

Code

Issues

Pull requests 1

Actions

Projects








Security

...

 master ▾

...

[FT-M3](#) / 02-Promises /

 JJSolari ...	on 8 Apr 
..	
 demo	2 years ago
 homework	7 months ago
 README.json	7 months ago
 README.md	9 months ago
 ejemplos.js	3 years ago



Hacé click acá para dejar tu feedback sobre esta clase.



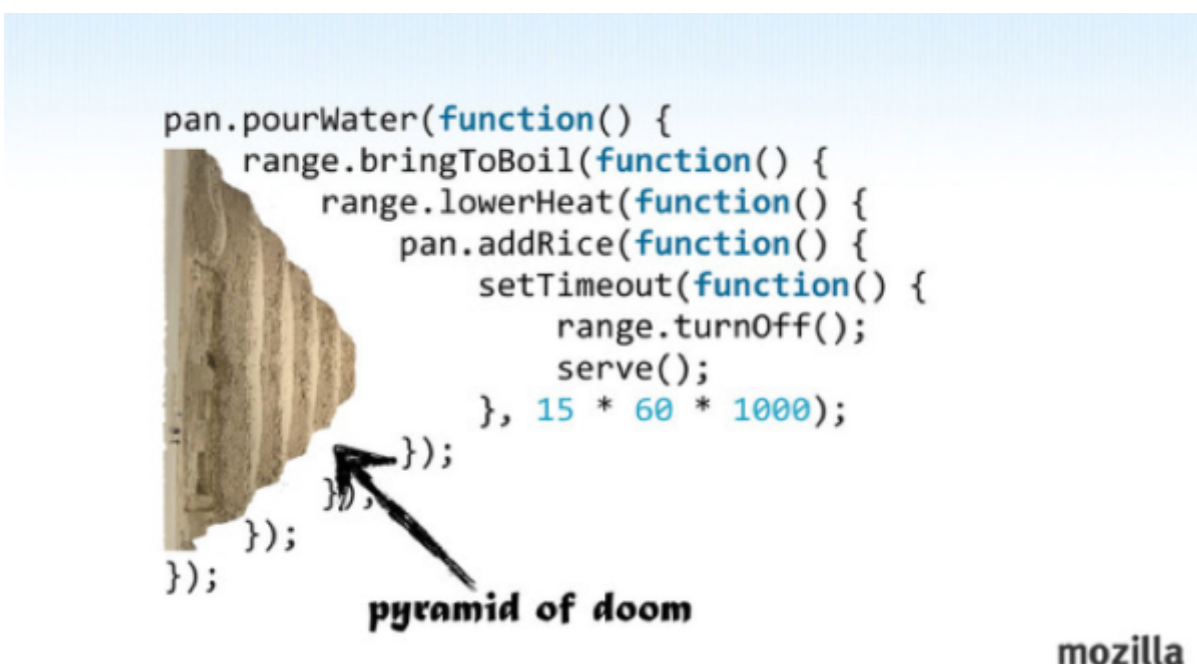
Hacé click acá completar el quizz teórico de esta lecture.

# Promises

Javascript es muy potente a la hora de trabajar con tareas asíncronas, de hecho, ya venimos programando funciones y código que hacen uso de `callbacks` para ejecutar código en el futuro cercano, cuando un evento sucede o cuando se termina la ejecución de un proceso (escribir en la bd, o escribir en el disco, hacer un request http, etc...). Esto es genial, pero a veces nos sucede que tenemos callbacks anidadas, es decir, que dentro de un callback tenemos otro callback y así sucesivamente (inception de callbacks), y también tenemos problemas donde tenemos que esperar que *dos o más* eventos terminen para continuar la ejecución de nuestro código. Si bien podemos resolverlo sin problemas con callbacks, vamos a ver que nuestro código empieza a hacerse difícil de leer, muy difícil de controlar si hay errores (no sabemos qué función es la que realmente produjo el error), y si tenemos que buscar un bug dentro del código nos daremos cuenta que, sin querer, hemos terminado dentro del 🤪 **Callback Hell**:



