**Asincronía y Promesa**

**El tiempo (o la experiencia) vale oro**

\*“Si logras construir una gran experiencia, los clientes se lo dicen unos a otros. El boca a boca es muy poderoso.” [**Jeff Bezos**](https://es.wikipedia.org/wiki/Jeff_Bezos) - CEO de Amazon.com\*



Photo by [NordWood Themes](https://unsplash.com/@nordwood?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/waiting?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

El destino final de este programa es enseñarte a construir el backend para una aplicación. Durante el primer sprint adquiriste habilidades que te permitieron construir y documentar una API; aprendiste sobre HTML y conociste los fundamentos de programación en JavaScript. En este segundo sprint, ¡iremos por más! Aprenderás sobre bases de datos, caché, autentificación, testing y arquitectura de una aplicación, entre otras cosas.

Para alcanzar esta misión conocerás un nuevo lenguaje, SQL (por sus siglas en inglés, Structured Query Language) para manejar bases de datos relacionales y no relacionales. Si no sabes lo que son, no te preocupes, pronto veremos las diferencias entre ambas.

Hacia el final del sprint, serás capaz de almacenar en una base de datos toda la información a través de tu API, así como también podrás agregar nuevas funcionalidades. ¡Sigue por este camino! Al finalizar, no sólo habrás recorrido ya la mitad del programa (¡wow!) sino que estarás un paso más cerca de ser un desarrollador backend. Ahora, es momento de continuar…

En esta toolbox, te presentaremos 3 conceptos claves que te servirán de ahora en adelante: asincronía, callback y promesas. Veamos un poco más sobre estos y cómo van a serte de utilidad en este nuevo sprint.

**Asincronía**

Cuando desarrollamos, nos interesa poder gestionar el modo en que las instrucciones que le damos a un programa son ejecutadas, porque estaremos determinando —o al menos seremos conscientes de— el tiempo de ejecución. Podremos hacer que nuestro programa sea ‘multitasker’ o asincrónico (ejecutará una acción tras otra sin esperar respuestas) o que sea ‘unitasker’ o sincrónico (ejecutará una acción, esperará la respuesta, y recién ahí ejecutará una nueva acción).

Cuando un programa desea acceder a un recurso material, no necesita enviar información específica a los dispositivos periféricos (teclado, mouse, cámara, etc): simplemente envía la información al sistema operativo, el cual la transmite a los periféricos correspondientes a través de su controlador (¡el famoso *driver*!).

Es aquí donde para los/as developers empieza lo divertido. La base de JavaScript que aprendiste en el sprint anterior te preparó para lo que se viene. En esta toolbox aprenderás sobre el manejo de los tiempos de una aplicación con los diferentes métodos que utilizan los lenguajes de programación.

**El tiempo de las aplicaciones**

Para cumplir su función, una aplicación se alimenta de entradas, realiza un proceso y produce una salida. Como ya te anticipamos, este conjunto de acciones ocurre en un tiempo de ejecución. Hay dos tipos de procesos de ejecución, que se corresponden con dos tipos de métodos utilizados por los lenguajes de programación: el método sincrónico y el método asincrónico.

* **Método sincrónico.** La ejecución espera a que un resultado sea devuelto para ejecutar la siguiente instrucción.La mayoría de los lenguajes de programación suelen ser sincrónicos y las instrucciones se ejecutan una detrás de la otra:

console.log("¿Hola ");

console.log("cómo ");

console.log("están? ");

El resultado sería:

¿Hola

cómo

están?

