

**DWC**

**(Desarrollo Web en entorno cliente)**



**JavaScript**

**Tema 15**



**Promesas**

## Índice

1.- Promesas .....	1
2.- Crear una promesa .....	1
3.- Consumir promesas: then, catch, finally .....	4
3.1.- Then .....	5
3.2.- Catch.....	6
3.3.- Finally .....	7
4.- Promesas vs Callbacks .....	8
5.- Promesas encadenadas .....	9
6.- El método all.....	12
7.- Transformando XMLHttpRequest en promesas .....	15

## 1.- Promesas



Una promesa es un objeto que representa **la terminación o el fracaso** de una **operación asíncrona**. Dado que la mayoría de los programadores **consumen promesas ya creadas**, es importante conocer como consumirlas tanto o más que incluso saber cómo crearlas.  

Esencialmente, una promesa es un objeto devuelto al cuál se adjuntan funciones callback, en lugar de pasar callbacks a una función.

## 2.- Crear una promesa

La sintaxis del constructor para un objeto de promesa es:

```
let promise = new Promise(function(resolve, reject) {  
  // executor (the producing code)  
});
```

La función pasada a new Promise se llama ejecutor. Cuando new Promise se crea, el ejecutor se ejecuta automáticamente. Contiene el código que debería producir el resultado. Lo que no sabemos es cuando finalizara su ejecución y estarán disponibles los resultados".

Sus argumentos **resolve** y **reject** son callbacks provistos por el propio JavaScript. Nuestro código solo está dentro del ejecutor.

Cuando el ejecutor obtiene el resultado, ya sea pronto o tarde, no importa, debe llamar a una de estas devoluciones de llamada:

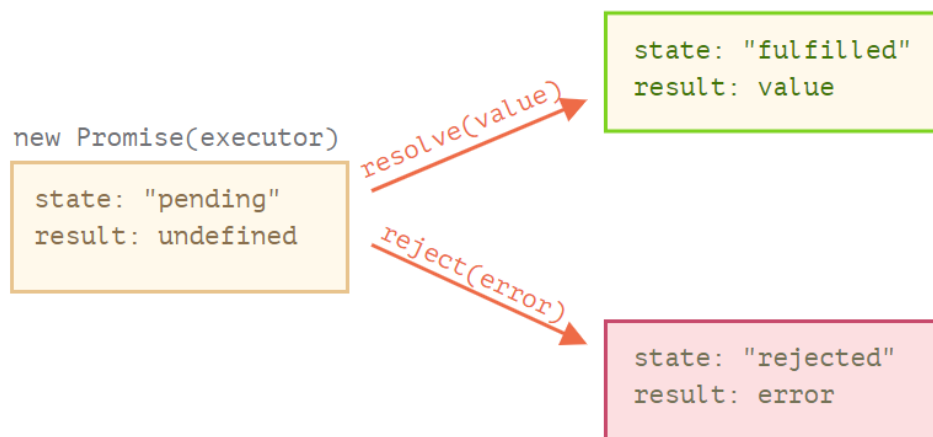
- **resolve(value)**- si el trabajo finalizó con éxito, con resultado value.
- **reject(error)**- Si ocurrió un error, errores el objeto de error.

Para resumir: el ejecutor se ejecuta automáticamente e intenta realizar un trabajo. Cuando termina con el intento, llama a **resolve si tuvo éxito** o llama a **reject si hubo un error**.

El objeto promesa devuelto por el new constructor Promise tiene estas propiedades internas:

- **state**- inicialmente *"pending"* (pendiente), luego cambia a *"fulfilled"* (cumplida) cuando se llama a resolve o *"rejected"* (rechazada) cuando se llama a reject.
- **result**- inicialmente undefined, luego cambia a value cuando se llama a resolve(value) o a error cuando se llama a reject(error).