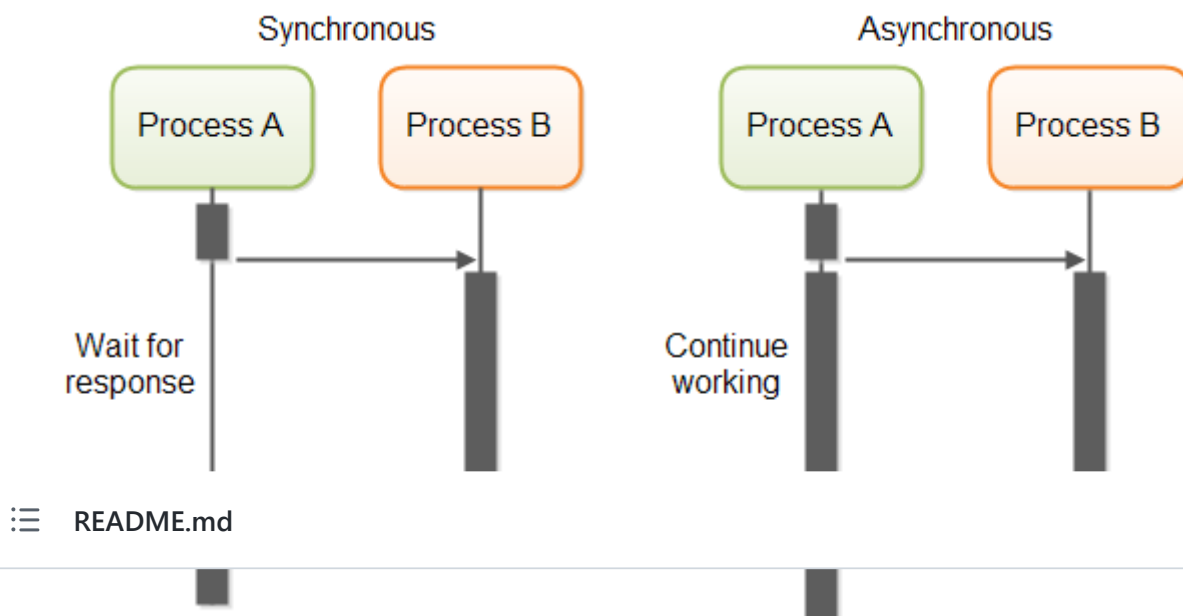
También conocido como Pyramid of Doom 🤪:



Se pueden imaginar, por los nombres que eligieron para esto, que no es una situación deseada en nuestro código. También van a pensar, que es un mal necesario, ya que de esta forma puedo lograr que mi aplicación se comporte de acuerdo a los requerimientos asincrónicos que hay en el medio.

## Promises al rescate

Cuando programamos en lenguajes que son bloqueantes, como `c++` o `python`, perdemos el poder del asincronismo, pero ganamos legibilidad, ya que una línea de código se ejecuta exactamente cuando termina al anterior (si la anterior tarda 3 horas, vamos a esperar a que termine), esto hace que el código sea fácil de leer ya que siguiendo la línea de ejecución vamos a ver que cosas suceden antes o después, esto mismo no lo podemos hacer con callbacks (o por lo menos sin entrar al *Callback hell*):



¿No sería genial si pudieras escribir código como si fuera sincrónico, pero que la ejecución fuera asincrónica? Esta pregunta seguramente se hicieron los inventores de las **Promises** de javascript! Justamente las **Promises** en Js intentan solucionar el problema del *callback hell* y lograr que el código sea más legible, más fácil de debuggear y que tengamos mayor control sobre los errores. Veamos como funcionan las promises.