Si tomamos como ejemplo al [lenguaje PHP](https://es.wikipedia.org/wiki/PHP), en el [artículo](http://panamahitek.com/introduccion-al-paradigma-de-la-programacion-asincrona/) de Panama Hitek, sucede lo siguiente:

\*“...Cuando un usuario intenta acceder a un archivo de PHP por medio de un navegador web, un subproceso de Apache2 (el cual se encarga de manejar todas las llamadas a los archivos web y dar soporte a PHP) es llamado. El subproceso no se cierra hasta que el código termina de ejecutarse. Esto quiere decir que si tu archivo tarda un poco en ejecutarse y miles de personas intentan acceder al mismo archivo, se llega a un punto donde no se pueden abastecer a más usuarios debido a la insuficiencia de recursos.” \*

El problema de estos procesos es que ocupan más memoria, por lo que suelen ser menos eficientes.

* **Método asincrónico.**La ejecución no espera un resultado.El proceso es enviado y queda en cola hasta que el sistema operativo decida cuándo están dadas las condiciones para ser ejecutado, y es entonces cuando éste se reanuda.

Por ejemplo, si tenemos que ir a buscar un dato fuera de nuestro sitio web, todo el ida y vuelta tendrá una demora de “x” cantidad de tiempo, en el cual el proceso quedará en “pausa” hasta recibir una respuesta. Para optimizar los tiempos de espera que se producen, una solución típica es asignarle a otro proceso al CPU para que pueda ejecutarse. De este modo el uso del CPU es optimizado. Siguiendo el ejemplo, si tenemos las siguientes líneas de programación:

console.log("¿Hola ");

console.log("cómo ");

-- Realizar una búsqueda en Google (ya veremos las instrucciones)

console.log("están?");

El resultado sería:

¿Hola

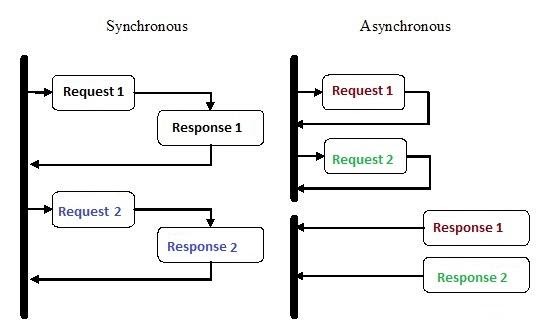
cómo

están?

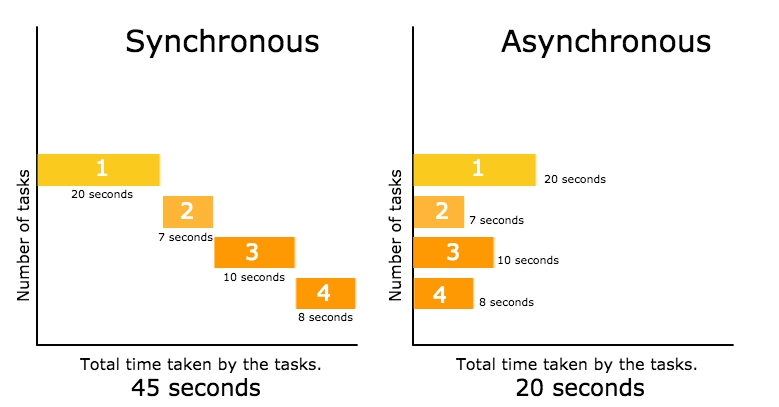
-- resultado de la búsqueda en Google

Como puedes ver, el resultado no fue secuencial, en vez se mostraron los resultados de las instrucciones 1, 2, 4 y 3, ya que la aplicación no esperó a que termine la instrucción de buscar algo en Google.

