# **Qué es el asincronismo**

* **Thread**: Permite realizar programación multihilos en este entorno. En realidad, simula la creación y ejecución de hilos, pero para el desarrollador es lo mismo. Esto simplifica muchísimo la creación de aplicaciones Javascript.
* **Bloqueante**: Una llamada u operación bloqueante no devuelve el control a la aplicación hasta que se ha completado. Por tanto, el thread queda bloqueado en estado de espera.
* **No bloqueante**: Una tarea no bloqueante se devuelve inmediatamente con independencia del resultado. Si se completó, devuelve los datos. Si no, un error.
* **Síncrono**: Las tareas se ejecutan de forma secuencial, se debe esperar a que se complete para continuar con la siguiente tarea.
* **Asíncrono**: Las tareas pueden ser realizadas más tarde, lo que hace posible que una respuesta sea procesada en diferido. La finalización de la operación I/O (entrada/salida) 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 paralelo.
* **Paralelismo**: Es la ejecución simultánea de dos o más tareas. Algunas tareas se pueden dividir en partes más pequeñas que pueden ser resueltas simultáneamente.
* **Concurrencia**: La concurrencia es la capacidad de un algoritmo o programa para ejecutar más de una tarea a la vez. El concepto es similar al procesamiento paralelo, pero con la posibilidad de que muchos trabajos independientes hagan diferentes cosas a la vez en lugar de ejecutar el mismo trabajo.
  + **Eventloop** o **Loop de eventos**: El bucle de eventos es un patrón de diseño que espera y distribuye eventos o mensajes en un programa.

## **Formas de manejar la asincronía en JavaScript**

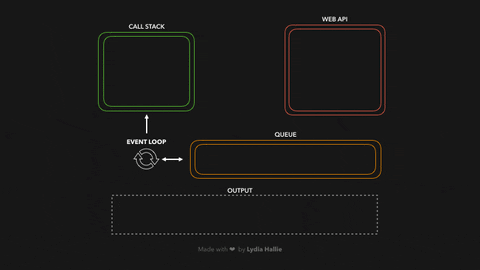
* **Callbacks**: Una función que se pasa como argumento de otra función y que será invocada.
* **Promesas**: (implementado en ES6) Una promesa es una función no-bloqueante y asíncrona la cual puede retornar un valor ahora, en el futuro o nunca.
* **Async** / **Await**: (implementado en ES2017) Permite estructurar una función asincrónica sin bloqueo de una manera similar a una función sincrónica ordinaria.

En JavaScript casi todas las operaciones de I/O (Entrada y Salida) no se bloquean. A esto se le conoce como **asincronismo**. Lo único que no es procesado antes de que termine la operación son los **callbacks**, ya que éstos están amarrados a una operación y esperan a que sea finalizada para poder ejecutarse.

El asincronismo es una manera de aprovechar el tiempo y los recursos de la aplicación, ejecutando tareas y procesos mientras otros son resueltos en background (como la llegada de la información de una API), para posteriormente continuar con las tareas que requerían esa información que no tenías de manera instantánea.

# **Event Loop**

* **Event Loop**: El buble de eventos de un patrón de diseño que espera y distribuye eventos o mensajes de un programa.
* **Memory Heap**: Los objetos son asignados a un montículo (espacio grande en la memoria no organizado).
* **Call Stack** (pila): Apila de forma organizada las instrucciones de nuestro programa.
* **Task Queue**: Cola de tareas, que se maneja la concurrencia, se agregan las tareas que una están listas para pasar al Stack (Pila). El stack debe de estar vacío.
* **MicroTask Queue**: Las promesas tienen otra forma de ejecutarse y de una prioridad superior.
* **Web APIs**: JavaScript del lado del cliente: setTimeout XMLHttpRequest, File Reader, dom. Node: fs, https.
* **Event Loop**: Tarea asignada para mover el Task Queue al Stack, solo si el Stack está vacío.