## 🔗 ¿Qué es una Promesa?

Una *promesa* representa el resultado eventual (en un futuro incierto) de una operación asincrónica, como por ejemplo: un registro de una db, una página que pedimos por http, un objeto JSON que es respuesta de un request a un API. Es decir, que representa un *placeholder* (un lugar reservado) donde vamos a guardar la respuesta del método asincrónico ( o un posible error en caso que no sea exitosa ). Como nos podemos imaginar, una promesa puede ser exitosa o no, y una misma promesa no se puede ejecutar dos veces, una vez que termina se convierte en **immutable** (no puede cambiar). Además si a una promesa le agregamos un callback (le podemos agregar en cualquier momento), este será ejecutado cuando esta termine. Esto es genial, porque ya no nos interesa en qué momento se producen las cosas, si no en reaccionar al resultado de esas cosas.

## Terminología de promises

Una promesa puede estar:

- *Pendiente* (pending): El estado inicial de un promise.
- *Completada* (fulfilled): Representa que se completó de manera exitosa.
- *Rechazada* (rejected): La operación terminó, pero de manera fallida.
- *Terminada* (settled): La operación terminó, de cualquiera de las dos maneras anteriores.

## Promises en JavaScript

---

Antes de **ES6** (EcmaScript 6) javascript no soportaba nativamente las *Promises*, pero como el concepto ya estaba dando vuelta en la cabeza de algunos desarrolladores aparecieron varias librerías que lo implementaban, por ejemplo:

- [Q](#)
- [When](#)
- [WinJS](#)
- [RSPV.js](#)
- [Bluebird](#)

Si bien el concepto es el mismo, y de hecho se creó un standard para las promises: el [Promises/A+](#), no todas las implementaciones son iguales ni respetan el standard. Ustedes se preguntarán porque no usamos directamente las promises que definieron los científicos de EcmaScript, pero resulta que hay un *serio* debate online por las performances de las mismas en términos de tiempo y memoria, la gente de Bluebird hizo algunos [benchmarks](#), y si los examinamos vemos que las promesas *nativas* de **ES6** están todavía muy abajo en la lista. Esto es debido a la implementación de cada librería sobre las promesas, y por lo visto la implementación nativa no es tan buena todavía. Así que por ahora todavía nos conviene usar alguna librería externa, como por ejemplo *Bluebird* que veremos más abajo.

El autor de *Bluebird* explica un poco porqué tanta diferencia de performance en este [link](#).

## Creando una promesa

Una *Promise* es una clase en JavaScript, así que para instanciar una nueva vamos a el keyword `new`. Una *Promise* recibe un sólo argumento: una función con dos parámetros: `reject` y `resolve`. Dentro de esta función vamos a hacer lo que necesitemos y, si todo salió bien, llamamos `resolve` y si no, (algo salió mal), llamamos `reject`.

```
var promise = new Promise(function(resolve, reject) {  
  // Hacer cosas acá dentro, probablemente asíncronicas.  
  
  if (/* Todo funcionó como esperabamos */) {  
    resolve("Jooya!");  
  }  
  else {  
    reject(Error("Algo se rompió"));  
  }  
});
```

En el método `reject` podemos pasar cualquier cosa, pero se recomienda devolver un objeto de tipo `Error` ya que estos tienen el `stack trace` y es más fácil de debugear.

## Haciendo algo cuando una promesa se 'cumple'

En el statement anterior hemos *definido* una promesa y la hemos guardado en la variable `promise`. Ahora, para usar esa promesa, debemos de alguna forma poder decirle qué hacer cuando se resuelva o se rechaze la promesa. Para eso vamos a usar el método `then`:

```
promise.then(function(data) {  
  // Ejecuto código sabiendo que todo salió bien  
  // Siguiendo el ejemplo de arriba, data tendría adentro el string: 'Jooya!'  
}, function(error) {  
  // La promesa fue rechazada, aca ejecutamos código para ese caso  
  // Siguiendo el ejemplo de arriba, error tendría adentro el string: 'Algo se romp  
});
```

