Índice

[**API** 1](#_Toc84187898)

[**Concurrencia** 1](#_Toc84187899)

[**Paralelismo** 1](#_Toc84187900)

[**Bloqueante** 1](#_Toc84187901)

[**Síncrono** 2](#_Toc84187902)

[**Asíncrono** 2](#_Toc84187903)

[**Call Stack** 2](#_Toc84187904)

[**¿Para qué sirven?** 4](#_Toc84187905)

[**Heap** 4](#_Toc84187906)

[**Cola o Queue** 5](#_Toc84187907)

[**Eventloop o Loop de eventos** 5](#_Toc84187908)

[**Hoisting** 5](#_Toc84187909)

[**DOM** 5](#_Toc84187910)

[**XML** 6](#_Toc84187911)

[**Events** 6](#_Toc84187912)

[**Compilar** 6](#_Toc84187913)

[**Transpilar** 6](#_Toc84187914)

[**Promesas** 7](#_Toc84187915)

[**Crear una promesa** 8](#_Toc84187916)

[**Estados de las promesas** 8](#_Toc84187917)

[**Promesas con parámetros** 9](#_Toc84187918)

[**Encadenando promesas** 11](#_Toc84187919)

[**Promesas en paralelo** 12](#_Toc84187920)

[**Carrera de promesas** 13](#_Toc84187921)

[**Promesas resueltas inmediatamente** 13](#_Toc84187922)

[**Promesas rechazadas inmediatamente** 14](#_Toc84187923)

[**Conociendo Async/await** 15](#_Toc84187924)

[**Ventajas y desventajas:** 16](#_Toc84187925)

[**Callbacks Vs Promesas Vs Async/Await** 16](#_Toc84187926)

[**Callbacks** 16](#_Toc84187927)

[**Promesas** 16](#_Toc84187928)

[**Async/Await** 17](#_Toc84187929)

# **API**

Interfaz de programación de aplicaciones (Application Programming Interface). Es un conjunto de rutinas que provee acceso a funciones de un determinado software.

# **Concurrencia**

Cuando dos o más tareas progresan simultáneamente.

# **Paralelismo**

Cuando dos o más tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.

# **Bloqueante**

Una llamada u operación bloqueante no devuelve el control a nuestra aplicación hasta que se ha completado. Por tanto, el **thread** queda bloqueado en estado de espera.

# **Síncrono**

Es frecuente emplear ‘**bloqueante’** y ‘**síncrono’** como sinónimos, dando a entender que toda la operación de **entrada**/**salida** se ejecuta de forma secuencial y, por tanto, debemos esperar a que se complete para procesar el resultado.

# **Asíncrono**

La finalización de la operación I/O se señaliza más tarde, mediante un mecanismo específico como por ejemplo un callback, una promesa o un evento, lo que hace posible que la respuesta

sea procesada en diferido.

# **Call Stack**

Los **callbacks** son el nombre de una **convención** para usar funciones que llaman a otras en JavaScript. No hay una palabra reservada llamada “callback” en el lenguaje JavaScript que haga que nuestro código sea diferente o especial, **es más una convención**.

**Ejemplo**





## **¿Para qué sirven?**

La mayoría estamos acostumbrados a programar de manera **síncrona**, es decir, le indicamos al código que por ejemplo defina un Valor “**X**” y con otro valor “**Y**” y realizamos un cálculo (Por ejemplo, una multiplicación).

El problema radica en que por ejemplo si quisiéramos crear un programa que nos convierta nuestra moneda (**pesos**) a su equivalente en **Bitcoin**, podemos definir **X** (Valor de nuestro dinero) pero **NO** podemos definir de manera implícita “**Y**” (Precio del Bitcoin) porque es algo muy **volátil**. Entonces necesitamos obtener el precio del Bitcoin de una API, nuestro programa realiza una multiplicación de **X** \* **Y** sin embargo no tenemos **Y** (precio del bitcoin) porque tenemos que esperar que el API nos conteste cual es este valor. Es ahí donde sirven los **callback**.

Existen dos Métodos A y B

* El método **B** hace el cálculo de nuestros **Pesos** \* **PrecioBitcoin**
* El método **A** obtiene el precio del API de Bitcoin.

Entonces el método **B** es llamado por el método **A** cuando por fin lee y obtiene el precio del Bitcoin, solo hasta entonces tiene sentido que multipliquemos nuestros valores.

# **Heap**

Región de memoria libre, normalmente de gran tamaño, dedicada al alojamiento dinámico de objetos. Es compartida por todo el programa y controlada por un recolector de basura que se encarga de liberar aquello que no se necesita.

