/await nos permite **escribir código secuencialmente**, en lugar de anidar callbacks o encadenar promesas.

Internamente, **async/await se basa en promesas**. De hecho `then` / `catch` y `async` / `await` son funcionalmente equivalentes.

Pero async/await proporciona **una sintaxis más cómoda de usar**, que anidar promesas *(es syntacic sugar para simplificarnos la vida)*.

Otra ventaja de async/await es que **maneja los errores de manera forma más sencilla**. Con async/await, puede usar la estructura `try/catch` *(que es más familiar y fácil de entender para los desarrolladores)*.

Sintaxis de `async/await`

La sintaxis async/await se basa en dos palabras clave,

- **async** se coloca antes de la función para indicar que ***contiene código asincrónico***.
- **await** se coloca antes de cualquier operación que devuelva una promesa para indicar que ***el código debe esperar*** a que se resuelva la promesa antes de continuar.

Async

La palabra clave `async` se usa para declarar una función asincrónica. Una función marcada con `async` siempre devuelve una promesa.

- Si la función retorna un valor, la promesa se resuelve con ese valor.
- Si la función lanza una excepción, la promesa se rechaza con esa excepción.

```
async function miFuncion() {
  return 'Hola, mundo';
}

miFuncion().then(console.log); // 'Hola, mundo'
```

En el ejemplo anterior, `miFuncion` es una función asincrónica que devuelve una promesa que se resuelve con el valor `'Hola, mundo'`.

Await

La palabra clave `await` se utiliza dentro de una función `async` para esperar la resolución de una promesa.

`await` pausa la ejecución de la función `async` hasta que la promesa se resuelve o se rechaza.

```
async function miFuncion() {
  let valor = await Promise.resolve('Hola, mundo');
  console.log(valor); // 'Hola, mundo'
}

miFuncion();
```

En este ejemplo, `await` se usa para esperar a que la promesa se resuelva, y luego se imprime el valor resultante.

Ejemplo básico

Vamos a verlo con un ejemplo sencillo,

```
async function obtenerDatos() {
  //funcion que simula devolver una promesa
  const respuesta = await FuncionQueDevuelvePromesa();

  console.log(respuesta);
}
```

En este ejemplo,

- La función `obtenerDatos` es asincrónica
- La `FuncionQueDevuelvePromesa` es una función que simula devolver una promesa *(es la respuesta de una solicitud de red)*.
- Usamos la palabra clave `await` para esperar a que se resuelva una promesa antes de continuar con la siguiente línea de código.
- Después de que se resuelve la promesa, podemos hacer lo que queramos con `respuesta`.

Esto sería equivalente a este código, sin usar async y await

```
function obtenerDatos() {
  // Llamar a la función que devuelve una promesa
  FuncionQueDevuelvePromesa()
    .then((respuesta) => {
      // Manejar la respuesta cuando la promesa se resuelve
```

```
      console.log(respuesta);
    })
    .catch((error) => {
      // Manejar errores si la promesa se rechaza
      console.error(error);
    });
}
```

Que como vemos es bastante más peñazo 😈

Promesas internas

Una función `async` siempre devuelve una promesa. Esta promesa se resuelve con el valor retornado de la función.

Si se lanza una excepción dentro de la función `async`, la promesa se rechaza con esa excepción.

```
async function funcionExito() {
  return 'Éxito';
}

async function funcionError() {
  throw new Error('Algo salió mal');
}

funcionExito().then(console.log); // 'Éxito'
funcionError().catch(console.error); // 'Error: Algo salió mal'
```

En `funcionExito`,

- El valor `'Éxito'` se envuelve en una promesa que se resuelve.
- En `funcionError`, la excepción lanzada se envuelve en una promesa que se rechaza.

Manejando errores

Para manejar errores en funciones `async`, puedes usar `try` / `catch` en lugar de `.catch()` como lo harías con promesas.

```
async function obtenerDatos() {
  try {
    let respuesta = await fetch('https://api.example.com/data');
    if (!respuesta.ok) {
      throw new Error('Error en la respuesta de la red');
    }
    let datos = await respuesta.json();
    console.log(datos);
```

```
  } catch (error) {
    console.error('Error:', error);
  }
}

obtenerDatos();
```

En este ejemplo, si ocurre un error en cualquier punto dentro del bloque `try`, el flujo de control pasa al bloque `catch`, donde se maneja el error.

Encadenamiento de operaciones asíncronas

Puedes encadenar múltiples operaciones asíncronas dentro de una función `async`.

```
async function procesarDatos() {
  let datos = await obtenerDatosDesdeAPI();
  let procesados = await procesarDatos(datos);
  console.log(procesados);
}

async function obtenerDatosDesdeAPI() {
  let respuesta = await fetch('https://api.example.com/data');
  return await respuesta.json();
}

async function procesarDatos(datos) {
  // Procesamiento de datos
  return datos.map(dato => dato * 2);
}

procesarDatos();
```

En este ejemplo, `procesarDatos` utiliza `await` para esperar los resultados de `obtenerDatosDesdeAPI` y `procesarDatos` en una secuencia de operaciones.

Funciones concurrentes

Si necesitas ejecutar múltiples operaciones asíncronas en paralelo, puedes usar `Promise.all` con `await`.

```
async function realizarOperaciones() {
  let promesa1 = fetch('https://api.example.com/data1');
  let promesa2 = fetch('https://api.example.com/data2');

  let respuestas = await Promise.all([promesa1, promesa2]);
  let datos = await Promise.all(respuestas.map(res => res.json()));
```

```
  console.log(datos);
}

realizarOperaciones();
```

En este caso,

- Las dos solicitudes `fetch` se ejecutan en paralelo.
- `Promise.all` espera a que ambas se completen antes de continuar.