24. Promesas

Las **promesas** son una herramienta esencial en JavaScript para manejar operaciones **asincrónicas** de manera más estructurada y legible. Permiten evitar problemas como el "callback hell" y facilitan la escritura de código limpio y mantenible al trabajar con tareas que tardan tiempo en completarse, como peticiones HTTP o lecturas de archivos.

¿Qué es una Promesa?

Una **promesa** es un objeto que representa la eventual finalización (o falla) de una operación asincrónica y su resultado. Tiene tres estados posibles:

1. **Pendiente (`pending`)**: Estado inicial; la promesa aún no se ha cumplido ni rechazado.
2. **Cumplida (`fulfilled`)**: La operación se completó exitosamente.
3. **Rechazada (`rejected`)**: La operación falló.

Creación de una Promesa

Sintaxis:

```
const miPromesa = new Promise((resolve, reject) => {
  // Código asincrónico
  if (condicion) {
    resolve(valor); // Operación exitosa
  } else {
    reject(error); // Operación fallida
  }
});
```

Ejemplo:

```
const descargarDatos = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { id: 1, nombre: "Ana" };
      const error = false;

      if (!error) {
        resolve(datos); // Resuelve la promesa
      } else {
        reject("Error al descargar datos"); // Rechaza la promesa
      }
    }, 2000);
  });
};
```

Manejo de Promesas

Para manejar una promesa, usamos `.then()` para procesar el resultado exitoso y `.catch()` para manejar errores.

Ejemplo:

```
descargarDatos()
  .then((datos) => {
    console.log("Datos descargados:", datos);
  })
  .catch((error) => {
    console.error("Ocurrió un error:", error);
  });
```

Salida (después de 2 segundos):

```
Datos descargados: { id: 1, nombre: "Ana" }
```

Si forzamos al código de la promesa con `const error = true` nos devolverá un error en la consola.

Encadenamiento de Promesas

Un beneficio clave de las promesas es que puedes encadenar múltiples operaciones asincrónicas sin caer en el "callback hell".

Ejemplo:

```
function procesarDatos(datos) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      datos.procesado = true;
      resolve(datos);
    }, 1000);
  });
}

descargarDatos()
  .then((datos) => {
    console.log("Datos originales:", datos);
    return procesarDatos(datos); // Devuelve otra promesa
  })
  .then((datosProcesados) => {
    console.log("Datos procesados:", datosProcesados);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Salida:

```
Datos originales: { id: 1, nombre: "Ana" }
Datos procesados: { id: 1, nombre: "Ana", procesado: true }
```

Métodos Estáticos de `Promise`

JavaScript proporciona métodos estáticos para trabajar con promesas.

a. `Promise.all(iterable)`

Espera a que **todas** las promesas del iterable se resuelvan. Si alguna falla, rechaza inmediatamente.

```
const promesa1 = Promise.resolve(1);
const promesa2 = new Promise((resolve) => setTimeout(() => resolve(2), 2000));
const promesa3 = new Promise((resolve) => setTimeout(() => resolve(3), 1000));

Promise.all([promesa1, promesa2, promesa3])
  .then((resultados) => {
    console.log(resultados); // [1, 2, 3]
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Salida:

```
Promise {<pending>}
(3) [1, 2, 3]
```

b. `Promise.race(iterable)`

Retorna la primera promesa que se resuelva o rechace.

```
const promesaRapida = new Promise((resolve) => setTimeout(() => resolve("Primero"), 500));
const promesaLenta = new Promise((resolve) => setTimeout(() => resolve("Segundo"), 1000));

Promise.race([promesaLenta, promesaRapida])
  .then((resultado) => {
    console.log(resultado); // "Primero"
  });
```

c. `Promise.allSettled(iterable)`

Espera a que **todas** las promesas terminen (ya sea cumplidas o rechazadas).

```
const promesa1 = Promise.resolve("Éxito");
const promesa2 = Promise.reject("Fallo");
```

```
const promesa3 = Promise.resolve("Otro éxito");

Promise.allSettled([promesa1, promesa2, promesa3])
  .then((resultados) => {
    console.log(resultados);
  });
```

Salida:

```
(3) [{...}, {...}, {...}]
0: {status: 'fulfilled', value: 'Éxito'}
1: {status: 'rejected', reason: 'Fallo'}
2: {status: 'fulfilled', value: 'Otro éxito'}
length: 3
[[Prototype]]: Array(0)
```

Promesas vs Callbacks

Característica	Callbacks	Promesas
Legibilidad	Puede volverse difícil (callback hell)	Código más limpio y estructurado
Manejo de Errores	Manual	Centralizado con `.catch()`
Encadenamiento	Complejo	Fácil con `.then()`

Buenas Prácticas

1. **Siempre maneja los errores**:
   Usa
   `.catch()` para capturar errores y evitar interrupciones en tu aplicación.
2. **Evita anidar promesas innecesariamente**:
   Usa el encadenamiento para mantener el código plano.
3. **Usa `async/await` si es posible**:
   Las promesas pueden simplificarse aún más con
   `async/await`.