# **Cola o Queue**

Cada vez que nuestro programa recibe una notificación del exterior o de otro contexto distinto al de la aplicación, el mensaje se inserta en una cola de mensajes pendientes y se registra su callback correspondiente.

# **Eventloop o Loop de eventos**

Cuando la pila de llamadas (call stack) se vacía, es decir, no hay nada más que ejecutar, se procesan los mensajes de la cola. Con cada ‘tick’ del bucle de eventos, se procesa un nuevo mensaje.

# **Hoisting**

Sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código en tiempo de compilación.

# **DOM**

DOM permite acceder y manipular las páginas XHTML como si fueran documentos XML. De hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.

# **XML**

Lenguaje de marcado creado para la transferencia de información, legible tanto para seres humanos como para aplicaciones informáticas, y basado en una sencillez extrema y una rígida sintaxis. Así como el HTML estaba basado y era un subconjunto de SGML, la reformulación del primero bajo la sintaxis de XML dio lugar al XHTML; XHTML es, por tanto, un subconjunto de XML.

# **Events**

Comportamientos del usuario que interactúa con una página que pueden detectarse para lanzar una acción, como por ejemplo que el usuario haga click en un elemento (onclick), que elija una opción de un desplegable (onselect), que pase el ratón sobre un objeto (onmouseover), etc.

# **Compilar**

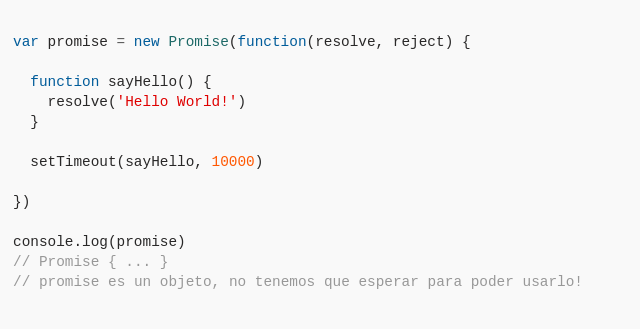
Compilar es generar código ejecutable por una máquina, que puede ser física o abstracta como la máquina virtual de Java.

# **Transpilar**

Transpilar es generar a partir de código en un lenguaje código en otro lenguaje. Es decir, un programa produce otro programa en otro lenguaje cuyo comportamiento es el mismo que el original.

# **Promesas**

Una promesa **es un objeto que guarda una operación asíncrona**.



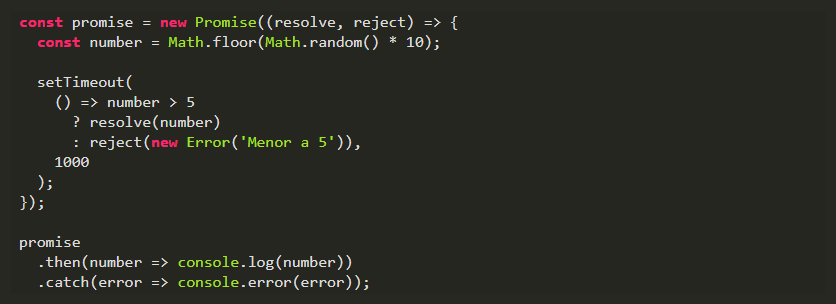
La variable **promise** es un objeto y podemos operar con él independientemente de que hasta dentro de 10 segundos no nos devuelva un valor y contenga el string ‘**Hello world**’.

Esto nos permite tener mayor control. Antes, la ejecución de nuestra app dependía de una llamada asíncrona que no sabíamos si iba o no iba a acabar. **Con las promesas seguimos trabajando independientemente de lo que pase con la llamada asíncrona**.

Además, contamos con la ventaja de que, si el código lanza una **excepción** dentro de la **Promesa**, ésta la captura y la devuelve con la función ‘**catch’**.

## **Crear una promesa**

Las promesas se crean usando un constructor llamado **Promise** y pasándole una función que recibe dos parámetros, **resolve** y **reject**, que nos permiten indicarle a esta que se **resolvió** o se **rechazó**.