La función `then` de las *promises* recibe dos argumentos, un callback de `sucess` y un callback de `failure`, que van a ser llamadas según si el promise terminó en un `resolve` o un `reject` respectivamente. Los parámetros que le llegan a esta función (en el ejemplo `data` y `error`) son los mismos parámetros con los que llamamos a las funciones `resolve` y `reject`. Podemos escribir lo mismo que arriba, pero en vez de pasar dos callbacks a `then`, vamos a usar otro método similar llamado `catch`. Básicamente, al separarlos de este modo, con el `then` vamos a llamar un callback cuando el Promise termina en éxito, y con `catch` vamos a llamar un callback cuando termina en error:

```
promise.then(function(data) {  
  // Ejecuto código sabiendo que todo salió bien  
  // Siguiendo el ejemplo de arriba, data tendría adentro el string: 'Jooya!'  
}).catch(function(error) {  
  // La promesa fue rechazada, aca ejecutamos código para ese caso  
  // Siguiendo el ejemplo de arriba, error tendría adentro el string: 'Algo se rompió'  
});
```

Este último pedazo de código con `then-catch` es equivalente (hace lo mismo) que el anterior donde `then` recibe dos callbacks. Es simplemente una forma más simple de escribir código.

## Encadenando Promesas

Algo muy potente de las *Promises* es que vamos a poder encadenarlas (*chaining*), ya que podemos hacer que un *Promise* **retorne** otro *Promise*, y de esa forma ir encadenando métodos. Por ejemplo:

```
var primerMetodo = function() {  
  var promise = new Promise(function(resolve, reject){  
    setTimeout(function() {  
      console.log('Terminó el primer método');  
      resolve({num: '123'}); //pasamos unos datos para ver como los manejamos  
    }, 2000); // para simular algo asincronico hacemos un setTimeout de 2 s  
  });  
  return promise;  
};  
  
var segundoMetodo = function(datos) {  
  var promise = new Promise(function(resolve, reject){  
    setTimeout(function() {  
      console.log('Terminó el segundo método');  
      resolve({nuevosDatos: datos.num + ' concatenamos texto y lo pasamos'});  
    }, 2000);  
  });  
  return promise;  
};
```


```

var tercerMetodo = function(datos) {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el tercer método');
      console.log(datos.nuevosDatos); //imprimos los datos concatenados
      resolve('hola');
    }, 3000);
  });
  return promise;
};

primerMetodo()
  .then(segundoMetodo)
  .then(tercerMetodo)
  .then(function(datos){
    console.log(datos); //debería ser el 'hola' que pasamos en tercerMetodo
  });

```

En el ejemplo hemos creado tres métodos donde simulamos algo asíncronico, o sea que no sabemos cuando se va a terminar de ejecutar y como vemos, todos crean una *Promise* nueva dentro de ellos y la **retornan**. Para llamarlos, invocamos al primer métodos y le decimos con `then` que si termina exitosamente ejecute la función `segundoMetodo`, esta también devuelve una *Promise*, por lo tanto también podemos llamar a `then` sobre ella, con esto invocamos `tercerMetodo` (que tambien retorna una *Promise*) y a este última le pasamos una función anónima pidiendo que imprima por consola los *datos* que recibió como argumento en `resolve`. Si lo ejecutan en su browser verán cómo es el flujo de datos y en qué orden se imprimen los `console.log`s.

Intenten hacer el mismo ejemplo, pero sin usar Promises. Haciéndolo van a notar la diferencia y vean cuan inentendible puede ser el  *Callback Hell*

## Esperando que varias Promesas se cumplan para hacer algo

A veces no sólo necesitamos esperar por un evento para hacer algo, si no que queremos que varios eventos hayan sucedido para actuar. Por ejemplo, quiero guardar un arreglo de `alumnos` en una base de datos, y cuando estén todos correctamente guardados mostrar la respuesta al usuario. Para eso, la *API* de *Promises* no da el método `all`, el cual toma un arreglo de *Promises* y crea un *Promise* que se completa cuando todas las *Promises* que les pasamos estén completas. Al devolver una *Promise* vamos a poder invocar el método `then` sobre ella, y en el callback que les pasemos vamos a recibir como argumento un arreglo de los resultados de las *Promises* que les pasamos, en el mismo orden que se las pasamos:

```

Promise.all(arregloDePromesas).then(function(arregloDeResultados) {
  //...
});

