**Funciones Asíncronas**

**Comunicación y funcionalidades de los navegadores**

\*“La Red o Internet es considerada un gran aliado a la hora de potenciar la creatividad. Además utilizamos la Red para acceder a la información, comunicar, almacenar datos y divertirnos.” Anthony Peter Buzan — Psicólogo y escritor, inventor del Mind Mapping\*

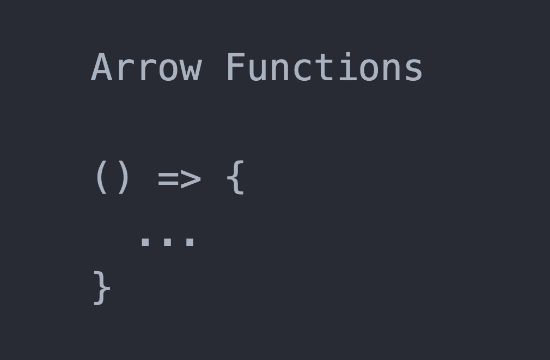


Photo by [Aditya Chinchure](https://unsplash.com/@adityachinchure?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/collections/589840/workstation?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText" \t "_blank)

El surgimiento de la computadora e Internet cambiaron la forma en la que nos comunicamos —y muchas otras cosas más— y todo se volvió más veloz. Actualmente contamos con una gran variedad y cantidad de canales para poder conectarnos entre nosotros/as y con diferentes lenguajes para hacerlo.

Es aquí donde vuelve a tomar importancia el backend en JavaScript para comprender el funcionamiento de lo que se encuentra detrás de una computadora y, en definitiva, lo que permite esa versatilidad y velocidad que todos experimentamos como usuarios/as. En esta toolbox vamos a profundizar en el trabajo con asincronía dando pie a escribir nuevas funciones para detener el flujo de tu aplicación en el momento que quieras

**Diferentes formas de escribir**



Para lo que viene es muy importante comprender que existe otra forma de escribir funciones más resumida, lo que ayuda a reducir la cantidad de líneas de código fuente que escribes. Estas son las funciones flecha, o como se denominan en inglés arrow function.

Su estructura es la misma, se define un nombre, parámetros, lo que deseamos que realice y retorna, o no, un valor de acuerdo a lo que necesitemos.

Veamos un ejemplo de una función tradicional para sumar 2 números enteros.

function sumar (n1, n2){

const total = n1 + n2;

return total;

}

Esta misma función escriba de manera “arrow” sería así

const sumar = (n1, n2) => {

const total = n1 + n2;

return total;

}

A simple vista es muy parecida, pero si observan bien a la función tipo arrow se le define un scope, en este caso usamos const, por lo tanto puedo elegir dónde quiero disponibilizar mi función.

En las arrow function no es necesario abrir las llaves para definir el body de nuestra función cuando solo tenemos una instrucción. Tampoco es necesario utilizar return, ya que la función devuelve automáticamente el valor de la única instrucción que realicemos, entonces nuestra función de sumar podría quedar en una sola línea de la siguiente manera:

const sumar = (n1, n2) => n1 + n2

De cualquier manera que escribas tus funciones, y en todas las maneras que vimos hasta ahora, la manera de llamarlas es la misma:

sumar(4, 5)

También puedes encontrarte con funciones que reciban un solo parámetro. En este caso no es necesario abrir paréntesis, por ejemplo una función que reciba un número y retorne su cuadrado podríamos escribirla así:

const cuadrado = numero => numero \* numero

console.log(cuadrado(7)) imprime 49

¿Y, existen funciones sin parámetros? Sí, también es posible escribirlas de “modo arrow”:

var hi = () => 'Hello World';

console.log(hi()); imprime Hello World

En el [este video](https://www.youtube.com/watch?v=aIKL5tQP25Y) podrás ver de manera breve, cómo escribir las arrow functions.

**Nueva forma de trabajar las promesas**

En la toolbox anterior nos detuvimos a trabajar con promesas, este concepto te acompañará a lo largo de tu vida profesional como developer, por eso seguiremos trabajando con promesas a lo largo del programa. Tomate tu tiempo para comprender su funcionamiento, para trabajar los procesos asíncronos, y manejar flujos de datos más complejos de una forma más sencilla y práctica. A continuación, vamos a expandir lo que en programación podemos llegar a hacer con las promesas.

A medida que pasa el tiempo, JavaScript va evolucionando y en su versión de 2017 fueron incorporadas las funciones asincrónicas. Estas son más convenientes para escribir código de una manera más simple porque permiten:

* **Escribir un código basado en promesas** como si fuese sincrónico sin bloquear el hilo principal. Devuelve una promesa y puede, en su cuerpo, especificar en qué lugares queremos que trabaje de manera sincrónica.
* **Trabajar con código asíncrono complejo,**en el cual se ejecutan procesos costosos en un segundo plano sin penalizar la experiencia del usuario/a. Por ejemplo en el caso de llamadas a servidor.