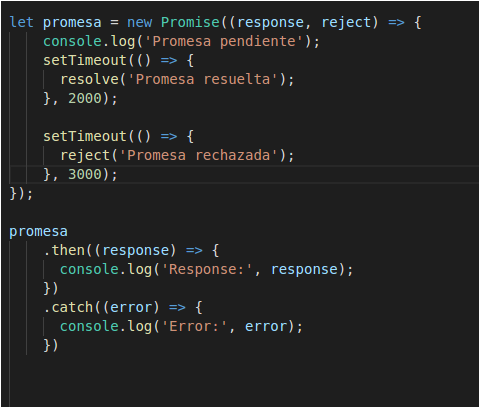
Lo que acabamos de hacer es crear una nueva promesa que se va a completar luego de 1 segundo, si el número aleatorio que generamos es mayor a 5 entonces se resuelve, si es menor a 5 entonces es rechazada y obtenemos un error.

## **Estados de las promesas**

Esto nos lleva a hablar del estado de una promesa, básicamente existen 3 posibles estados.

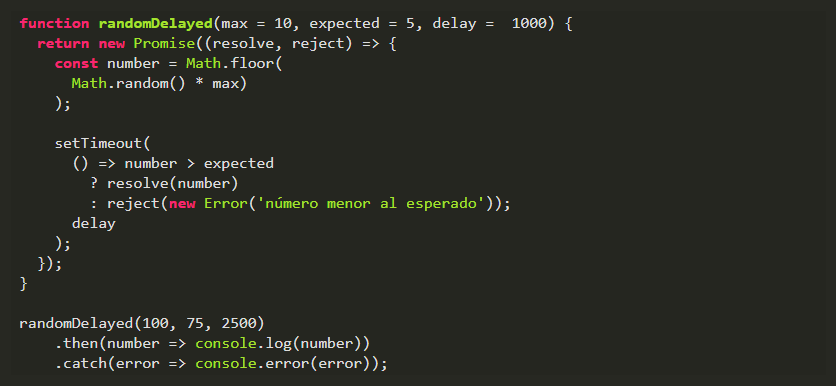
* pending (pendiente)
* fullfilled (resuelta exitosamente)
* rejected (rechazada)

Una promesa originalmente está **Pendiente**. Cuando llamamos a **resolve** entonces la promesa pasa a estar **Resuelta**, si llamamos a **reject** pasa a estar **Rechazada**, usualmente cuando es rechazada obtenemos un error que nos va a indicar la razón del rechazo. Cuando una promesa se resuelve entonces se ejecuta la función que pasamos al método **.then**, si la promesa es rechazada entonces se ejecuta la función que pasamos a **.catch**, de esta forma podemos controlar el flujo de datos.



## **Promesas con parámetros**

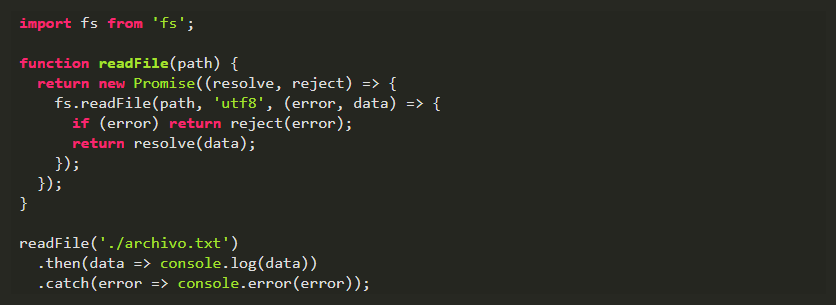
Es muy sencillo, basta con crear una función que reciba esos parámetros, meter la promesa dentro y devolver su instancia.



Cuando ejecutamos **randomDelayed(100, 75, 2500)** creamos una promesa que luego de 2.5 segundos se va a resolver siempre que el número generado (entre 0 y 100) sea mayor a 75. Lo mismo que habíamos hecho antes, pero esta vez personalizable.

Pasando de callback a promesas

¿Qué ocurre si una función que queremos utiliza callbacks? ¿Cómo podríamos usarla con promesas? Muy simple, podemos crear una versión con promesas de esa función haciendo lo que hicimos arriba. Por ejemplo, leer un archivo usando el módulo **fs** de **Node.js**.

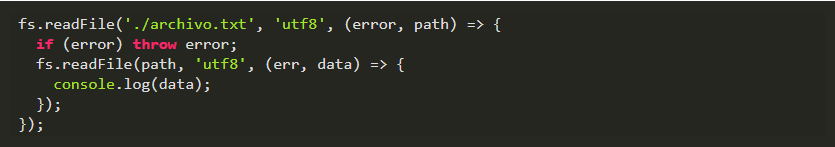


De esta forma creamos una función que lee un archivo del disco como utf8 y si no ocurre ningún error entonces se **resuelve**, si hay un error es **rechazada**.

