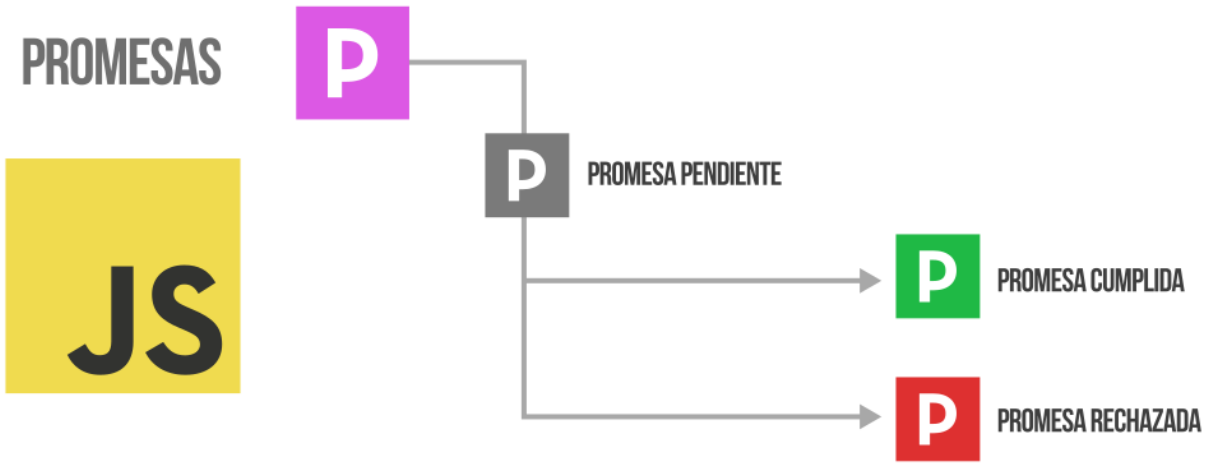
¿Qué son las promesas? - Javascript en español

Las **promesas** son un concepto para resolver el problema de asincronía de una forma mucho más elegante y práctica que, por ejemplo, utilizando funciones callbacks directamente.

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:



- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Con estas sencillas bases, podemos entender el funcionamiento de una promesa en Javascript. Antes de empezar, también debemos tener claro que existen dos partes importantes de las promesas: **como consumirlas** (*utilizar promesas*) y **como crearlas** (*preparar una función para que use promesas y se puedan consumir*).

Promesas en Javascript

Las **promesas** en Javascript se representan a través de un , y cada **promesa** estará en un estado concreto: `pendiente`, `aceptada` o `rechazada`. Además, cada **promesa** tiene los siguientes métodos, que podremos utilizar para utilizarla:

Métodos	Descripción
`.then(resolve)`	Ejecuta la función callback `resolve` cuando la promesa se cumple.
`.catch(reject)`	Ejecuta la función callback `reject` cuando la promesa se rechaza.
`.then(resolve,reject)`	Método equivalente a las dos anteriores en el mismo `.then()`.
`.finally(end)`	Ejecuta la función callback `end` tanto si se cumple como si se rechaza.

Más adelante veremos, que a diferencia del apartado anterior donde se utilizaban solamente funciones callback, en este enfoque se tiende a **no anidar promesas**, evitando así el famoso **Callback Hell**, y haciendo el código mucho más legible.

Consumir una promesa

La forma general de consumir una promesa es utilizando el `.then()` con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla:

```
fetch("/robots.txt").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */
});
```

Lo que vemos en el ejemplo anterior es el uso de la función fetch(), la cuál devuelve una promesa que se cumple cuando obtiene respuesta de la petición realizada. De esta forma, estaríamos preparando (*de una forma legible*) la forma de actuar de nuestro código a la respuesta de la petición realizada, todo ello de forma asíncrona.

Recuerda que podemos hacer uso del método `.catch()` para actuar cuando se rechaza una promesa:

```
fetch("/robots.txt")
  .then(function(response) {
    /* Código a realizar cuando se cumpla la promesa */
  })
  .catch(function(error) {
    /* Código a realizar cuando se rechaza la promesa */
  });
```

Observa como hemos indentado los métodos `.then()` y `.catch()`, ya que se suele hacer así para que sea mucho más legible para el. Además, se pueden encadenar varios `.then()` si se siguen generando promesas y se devuelven con un `return`:

```
fetch("/robots.txt")
  .then(response => {
    return response.text(); // Devuelve una promesa
  })
  .then(data => {
```

```
    console.log(data);
  })
  .catch(error => { /* Código a realizar cuando se rechaza la promesa */ });
```

> No olvides indicar el `return` para poder encadenar las siguientes promesas con `.then()`. Tras un `.catch()` también es posible encadenar `.then()` para continuar procesando promesas.

De hecho, usando **arrow functions** se puede mejorar aún más la legibilidad de este código, recordando que cuando sólo tenemos una sentencia en el cuerpo de la **arrow function** hay un `return` implícito:

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => console.log(data))
  .finally(() => console.log("Terminado."))
  .catch(error => console.error(data));
```

Observese además que hemos añadido el método `.finally()` para añadir una función callback que se ejecutará **tanto si la promesa se cumple o se rechaza**, lo que nos ahorrará tener que repetir la función en el `.then()` como en el `.catch()`.

> En todo este apartado hemos visto como utilizar o consumir una promesa haciendo uso de `.then()`, que es lo que en la mayoría de los casos necesitaremos. Sin embargo, vamos a ver en el siguiente apartado como crear o implementar las promesas para su posterior consumo.

Asincronía con promesas

Vamos a implementar el ejercicio base que hemos comentado en el primer capítulo de este tema utilizando **promesas**. Observa que lo primero que haremos es crear un nuevo objeto que «envuelve» toda la función de la tarea `doTask()`.

Al `new Promise()` se le pasa por parámetro una función con dos callbacks, el primero `resolve` el que utilizaremos cuando se cumpla la promesa, y el segundo `reject` cuando se rechace:

```
/* Implementación con promesas */
const doTask = (iterations) => new Promise((resolve, reject) => {
  const numbers = [];
  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      reject({
        error: true,
        message: "Se ha sacado un 6"
      });
    }
  }
  resolve({
    error: false,
    value: numbers
  });
});
```

Como ves, se trata de una implementación muy similar a los callbacks que vimos en el apartado anterior, pero utilizan una nativa para poder luego consumirla cómodamente:

```
doTask(10)
  .then(result => console.log("Tiradas correctas: ", result.value))
  .catch(err => console.error("Ha ocurrido algo: ", err.message));
```

Imagina el caso de que **cada lanzamiento del dado** (*la parte donde genera el número aleatorio*) fuera un proceso más costoso que **tardara un tiempo considerable**, quizás de esa forma se vea más clara la necesidad de una tarea asíncrona, controlada con promesas.

En el siguiente capítulo veremos como trabajar con múltiples promesas y hacer acciones compuestas con varias de ellas.

> Si el ejemplo anterior te resulta demasiado críptico por las funciones `resolve` y `reject`, es muy probable que echar un vistazo al tema de las funciones callback te aclare muchos detalles.