```

Vamos a usar el ejemplo anterior, pero ahora queremos que los métodos se ejecuten sólo cuando se completaron los tres, para eso vamos a tener que cambiar cada método para que no utilice ningún dato del método anterior (ahora estamos llamando los tres al mismo tiempo) y la última parte y reemplazarla por `all`, en el callback de esta vamos a hacer un `console.log` del arreglo que nos devolvió:

```
var primerMetodo = function() {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el primer método');
      resolve({num: '123'}); //pasamos unos datos para ver como los manejamos
    }, 2000); // para simular algo asincronico hacemos un setTimeout de 2 s
  });
  return promise;
};

var segundoMetodo = function() {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el segundo método');
      resolve({texto: ' segundo metodo'});
    }, 1000);
  });
  return promise;
};

var tercerMetodo = function(datos) {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el tercer método');
      resolve({hola:1});
    }, 3000);
  });
  return promise;
};

Promise.all([primerMetodo(), segundoMetodo(), tercerMetodo()])
  .then(function(resultado){
    console.log(resultado); //un arreglo con los valores pasamos a resolve en cada meto
  });
```

Genial, como vemos las promesas se completaron según cuanto tardaba cada una, en el ejemplo pusimos a propósito que el segundo método tarde menos que los demás. Lo importante a notar es que en el arreglo que recibimos al final los resultados vienen ordenados en el mismo orden en el que pasamos las *promises* a `all` y *no* en el orden en que se resuelven.



Seguramente se preguntarán qué pasa cuando una *Promise* es rechazada ( o sea que su ejecución termina en un `reject` ). `Promise.all` fue diseñada con un comportamiento llamado *fail-fast*, esto quiere decir que ni bien existe un `reject`, `all` ya lo reporta y devuelve el error. Es decir que el método `catch` va a seguir cómo antes, recibiendo sólo un parámetro (el error), y este corresponderá a la *Promise* que falló primero.

Modifiquemos el código para que el método del medio termine en error. Vamos a rechazar dos promesas, una que termine primero y otra al final, vamos a ver que siempre se mostrará el error de la que **falla primero**:

```
var primerMetodo = function() {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el primer método');
      reject('Esto es una prueba controlada');
    }, 2000); //
  });
  return promise;
};

var segundoMetodo = function() {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el segundo método');
      resolve({texto: 'segundo metodo'});
    }, 1000);
  });
  return promise;
};

var tercerMetodo = function(datos) {
  var promise = new Promise(function(resolve, reject){
    setTimeout(function() {
      console.log('Terminó el tercer método');
      reject('Tercer método falla');
    }, 3000);
  });
  return promise;
};

Promise.all([primerMetodo(), segundoMetodo(), tercerMetodo()])
  .then(function(resultado){
    console.log(resultado); //un arreglo con los valores pasamos a resolve en cada meto
  })
  .catch( function(err){
    console.warn(err); //mostramos el error por consola. Veremos que es el que falló pr
  });
```

Noten que la Promise retorna y ejecuta el `catch` inclusive antes que termine el tercer método, esto se debe a que el primer método falla antes que se ejecute el tercero.

## BlueBird

---

BlueBird es una librería enfocada en *Promises*, esta cumple con el standard de [Promises/A+](#). Como dijimos antes, esta librería es más performante que las *Promises* nativas de **EC6** y sus diseñadores se aprovecharon de ello ya que las dos librerías son perfectamente intercambiables, es decir, que la sintaxis de *Promises* en nativo es idéntica a la de *Bluebird* (excepto en un [estos](#) casos puntuales). Por lo tanto, para usarla vamos primero a instalarla: `npm install bluebird` y luego, vamos a reemplazar el objeto `Promise` que contiene la clase de las *Promises* con el módulo de *Bluebird*:

```
var Promise = require("bluebird");
```

Listo! ahora podemos usar las Promesas de Bluebird mejorando la performance de nuestra aplicación!

Para ver todo lo que se puede hacer con *Promises* podemos ir a leer la [documentación](#) de Bluebird.

## Homework

---

Completa la tarea descrita en el archivo [README](#)

---