El esquema es el siguiente:



Hasta ahora sólo hemos visto cómo se crea una promesa, nos falta por ver como se usaran los datos devueltos cuando la promesa se resuelva o como se gestionara el error

Un ejemplo de un constructor de promesas y una función ejecutora simple con una ejecución asíncrona con `setTimeout`

```
let promise = new Promise(function(resolve, reject) {  
  // the function is executed automatically when the promise is constructed  
  // after 1 second signal that the job is done with the result "done"  
  setTimeout(() => resolve("done"), 1000);  
});
```

Podemos ver varias cosas en el código anterior:

- El ejecutor es llamado automáticamente e inmediatamente (por `new Promise`).

- El ejecutor recibe dos argumentos: `resolve` y `reject`. Estas funciones están predefinidas por el motor de JavaScript, por lo que no necesitamos crearlas. Solo debemos llamar a uno de ellas cuando sea necesario.
- Después de un segundo de "procesamiento" (es lo que simula el uso de la función `setTimeout`), el ejecutor llama `resolve` ("done") para producir el resultado (quiere decir que la promesa devuelve la cadena de texto 'done'. Esto cambia el estado del objeto `promise`:

```
new Promise(executor)
```

```
state: "pending"  
result: undefined
```

`resolve("done")`

```
state: "fulfilled"  
result: "done"
```

Ese es un ejemplo de **finalización con éxito** del trabajo a realizar en la función ejecutor, una "**promesa cumplida**".

Y ahora el mismo ejemplo del ejecutor rechazando la promesa con un error:

```
let promise = new Promise(function(resolve, reject) {  
  // after 1 second signal that the job is finished with an error  
  setTimeout(() => reject(new Error("Whoops!")), 1000);  
});
```

La llamada a `reject(...)` modifica el estado y el resultado de la promesa:

```
new Promise(executor)
```

```
state: "pending"  
result: undefined
```

`reject(error)`

```
state: "rejected"  
result: error
```

Para resumir, el ejecutor debe realizar un trabajo (generalmente algo que lleva tiempo) y luego llamar `resolve` o `reject` para cambiar el estado del objeto de promesa correspondiente.

Una promesa que se resuelve o se rechaza se denomina "liquidada", en oposición a una promesa inicialmente "pendiente".

**Solo puede haber un único resultado o un error.**

El ejecutor debe llamar solo a uno resolve o a un reject. Cualquier cambio de estado es final. Todas las llamadas adicionales de resolve y reject se ignoran:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("...")); // ignored
  setTimeout(() => resolve("...")); // ignored
});
```

La idea es que un trabajo realizado por el ejecutor puede tener solo un resultado o un error.

**Ejecución de promesas de forma inmediata.**

En la práctica, un ejecutor generalmente hace algo de forma asíncrona y llama resolve o reject después de un tiempo, pero no tiene porque ser así siempre. También podemos llamar a resolve o reject inmediatamente, así:

```
let promise = new Promise(function(resolve, reject) {
  // not taking our time to do the job
  resolve(123); // immediately give the result: 123
});
```

**Los valores state y result son internos**

Las propiedades state y result del objeto Promise son internas. No podemos acceder directamente a ellos. Podemos usar los métodos **.then**, **.catch** y **.finally** para eso. Los veremos a continuación.

**3.- Consumir promesas: then, catch, finally**

Un objeto Promise sirve como enlace entre el ejecutor (código que se ejecuta al crear la promesa) y las funciones consumidoras (código que gestionara el resultado de la promesa a través del resolve o del reject),

**Las funciones de consumo pueden registrarse (suscribirse)** utilizando los métodos **.then**, **.catch** y **.finally**.

### 3.1.- Then

Es el método más importante pues permite gestionar la finalización completa de la promesa.

La sintaxis es:

```
promise.then(  
  function(result) { /* handle a successful result */ },  
  function(error) { /* handle an error */ }  
);
```

- **El primer argumento de .then** es una función que se ejecuta cuando se resuelve la promesa y recibe el resultado.
- **El segundo argumento de .then** es una función que se ejecuta cuando se rechaza la promesa y recibe el error.

Un ejemplo de una promesa resuelta con éxito:

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => resolve("done!"), 1000);  
});  
  
// resolve runs the first function in .then  
promise.then(  
  result => alert(result), // shows "done!" after 1 second  
  error => alert(error) // doesn't run  
);
```

En este caso hemos creado una promesa que se resuelve de forma satisfactoria después de 1 segundo. La promesa devuelve como resultado la cadena de texto "done!".

