

Promesas y Async/Await



Elaborado por: César Guerra

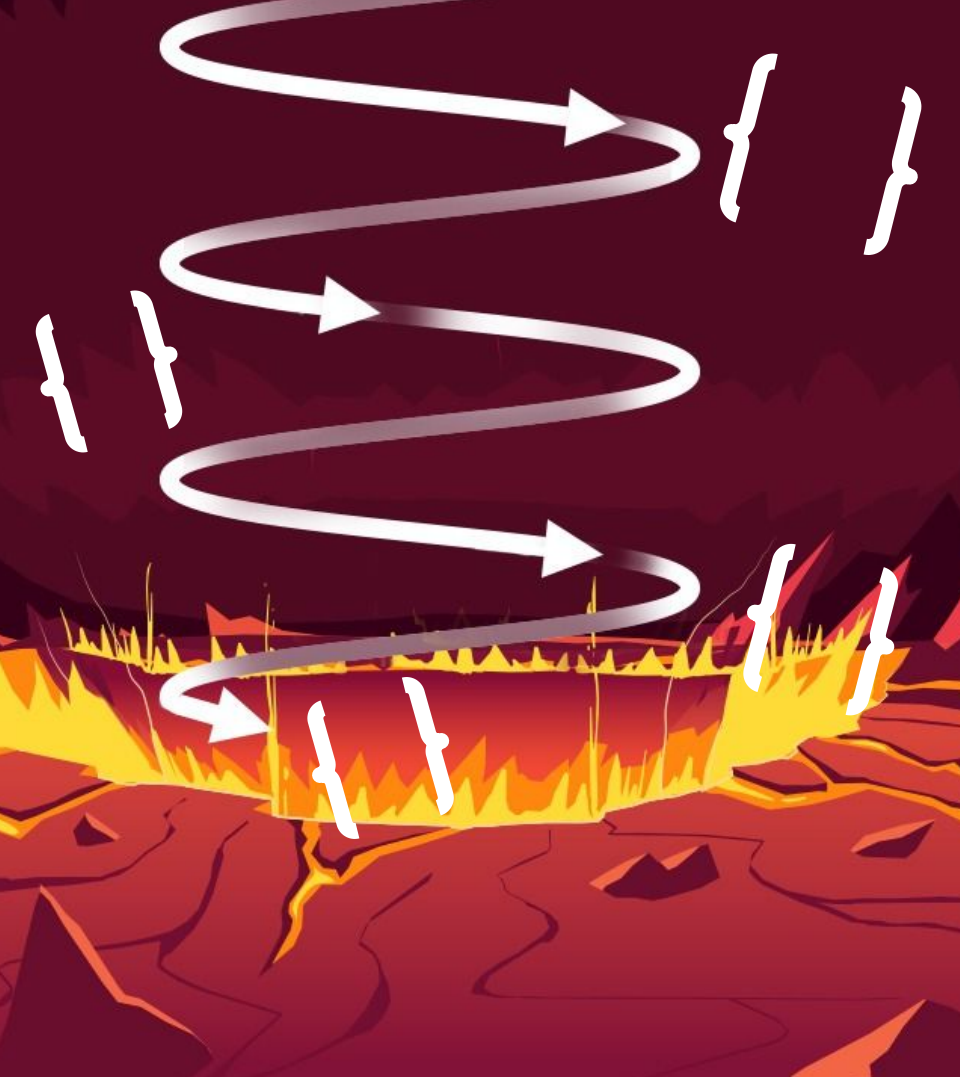
www.cesarquerra.mx

dev

Promesas

DEV.F
DESARROLLAMOS(PERSONAS);

dev




La Problemática

Uno de los desafíos que encontramos al comenzar a desarrollar código asíncrono es que normalmente ocupamos hacer varias operaciones en una sola instrucción que involucran usar el resultado de una invocación asíncrona previa para continuar con nuestro código, y así sucesivamente...

Callback Hell


La causa del **callback hell** es cuando las personas intentan escribir JavaScript de forma tal que la ejecución se realiza **visualmente** de arriba a abajo. Esto porque aprendimos a programar de forma síncrona donde sabemos que la línea 2 se ejecuta después de la línea 1.

```
pan.pourWater(function() {  
  range.bringToBoil(function() {  
    range.lowerHeat(function() {  
      pan.addRice(function() {  
        setTimeout(function() {  
          range.turnOff();  
          serve();  
        }, 15 * 60 * 1000);  
      });  
    });  
  });  
});
```