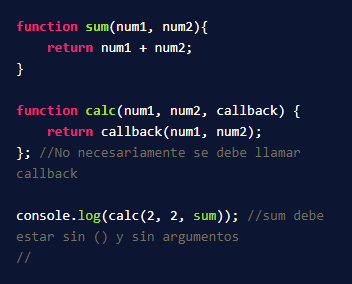


Conceptos fundamentales

* **Web APIs** JavaScript del lado del cliente: **setTimeout**, **XMLHttpRequest**, **File Reader**, **DOM**. Node: **fs**, **https**.
* **API**: El término API es una abreviatura de “Application Programming Interface” (Interfaz de programación de aplicaciones en español). Es un conjunto de rutinas que provee acceso a funciones de un determinado software.
* **Hoisting**: Sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código en tiempo de compilación.
* **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.
* **DOM**: El 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.
* **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.

# **Qué son los Callbacks**

Es una una función que se pasa como argumento de otra función y que será invocada.



# **XMLHTTPRequest**

Es un objeto de JS que permite hacer peticiones hacia servicios en la nube (URLs o APIs).

## **Estados**

* **0**: Se ha inicializado.
* **1**: Loading (cargando).
* **2**: Se ha cargado.
* **3**: Procesamiento si existe alguna descarga.
* **4**: Completado.

## **Métodos y propiedades**

* **xmlhttp.open**(): Prepara la petición para ser enviada tomando tres parámetros: **protocolo**, **url**, **asíncrono** (true).
* **xmlhttp.readyState**: Retorna el estado de la petición.
* **xmlhttp.onreadystatechange**: Un eventHandler que es llamado cuando la propiedad readyState cambia.
* **xmlhttp.status**: Retorna el estado de la respuesta de la petición. (200,400,500)
* **xmlhttp.send**(): Envía la petición.

## **Características del protocolo http**

* **Verbos**: Los verbos indican acciones que están asociadas a peticiones y recursos, es decir, sirven para la manipulación de recursos cliente/servidor. Los Verbos http son:
  + **GET** → Solicita un recurso.
  + **HEAD** → Solicita un recurso, pero sin retornar información, la estructura de esta petición es igual que get tanto en su headers como estatus. Es útil cuando vamos a utilizar API, para comprobar si lo que vamos a enviar esta correcto y puede ser procesado.
  + **POST** → Sirve para la creación de recursos en el servidor.
  + **PUT** → Actualiza por completo un recurso, reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
  + **PATCH** → Actualiza parcialmente un recurso.
  + **DELETE** → Elimina un recurso.

## **Los códigos de estados del servidor**

El código de estado (status codes) sirve para describir el estado de la petición hecha al servidor.

* 1xx: Indican que la petición fue recibida por el servidor, pero está siendo procesada por el servidor.
* 2xx: Indican que la petición fue recibida, aceptada y procesada correctamente.
* 3xx: Indican que hay que tomar acciones adicionales para completar la solicitud.
* 4xx: Indican errores del lado del cliente que hizo mal una solicitud.
* 5xx: Indican errores del servidor. Suelen aparecer cuando existe un fallo en la ejecución en el servidor.

**Los códigos más comunes a la hora de interactuar con una** **API** **son**

* 200 **OK**: Indica que todo está correcto.
* 201 **Created**: Todo está correcto cuando se hizo una solicitud POST, el recurso se creó y se guardó correctamente.
* 204 **No Content**: Indica que la solicitud se completó correctamente pero no devolvió información. Este es común cuando se hacen peticiones con el verbo DELETE.
* 400 **Bad Request**: Indica que algo está mal en la petición (no encontró algo).
* 401 **Unauthorized**: Significa que antes de hacer una solicitud al servidor nos debemos autenticar.
* 403 **Forbidden**: Indica que no tenemos acceso a ese recurso, aunque se esté autenticado.
* 404 **Not Found**: Indica que no existe el recurso que se está intentando acceder.
* 500 **Internal Server Error**: Indica que algo falló, es un error que retorna el servidor cuando la solicitud no pudo ser procesada.

# **Qué son las promesas**

Son **asíncronas**, por lo que el código continuará su ejecución normalmente y luego dirá si la promesa se resolvió o se rechazó. Por lo que varias promesas pueden llegar a entrar en ejecución al mismo tiempo.

Las promesas pueden suceder:

* Ahora.
* En el futuro.
* Nunca.