Nos subscribimos a la promesa mediante then y cuando la promesa acabe (en este caso después de 1 segundo) se ejecutará el código correspondiente según la finalización de la promesa. En este caso como la promesa ha tenido éxito se ejecutara la primera función del then, con lo cual se mostrara un alert con el valor

devuelto por `resolve`, que es pasado como el parámetro `result` a la primera función.

En el caso de un `reject` (rechazo) en la promesa

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

**Si solo nos interesan las terminaciones con éxito**, entonces podemos proporcionar solo un argumento de función para `.then`:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

### 3.2.- Catch

Si estamos interesados sólo en los errores, entonces podemos utilizar `then` con `null` como primer argumento:

```
promise.then(null, errorHandlerFunction).
```

O podemos usar `.`

```
promise.catch(errorHandlerFunction)
```

La segunda forma es la más utilizada



```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(error=>alert(error)); // shows "Error: Whoops!" after 1 second
```

### 3.3.- Finally

El método `.finally(f)` es similar a `.then(f, f)` en el sentido de que `f` siempre se ejecuta cuando se cumple la promesa: ya sea resolver o rechazar.

Imaginad que construimos una promesa para traer datos desde el servidor, como será una operación asíncrona y no sabemos cuánto va a tardar, para que el usuario sepa que se está haciendo algo vamos a poner al inicio de la promesa un gif animado de un loader. Si la promesa nos devuelve los datos de forma correcta o si nos devuelve un error haremos lo que toque para cada caso, pero en los dos casos deberemos de quitar el gif animado del loader.

```
new Promise((resolve, reject) => {
  start loading indicator
  /* do something that takes time, and then call resolve/reject */
})
// runs when the promise is settled, doesn't matter successfully or not
.finally(() => stop loading indicator)
.then(result => show result, err => show error)
```

Hay que tener en cuenta:

- Un controlador `finally` no tiene argumentos. En `finally` no sabemos si la promesa es exitosa o no. Eso está bien, ya que nuestra tarea generalmente es realizar procedimientos de finalización "generales".
- Un controlador `finally` pasa resultados y errores al siguiente controlador.

Por ejemplo, aquí el resultado se pasa de `finally` a `then`:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- .then handles the result
```

Y aquí hay un error en la promesa, transmitido de finally a catch:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
  .catch(err => alert(err)); // <-- .catch handles the error object
```

Eso es porque finally no está destinado a procesar un resultado de una promesa, entonces lo pasa a los controladores destinados a esa tarea (then y catch).

## 4.- Promesas vs Callbacks

Las promesas aparecen como una nueva forma para gestionar las operaciones asíncronas y nos proporcionan una forma más sencilla de escribir el código. Podemos ver inmediatamente algunos beneficios sobre el patrón basado en callbacks.

Promesas	Callbacks
Las promesas nos permiten hacer las cosas en el orden natural. Primero ejecutamos la función para obtener algo y al acabar con .then escribimos qué hacer con el resultado.	Debemos saber qué hacer con el resultado <i>antes de que se hagan las operaciones para obtenerlo</i>
Podemos subscribirnos a una promesa con .then tantas veces como queramos.	Solo puede haber una devolución de llamada.

Las promesas nos dan un mejor flujo de código y flexibilidad.

## 5.- Promesas encadenadas

Supongamos que tenemos una secuencia de tareas asíncronas que se deben realizar una tras otra. Las promesas nos ofrecen una forma muy sencilla de poder realizarlo mediante el encadenamiento o anidamiento.