pyramid of doom

```
1 function hell(win) {  
2   // for listener purpose  
3   return function() {  
4     loadLink(win, REMOTE_SRC+'assets/css/style.css', function() {  
5       loadLink(win, REMOTE_SRC+'lib/async.js', function() {  
6         loadLink(win, REMOTE_SRC+'lib/easyXDM.js', function() {  
7           loadLink(win, REMOTE_SRC+'lib/json2.js', function() {  
8             loadLink(win, REMOTE_SRC+'lib/underscore.min.js', function() {  
9               loadLink(win, REMOTE_SRC+'lib/backbone.min.js', function() {  
10                loadLink(win, REMOTE_SRC+'dev/base_dev.js', function() {  
11                 loadLink(win, REMOTE_SRC+'assets/js/deps.js', function() {  
12                  loadLink(win, REMOTE_SRC+'src/' + win.loader_path + '/loader.js', function() {  
13                    async.eachSeries(SRIPTS, function(src, callback) {  
14                      loadScript(win, BASE_URL+src, callback);  
15                    });  
16                  });  
17                });  
18              });  
19            });  
20          });  
21        });  
22      });  
23    });  
24  });  
25  };  
26 }
```



Las Promesas al Rescate

DEV.F
DESARROLLAMOS(PERSONAS);

dev



{ api }

Promesas

Las promesas es una forma de garantizar una respuesta de un **callback** (qué recordemos es asíncrono), ya que estas no te pueden dar una respuesta al momento, si no en un futuro.

Dicha respuesta de la promesa **puede ser cumplida satisfactoriamente o rechazada** por algún motivo.



JS

En otras palabras..

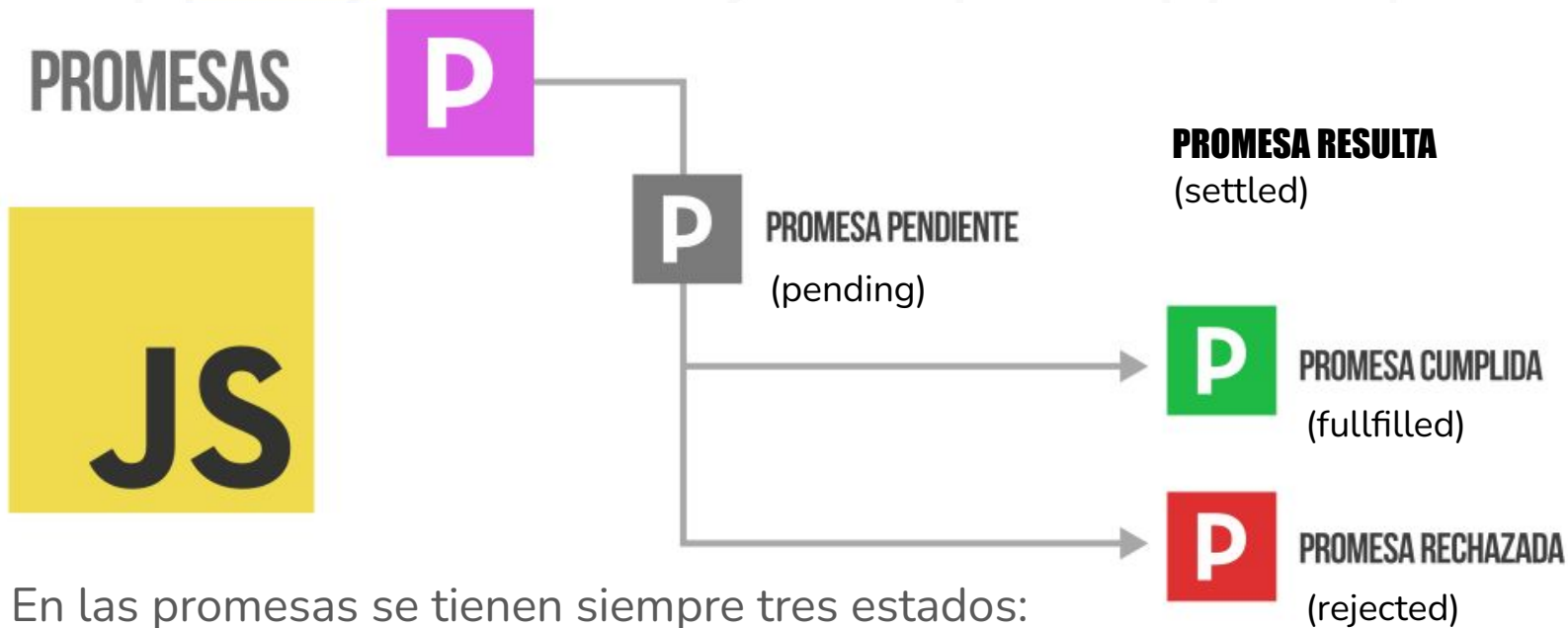


Estructura de una Promesa en JS

DEV.F
DESARROLLAMOS(PERSONAS);

dev

Flujo de Estados de una Promesa



En las promesas se tienen siempre tres estados:

- **Pendiente (Pending):** Estado inicial de la promesa.
- **Fullfilled (Resuelta):** Todo se ejecutó correctamente.
- **Rejected (Rechazada):** Hubo un problema.