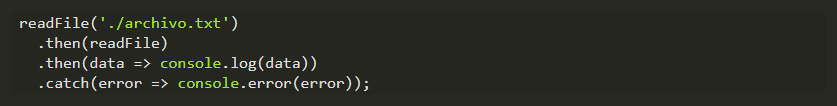
## **Encadenando promesas**

Básicamente nos evita anidar código, en vez de eso una promesa puede devolver otra promesa y llamar al siguiente **.then** de la cadena.

Supongamos que archivo.txt devuelve un string con el path de otro archivo, y queremos leer este segundo archivo, con callbacks quedaría algo así:



Como vemos dentro de nuestro primer callback tenemos que validar el primer error, luego llamar a otra función que obtiene los datos de verdad, y si tenemos que ir anidando muchas funciones que usen callback podemos llegar a tener muchos niveles de indentación. Con promesas esto quedaría así:

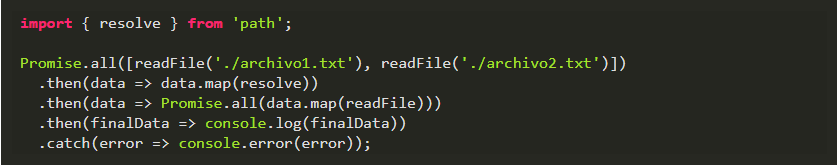


Qué hacemos acá, primero leemos **./archivo.txt**, si ocurre un error esta promesa se rechaza y lo mostramos en el **console.error**, si todo va bien se ejecuta el primer **.then**, este ejecuta un nuevo **readFile**, como **.then** recibe el path al nuevo archivo y **readFile** solo recibe un argumento (el path) entonces podemos pasar directamente **readFile** y la promesa se encarga de ejecutarlo.

Este segundo **readFile** devuelve una nueva **promesa**, otra vez si hay un error se ejecuta el .**catch**, pero si podemos leer el archivo sin problema entonces se ejecuta el segundo .**then**, el cual recibe el contenido del segundo archivo y lo muestra en consola.

## **Promesas en paralelo**

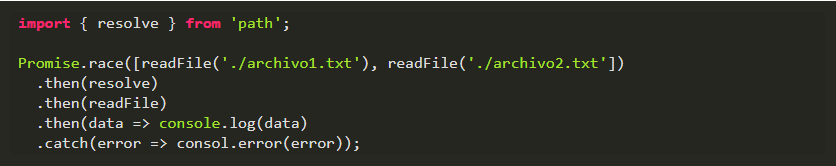
Hasta ahora solo vimos como ejecutar una función asíncrona a la vez (en serie), sin embargo, es muy común que necesitemos realizar múltiples al tiempo, por ejemplo, para obtener varios datos de un API. Para eso la clase **Promise** tiene un método estático llamado **Promise.all** el cual recibe un único parámetro, **una lista de promesas** las cuales se ejecutan simultáneamente**, si alguna de estas es rechazada entonces toda la lista lo es**, pero si todas se resuelven entonces podemos obtener una lista de todas las respuestas.



Lo que hacemos en el ejemplo de arriba es leer 2 archivos al mismo tiempo, eso nos devuelve una lista (**data**) de contenidos, los cuales contienen la ruta para otro archivo, los convertimos entonces a una nueva lista de rutas absolutas (**resolve**) y usamos esas rutas para crear una nueva lista de promesas a partir de **readFile**. Si en algún momento ocurrió un error lo mostramos como tal en consola, si todo se resuelve bien entonces escribimos en consola la lista de contenidos de archivos.

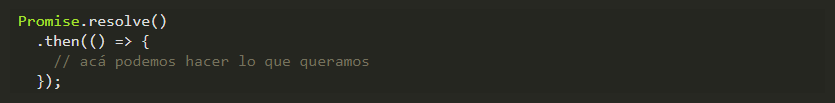
## **Carrera de promesas**

Antes hablamos de ejecutar varias promesas en paralelo y obtener una respuesta cuando todas se completen, existe otro método que nos permite correr varias al mismo tiempo, pero solo obtener el resultado de la **primera promesa**. Gracias a esto es posible mandar múltiples peticiones HTTP a un API y luego recibir **una** **sola respuesta**, la **primera**. Este método se llama **Promise.race**.

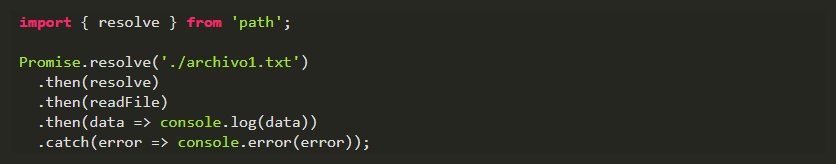


## **Promesas resueltas inmediatamente**

Algunas veces la forma de manejar el flujo de datos de las promesas encadenando then nos facilita trabajar con nuestro código, para eso podemos crear una promesa que inicie resuelta directamente usando un método estático de **Promise**.