`then()` no es el final del camino. Puedes encadenar varios `then` para transformar valores o ejecutar acciones asincrónicas adicionales una tras otra.

```
new Promise(function(resolve, reject) {  
  
    setTimeout(() => resolve(1), 1000); // (*)  
  
}).then(function(result) { // (**)  
  
    alert(result); // 1  
    return result * 2;  
  
}).then(function(result) { // (***)  
  
    alert(result); // 2  
    return result * 2;  
  
}).then(function(result) {  
  
    alert(result); // 4  
    return result * 2;  
  
});
```



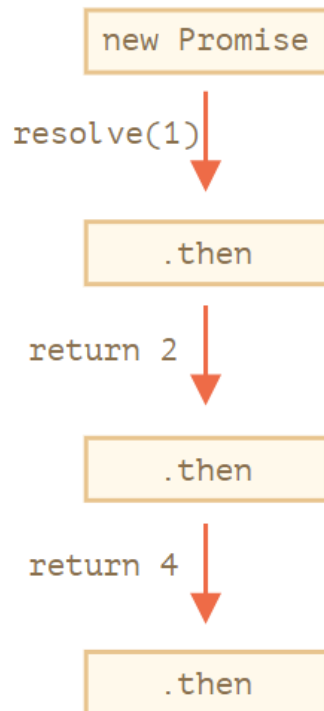
La idea es que el resultado se pase por la cadena de controladores `then`.

Aquí el flujo es:

1. La promesa inicial se resuelve con el valor 1 en un segundo (\*),
2. Entonces se llama al controlador `.then` y muestra el valor 1 en un `alert` y devuelve el valor 1 multiplicado por 2 (\*\*).

3. El valor que devuelve se pasa al siguiente controlador then que muestra el valor 2 en un alert y devuelve el valor 2 multiplicado por 2 (\*\*\*)
4. ...y así.

A medida que el resultado se pasa a lo largo de la cadena de controladores, podemos ver la secuencia de llamadas alert:1→2→4.



Todo funciona, porque una llamada a `promise.then` devuelve una promesa, para que podamos llamar a la siguiente `.then`.

Cuando un controlador devuelve un valor, se convierte en el resultado de esa promesa, por lo que se llama al siguiente `.then`.

Un error clásico es agregar muchos `.then` a una sola promesa. Esto no está encadenando promesas, sino ejecutando varias veces la misma.

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => resolve(1), 1000);  
});  
  
promise.then(function(result) {  
  alert(result); // 1
```

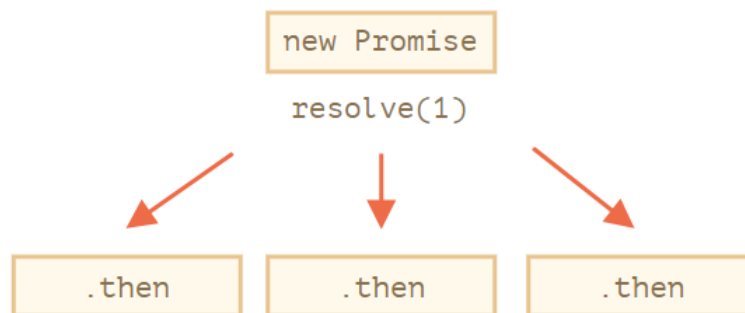
```
return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

Lo que hacemos aquí es usar varios controladores para una promesa. No se pasan el resultado el uno al otro; en su lugar lo procesan de forma independiente.

Aquí está la imagen, donde se pueden ver las diferencias con el otro caso:



## 6.- El método all

En algunas ocasiones lo que podemos querer hacer es ejecutar algo cuando se hayan realizado una serie de promesas a la vez sin que tengan que estar anidadas. Es decir lanzaremos varias promesas a la vez y deberemos esperar a que todas finalicen para ejecutar lo que deseemos.

Para este tipo de situaciones disponemos del método `all`.

El método `Promise.all(iterable)` devuelve una promesa que termina correctamente cuando todas las promesas en el argumento `iterable` han sido concluídas con éxito, o bien rechaza la petición con el motivo de error pasado por la primera promesa que es rechazada.

```
Promise.all(iterable);
```

- **Iterable** es cualquier elemento que se pueda iterar con las promesas a ejecutar.
- **El valor devuelto** es una promesa que se cumplirá cuando todas las promesas del argumento `iterable` hayan sido cumplidas, o bien se rechazará cuando alguna de ellas se rechace.

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "foo");
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

`Promise.all` se rechaza si uno de los elementos ha sido rechazado. Si tienes cuatro promesas que se resuelven después de un `timeout` y una de ellas falla inmediatamente, entonces `Promise.all` se rechaza inmediatamente.