Sintaxis de la DECLARACIÓN de una Promesa

El primer paso para usar una promesa es declararla. Utilizamos un constructor `Promise` y 2 parámetros.



```
//Declaración de una Promesa
const Promesa = new Promise((resolve, reject) => {
  //Código Asíncrono a Resolver
  // Ejemplo: Llamadas a APIs, Bases de Datos,
  // Leer Archivos, etc.
  resolve(cosaQueDevuelvoSiSeEjecutaBien);
  reject(cosaQueDevuelvoSiSeEjecutaMal);
})
```

- **resolve:** Se ejecuta cuando el objetivo de la promesa se efectuó de manera correcta
- **reject:** Se ejecuta cuando el objetivo de la promesa ocasionó un error o no se llegó a cumplir.



```
//Declaración de una Promesa
const Promesa = new Promise((resolve, reject) => {
  //Código Asíncrono a Resolver
  // Ejemplo: Llamadas a APIs, Bases de Datos,
  // Leer Archivos, etc.
  resolve(cosaQueDevuelvoSiSeEjecutaBien);
  reject(cosaQueDevuelvoSiSeEjecutaMal);
})

//Ejecución de la promesa
Promesa
  .then( (resultado) => {
    //logica con el resultado
  }).catch (error) => {
    //manejo el error, ejemplo:
    console.log(error);
  }
```

EJECUTANDO una Promesa

Podemos ejecutar una promesa por medio de los métodos **then** y **catch**.

Funciona muy similar a la lógica de **if/else**.

.then: es el código que se ejecuta cuando se resuelve la promesa de forma satisfactoria (fulfilled).

.catch: si la promesa no se resuelve (rejected) ejecutara esta parte del código.

Ejemplo

```
const promise = new Promise((resolve, reject) => {
  const number = Math.floor(Math.random() * 10);

  setTimeout(
    () => number > 5
      ? resolve(number)
      : reject(new Error('Menor a 5')),
    1000
  );
});

promise
  .then(number => console.log(number))
  .catch(error => console.error(error));
```

Proceso de Promesas



Kayo Underwood

I Promise to make
a cake for your
birthday

2
weeks

Based on real-life scenario

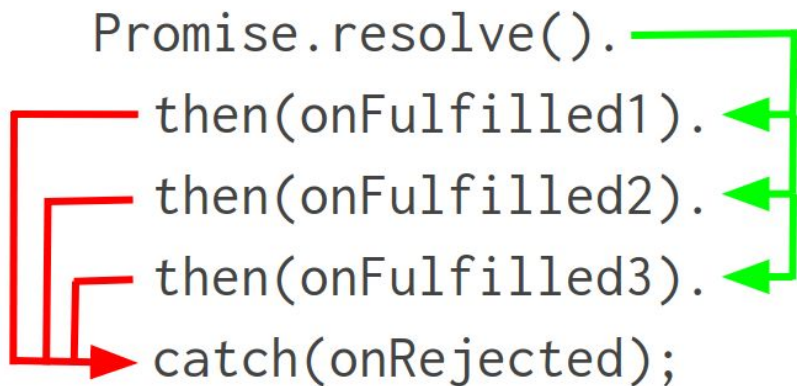
Made by Thu Nghiem - Founder at DevChallenges.io

Encadenamiento de Promesas

DEV.F
DESARROLLAMOS(PERSONAS);

dev

```
Promise.resolve().  
  then(onFulfilled1).  
  then(onFulfilled2).  
  then(onFulfilled3).  
  catch(onRejected);
```



Encadement de Promesas

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde **cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo.**

Logramos esto creando una cadena de objetos promises.