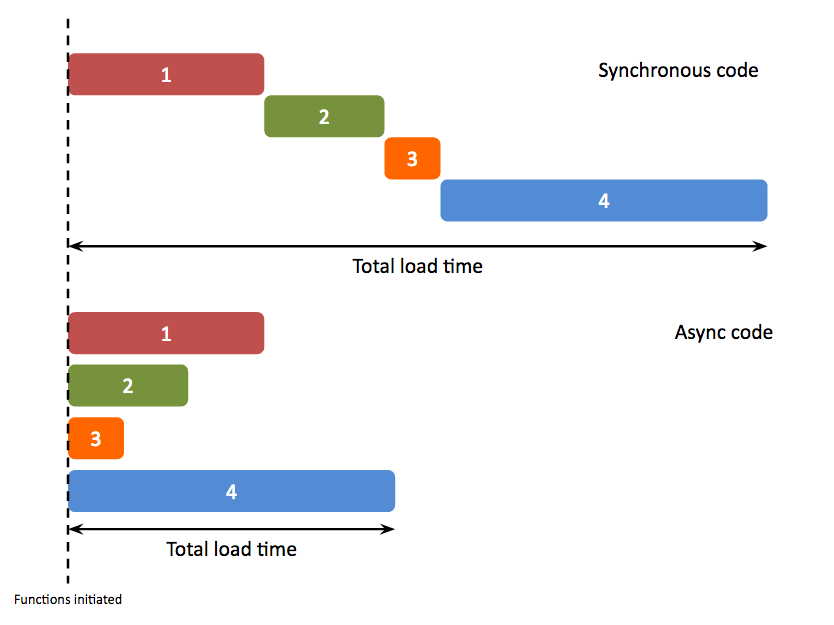
En el siguiente gráfico puedes ver las diferencias de tiempo entre ambos métodos:



Otras formas de verlo son:



Fuente: [PHP mind](http://www.phpmind.com/blog/2017/05/synchronous-and-asynchronous/)



Fuente: [Cambridge Intelligence](https://cambridge-intelligence.com/using-javascript-promises-in-keylines-part-1/)

**Callback**

Este concepto de Callback te será de suma utilidad para trabajar con asincronía, debido a que cómo hemos visto recientemente, para optimizar la velocidad de nuestro desarrollo, hay sentencias que se ejecutan y no sabemos cuando terminan, pero al terminar algo deberá ejecutarse en nuestro código para continuar con la acción, allí es donde entran los callbacks.

¿Qué es un callback? Un callback es una **función que se pasa a otra función con la expectativa de que la otra función la llame.**

Veamos un ejemplo que nos ayude a clarificar la teoría de la definición.

**PASO 1.**

Imagina que definimos tres funciones: saludar, bienvenida y adiós. Todas son independientes entre sí y cualquiera podría ser ejecutada de manera independiente desde nuestro hilo principal de la aplicación.

function saludar(callback) {

callback();

}

function bienvenida(){

console.log("Hola mundo")

}

function adios(){

console.log("Adios mundo")

}

La función saludar espera un parámetro que se le va a asignar a “callback”. La primera y única línea de esa función es tomar el dato del parámetro y ejecutarlo. Debido a que lo que queremos ejecutar dentro de “saludar” es una función, debemos utilizar los paréntesis (). Entonces nuestra instrucción quedará como: callback().

**Nota:**si bien utilizar el término “callback” para llamar a este parámetro es una convención muy utilizada, técnicamente se puede utilizar cualquier nombre.

**PASO 2.**

Ahora, pasemos a la ejecución: ejecutamos la función saludar y enviamos como parámetro bienvenida, sin comillas simples ni dobles porque es una función. Nuestra función saludar toma ese dato, lo ejecuta y muestra el mensaje.

saludar(bienvenida); // Imprime “Hola mundo”

Si reemplazamos callback por el valor que estamos enviando quedaría así:

bienvenida()

Si bien saludar recibe como parámetro una función, a la hora de llamarla puede escribir una función directamente cómo parámetro. En internet encontrarás cientos de ejemplos que la trabajan así.

saludar(function (){

console.log("Saludo intermedio")

})

Como puedes ver, estamos ejecutando un nuevo mensaje sin la necesidad de tener la función previamente definida.

Aquí puedes observar el código completo y en funcionamiento:

[Codepen: Ejemplo callback](https://codepen.io/acamica_dwfs/pen/JjYvvzZ" \t "_blank)

**Promesas**

Una promesa será una acción que podrá resolverse en un momento dado en el futuro. Nuestra promesa en algún momento generará un resultado, ya sea positivo o negativo, y nosotros seremos capaces de capturar esa respuesta para aplicar la acción que creamos conveniente.