## Creación de las promesas

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, "one");
});
var p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, "two");
});
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, "three");
});
var p4 = new Promise((resolve, reject) => {
  setTimeout(resolve, 4000, "four");
});
var p5 = new Promise((resolve, reject) => {
  reject("reject");
});
```

## Llamada al método all con error en una promesa

```
Promise.all([p1, p2, p3, p4, p5]).then(values => {
  console.log(values);
}, reason => {
  console.log(reason)      // From console: reject
});
```

Como la p5 falla inmediatamente, no se espera a que acaben la 4 anteriores, automáticamente termina con fallo.

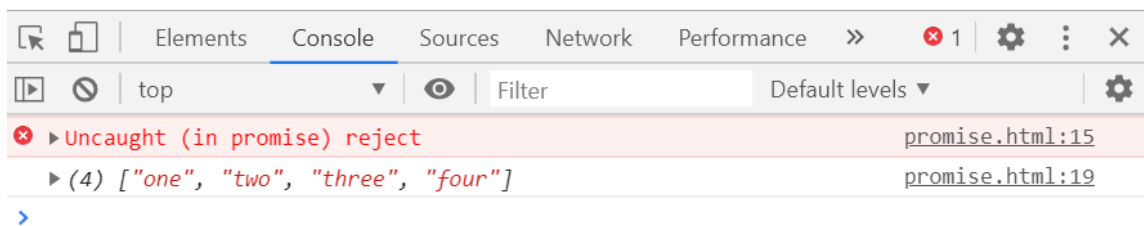
También podemos utilizar en los errores catch

```
Promise.all([p1, p2, p3, p4, p5]).then(values => {
  console.log(values);
}).catch(reason => {
  console.log(reason)
});
```

Si hubiéramos ejecutado el método `all` con las las promesas `p1`, `p2`, `p3` y `p4` que finalizan sin error hubiéramos obtenido los resultados de todas ellas.

```
Promise.all([p1, p2, p3, p4]).then(values => {  
  console.log(values);  
}, reason => {  
  console.log(reason)  
});
```

Resultado en la consola



El error que aparece es de ejecutar `p5` como promesa con error y no haber puesto el método `catch` para gestionarla.

El resultado del `all` como podemos ver es un array con los resultados de cada promesa.



## 7.- Transformando XMLHttpRequest en promesas

Como hemos visto para la gestión de operaciones asíncronas las promesas nos ofrecen muchas facilidades para escribir código complejo, por lo que muchas operaciones de este tipo se han “transformado” a promesas.

Vamos a ver cómo transformar una petición AJAX con XHR a una promesa.

El proceso de transformación más sencillo sería realizar la petición AJAX en una función que devolverá una promesa con los datos de la petición (resolve) o con el error en la petición (reject). Dentro de esta función es donde se implementaría el objeto XHR y se configuraría como ya sabemos hacerlo

```
function get(url) {  
  // Return a new promise.  
  return new Promise(function(resolve, reject) {  
    // Do the usual XHR stuff  
    var req = new XMLHttpRequest();  
    req.open('GET', url);  
  
    req.onload = function() {  
      if (req.status === 200) {  
        resolve(req.response); // Resolve the promise with the response text  
      }  
      else {  
        reject(Error(req.statusText)); // HTTP Error→ reject with statusText  
      }  
    };  
  
    // Handle network errors  
    req.onerror = function() {  
      reject(Error("Network Error"));  
    };  
  
    req.send(); // Make the request  
  });  
}
```

Ahora la llamada se haría de la siguiente forma:

```
get('localhost:3000/posts').then(function(response) {  
  console.log("Success!", response);  
}, function(error) {  
  console.error("Failed!", error);  
})
```

Como se puede ver ahora tenemos una serie de ventajas que ya hemos comentado anteriormente:

- Podemos trabajar de forma independiente la creación y gestión del objeto XHR de la gestión del resultado de la llamada.
- El código es más claro y sencillo de implementar.
- Podemos reutilizar la promesa tantas veces como queramos sin tener que implementar más objetos XHR ni utilizar clases