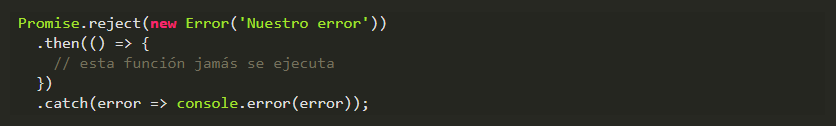
Otra opción es pasarle un parámetro a **resolve** para que nuestro primer **then** reciba ese valor.



Como vemos iniciamos la cadena con un string y desde ahí obtenemos el path absoluto, leemos el archivo y lo mostramos en consola.

## **Promesas rechazadas inmediatamente**

De la misma forma que creamos promesas resuelta inmediatamente podemos crear promesas rechazadas. Solo que esta vez usamos **Promise.reject**

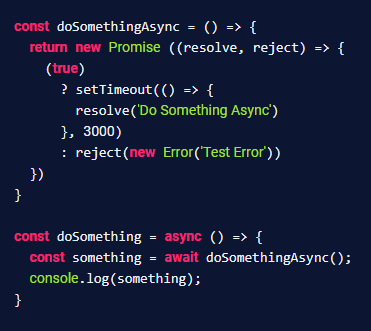


**¿Para qué nos sirve esto?**

Si tenemos una función síncrona que queremos resolver mediante promesas podríamos si da un error devolver Promise.reject con el error y Promise.resolve con la respuesta correcta. De esta forma en vez de crear una instancia de la clase Promise simplemente devolvemos la **promesa ya resuelta** o **ya rechazada**.

# **Conociendo Async/await**

**Async**/**await** no es más que **Syntax Sugar**. Es una manera más bonita de hacer lo mismo que estábamos haciendo con **.then()**. La clave es recordar que, si una función regresa una **promesa**, podemos usar el keyword **await**, que le indicia al navegador: “**Espera a que la promesa se resuelva y almacena su resultado en esta variable**”. Todo esto toma lugar dentro de una función asíncrona, así que usamos **async** para lograr esto.



# **Ventajas y desventajas:**

# **Callbacks Vs Promesas Vs Async/Await**

## **Callbacks**

* **Ventajas**
  + **Simpleza**: Nos permite disponer de una sintaxis fácil de entender y comprender qué sucederá al ser ejecutada.
  + **Compatibilidad**: Los Callbacks son funciones que no necesitan convertir tu código con un polyfill (**es una funcionalidad que debería de tener un navegador, pero no la tiene, el polyfill suple esa carencia**) para que funcionen con todos los navegadores modernos o versiones anteriores.
* **Desventajas**
  + Pueden disponer de una estructura que puede llegar a ser robusta, más cuando anidamos llamadas a otras funciones, su flujo se puede volver poco intuitivo lo cual nos hará no comprender claramente su estructura.
  + Manejo de Errores, con los Callbacks no tenemos un camino claro para manejar los errores lo cual se traduce en problemas a la hora de manejar la lógica de nuestro programa.

## **Promesas**

* **Ventajas**
  + **Flujo fluido**: Con las promesas podemos manejar un flujo complejo, anidar llamadas y tener una sintaxis clara que nos permite entender nuestro programa o la lógica que implica su uso.
  + **Manejo de Errores**: Las promesas nos proporcionan una forma clara de manejar errores, una sintaxis sencilla y una forma de entender qué sucederá cuando sean ejecutadas.
* **Desventajas**
  + **PolyFill**: Las promesas no son compatibles con todos los navegadores, si bien los navegadores modernos pueden interpretar sin problema alguno, navegadores como internet Explorer 11 necesitan transpilar el código para que funcionen correctamente.

## **Async/Await**

* **Ventajas**
  + **Sintaxis**: Tienen una sintaxis muy simple y clara de leer, lo que nos permite entender de forma muy sencilla su funcionamiento.
  + **Try/catch**: Podemos utilizar try/catch para el manejo de errores con lo cual podemos manejar una sintaxis clara para el manejo de los errores.
* **Desventajas**
  + **Polyfill**: Como las promesas, Async/Await aún no tienen toda la compatibilidad con los navegadores viejos, por lo cual necesitamos transpilar nuestro código para utilizarlos en cualquier navegador.