Ejemplo de cómo luce un encadementamiento



```
hazAlgo().then(function(resultado) {  
  return hazAlgoMas(resultado);  
})  
.then(function(nuevoResultado) {  
  return hazLaTerceraCosa(nuevoResultado);  
})  
.then(function(resultadoFinal) {  
  console.log('Obtenido el resultado final: ' + resultadoFinal);  
})  
.catch(falloCallback);
```

Resumen de Promesas

- Es usado para interacciones asíncronas
- Se compone de dos aspectos:
 - **Resolve:** Se ejecuta cuando el objetivo de la promesa se efectuó de manera correcta
 - **Reject:** Se ejecuta cuando el objetivo de la promesa ocasionó un error o no se llegó a cumplir.
- Se utiliza la palabra reservada ***“Promise”***
- En las promesas se tienen siempre tres estados:
 - **Pendiente:** Estado inicial de la promesa
 - **Resuelta:** Todo se ejecutó correctamente
 - **Rechazada:** Hubo un problema

Async / Await

DEV.F
DESARROLLAMOS(PERSONAS);

dev

Async () => { Await }

```
const getUserFollowersByEmail = async (email) => {  
  try {  
    const { idUser } = await fetchUser(email);  
    const followers = await fetchUserFollowers(idUser);  
    return followers;  
  } catch (error) {  
    throw new Error(error)  
  }  
}
```

Async/Await

Introducido en ES2017, **async/await** es una capa de "azúcar sintáctico" sobre las promesas que **nos permite escribir código asíncrono como si fuera síncrono**.

Esta sintaxis hace que el código sea aún más fácil de leer y entender, eliminando la necesidad de encadenar múltiples `.then()`. **Es la forma moderna y preferida de trabajar con operaciones asíncronas en JavaScript.**



```
// 1. Simulamos una función que devuelve una promesa
function obtenerUsuario() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ nombre: 'César Guerra', id: 101 });
    }, 2000);
  });
}

// 2. Usamos async/await para consumir la promesa
async function mostrarUsuario() {
  console.log('Buscando usuario...');

  // 'await' pausa la función aquí hasta que la promesa se cumpla
  const usuario = await obtenerUsuario();

  // Una vez cumplida, el código continúa
  console.log(`Usuario encontrado: ${usuario.nombre}`);
}

// Llamamos a la función asíncrona
mostrarUsuario();

// www.cesarguerra.mx - César Guerra
```

La Magia de Async/Await

La palabra clave **async** se coloca antes de la declaración de una función para indicar que esta **devolverá implícitamente una promesa**.

Dentro de una función **async**, podemos usar la palabra clave **await**.

await pausa la ejecución de la función, espera a que una promesa se resuelva (se cumpla o se rechace) y luego reanuda la **ejecución**. Si la promesa se cumple, **await** devuelve el valor. Si se rechaza, lanza una excepción.



```
function obtenerDatosFallidos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('Error 404: No se encontraron los datos.');
```



```
    }, 2000);
  });
}

// Usamos try...catch para manejar el error
async function procesarDatos() {
  console.log('Intentando obtener los datos...');
  try {
    // Intentamos obtener los datos. 'await' esperará el resultado
    const datos = await obtenerDatosFallidos();

    // Esta línea nunca se ejecutará porque la promesa se rechaza
    console.log('Datos recibidos:', datos);

  } catch (error) {
    // El error se disparará
    console.error('Ocurrió un error:', error);
  }
}

procesarDatos();

// www.cesarguerra.mx - César Guerra
```

Async/Await

Atrapando Errores de Forma Sincrónica

Una de las mayores ventajas de **async/await** es que nos permite manejar errores utilizando los bloques **try...catch**, que ya conocemos del código síncrono.

Simplemente envolvemos la llamada **await** en un bloque **try**. Si la promesa es rechazada, el control salta al bloque **catch**, donde podemos manejar el error de una manera limpia y estructurada, evitando los bloques **.catch()** separados.

Práctica de la Clase

DEV.F
DESARROLLAMOS(PERSONAS);

dev

Sistema de Reservas para un Restaurante

Construir un **sistema de reservas** utilizando **promesas** y **async/await**, con manejo de errores adecuado.

Problema: Sistema de Reservas para un Restaurante

Imagina que tienes un restaurante y deseas permitir a los clientes hacer reservas en línea. Para ello, el sistema debe hacer las siguientes acciones:

1. Verificar si hay mesas disponibles para el día y la hora solicitados.
2. Si las mesas están disponibles, confirmar la reserva.
3. Si todo está bien, enviar un correo de confirmación.
4. Manejar adecuadamente los errores (si no hay mesas disponibles o si hay un fallo en el envío del correo).

Enlace Campus:

https://edu.devf.la/campus/program/module/desarrollo_avanzado_javascript/promises-async-await/project/promises-async-await-project

Código Inicial:

<https://gist.github.com/heladio-devf-mx/4a1f75b7f422723c2ed52ca446650a3d>