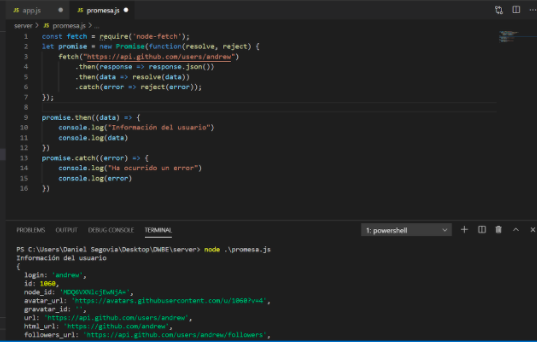
**Funciones asíncronas**

Para hacer que una función sea asincrónica, se indica la palabra clave async antes de *function*. Cuando esta función es llamada, devuelve una promesa. Si el cuerpo devuelve algo, esa promesa es resuelta. Si devuelve una excepción, la promesa es rechazada.

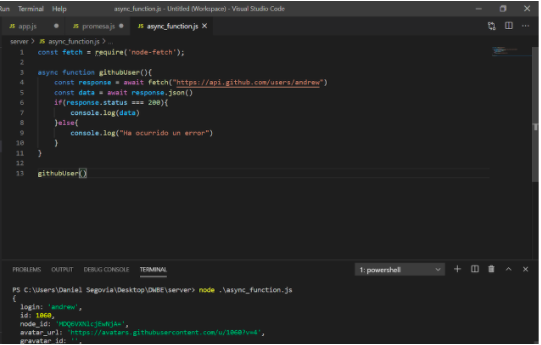
Dentro de la función async, podemos utilizar la palabra clave await antes de una expresión para esperar a una promesa antes de continuar con la ejecución de la función, es decir, detenemos la asincronía hasta que finalice la instrucción.

Esta nueva forma async/await proporciona una forma mucho más limpia, simple y legible para trabajar con promesas y  de forma síncrona. En el código se puede ver claramente:

**Promesas**



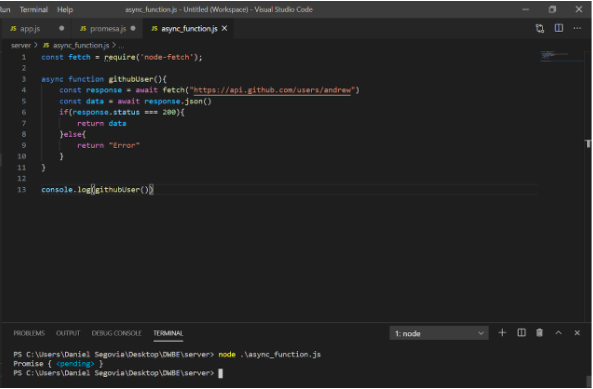
**Función asíncrona**



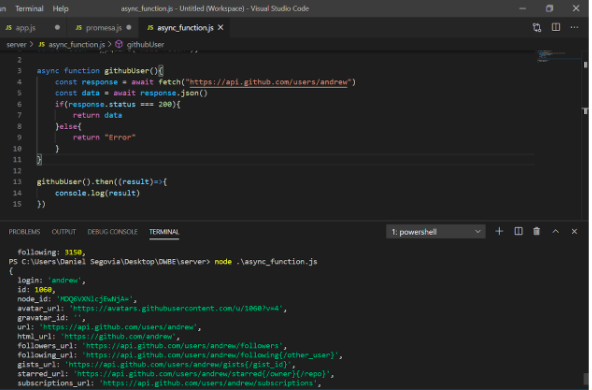
Como puedes observar, de ambas formas podemos obtener un mismo resultado: con funciones asíncronas podemos detener la ejecución donde queramos con await, mientras que con promesas debemos esperar que finalice el bloque de código entero para poder capturar con catch.

Muchas veces puede que necesites el resultado del procesamiento de una función asíncrona fuera de ella, es decir volver a tu flujo principal con el dato terminado. Lo que debes saber en este caso, es que las funciones asíncronas no devuelven datos, **siempre devuelven promesas,**por lo tanto si necesitas del dato del **return** deberás hacer de él con .then o .catch.

Observa cómo se retorna la variable data y al intentar imprimirla solo se ve una promesa:



Ahora sí, capturemos el resultado de esa promesa para obtener lo que más nos importa, el resultado (valga la redundancia).



Hasta aquí, hemos visto cóm trabajar con arrow functions. Ten en cuenta que también puedes escribir este tipo de funciones en “modo arrow”:

let hi = async () => console.log(“Hello World”)

**Cierre**

Quedó claro que el navegador es nuestro mejor aliado en JavaScript para la comunicación mediante protocolos. También vimos cómo se comunican las aplicaciones a través de las APIs y sabemos cómo utilizarlas para el intercambio de datos. ¡Prepárate para comenzar a poner en práctica todo esto en la próxima meeting!