El concepto de promesas está muy asociado a la asincronía. Esto se debe a que JavaScript es un lenguaje asincrónico, es decir que ejecuta instrucciones y no espera su resultado, por lo que cuando necesitamos consumir algo externo no sabemos cuánto puede llegar a demorar. De esta manera, no importa cuánto tiempo tarde la instrucción en finalizar, siempre va a haber algo que esté esperando la finalización de la promesa para ejecutarse.

Las promesas pueden pasar por tres estados:

1. Pendiente mientras se está ejecutando y todavía no hay un resultado.
2. Resuelta cuando la promesa se ejecutó exitosamente.
3. Rechazada si la promesa falló y finaliza sin el resultado deseado.

Antes de entrar en las líneas de código, llevemos el trabajo de promesas a un ejemplo de la vida cotidiana.

Esta noche quiero cenar Risotto Italiano, pero como no tengo los ingredientes necesarios debo ir al supermercado a conseguirlos.

cenar\_risotto = creo una promesa {

//ejecutó mis instrucciones

//ir al supermercado

//comprar ingredientes

//pagar en la caja

//volver a casa

//cocinar

if(todos los ingredientes? && pude preparar el risotto?){

promesa exitosa

}else{

promesa rechazada

}

}

Hasta aquí solo definí la promesa y qué es lo que debe hacer, pero como el tiempo en el supermercado, no sabemos a ciencia cierta cuánto tiempo puede demorar nuestra aplicación en ir realizando otras tareas. Aquí es donde empieza a tener más sentido el concepto de asincronía.

En el momento que la promesa se encuentra ejecutando instrucciones (ir al supermercado, buscar los ingredientes, etc.) no sabemos cuánto tiempo puede demorar en tener una resolución, por esta razón en este momento la promesa se encuentra en el estado pendiente.

Una vez que la promesa finalice, ya sea de manera exitosa o rechazada, contamos con 2 métodos para capturarlas y ejecutar las acciones que crea conveniente.

cenar\_risotto.exitosa( callback () { estoy cenando risotto } )

cenar\_risotto.rechazada( callback () { debo preparar otra cena } )

**¿Cómo trabajar con promesas?**

Mediante la clase Promise podemos construir un objeto con los métodos necesarios para ejecutar la resolución de las promesas. El constructor de la clase recibe un parámetro de tipo función, la cual será la encargada de contener las instrucciones para la lógica de tu promesa y definirá si es aprobada o rechazada:

let mi\_promesa = new Promise((resolve, reject) => {

const number = Math.floor(Math.random() \* 5);

if(number > 0){

resolve((number % 2)? "es impar" : "es par");

}else{

reject("es cero")

}

});

Dentro de la función que envió al constructor, se ejecuta tu serie de sentencias. En este caso: tomamos un número al azar entre 0 y 5:

* si el número tomado al azar es mayor a 0 finalizamos la promesa de manera exitosa, calculando si el número es par o impar.
* en cambio, si el número seleccionado al azar es cero, finalizamos la promesa rechazándola, enviando el mensaje de que el número es cero.

Como no sabemos cuánto tiempo puede demorar toda esta serie de instrucciones, cuando capturamos la resolución de las promesas tenemos los métodos then y catch.

* **Obtener el resultado de una promesa de forma exitosa.** Con el método then se puede registrar una función callback que será ejecutada cuando la promesa se resuelva y devuelva un resultado. Al utilizar then podemos indicar dos funciones, la primera se ejecutará en el caso en el que la promesa sea resuelta exitosamente, y la segunda cuando falle.
* **Capturar promesas rechazadas.**Con el método .catch podemos capturar las promesas rechazadas y definir nuestro bloque de código para trabajar con el error.

mi\_promesa

.then(number => console.log(number))

.catch(error => console.error(error));