**Creación de una promesa**

Utilizamos la palabra reservada **new** seguida de la palabra **Promise** que es el constructor de la promesa. Este constructor recibe un único parámetro que es una **función**, la cual, a su vez, recibe otros dos parámetros: **resolve** y **reject**.

* El parámetro **resolve** se utiliza para cuando la promesa devuelve el valor correctamente.
* El parámetro **reject**, se usa en el que caso de que no funcione.

**Una Promesa puede estar en uno de los siguientes estados:**

* **Pendiente pending** → Una promesa inicia en este estado: no cumplida, no rechazada:

Una promesa inicialmente está pendiente.

* **Cumplida fulfilled** → Significa que la operación se completó satisfactoriamente, **.then**(response => …)

Cuando llamamos a **resolve** entonces la promesa pasa a estar resuelta.

Cuando una promesa se resuelve entonces se ejecuta la función que pasamos al método .**then**

* **Rechazada rejected** → Significa que la operación falló, .**catch(err => …)**

Si llamamos a **reject** pasa a estar rechazada (obtenemos un error que nos va a indicar la razón del rechazo).

Si la promesa es rechazada entonces se ejecuta la función que pasamos a .**catch**

# **Fetch**

Por medio de **Fetch** se puede realizar peticiones HTTP asíncronas de JavaScript (JS) con promesas. La API Fetch proporciona una interfaz JS más cómoda para acceder y manipular, **fetch**() es un método global.

Para poder usar fetch dentro de node, primero tenemos que instalarlo.

* **npm i node-fetch**

Para poder compilar desde VSC, debemos registrar el módulo en **package.json**, abrimos el archivo y al final se agrega:



# **Fetch POST**

Los verbos indican acciones, estás acciones están asociadas a peticiones y recursos. En línea general sirve para la manipulación de recursos cliente/servidor. Los códigos de estados, los códigos son valores números que tienen un valor semántico.

**Algunos Verbos http son:**

* **GET** → Solicitar recurso.
* **POST** → Creación de recursos en el servidor.
* **PUT** → Actualizar por completo un recurso.
* **DELETE** → Eliminar un recurso.

# **Funciones asíncronas**

La declaración de función **async** define una función asíncrona que devuelve un **objeto**, lo cual permite a un programa correr una función sin congelar toda la compilación.

Dada que la finalidad de las funciones **async**/**await** es simplificar el comportamiento del uso síncrono de promesas, se hace más fácil escribir promesas.

La estructura se compone por las palabras reservadas **async** y **await**:

* La palabra **async** antes de la función, hace que la función devuelva una **promesa**.
* La palabra **await** se utiliza **dentro** de las funciones **async**, lo que hace que el programa **espere** hasta que la variable (promesa) se resuelva para continuar.

# **Try and catch**

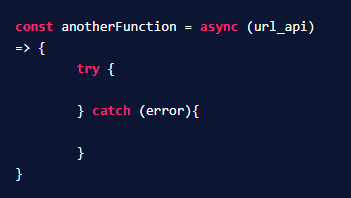
La palabra reservada **try** consiste en un bloque que contiene una o más sentencias, como hacíamos con **resolve**.

Su cuerpo está conformado por las llaves **{}** las cuales se deben utilizar siempre, incluso para un bloque de una sola sentencia.

También puede estar presente un bloque con la palabra reservada **catch**.

Un bloque **catch** es opcional (como hacíamos con **reject**) y contiene sentencias que especifican que hacer si una excepción es lanzada en el bloque **try**.

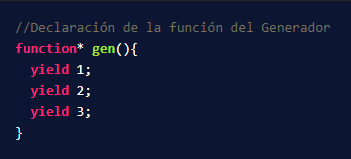
Si no se lanza ninguna excepción en el bloque **try**, el bloque **catch** se omite.



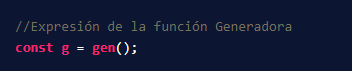
# **Generators**

Un **generador** en JavaScript consta de una función generadora que muestra un objeto **iterable Generator**. La palabra reservada **yield** se usa para **pausar** y **reanudar** una función generadora.

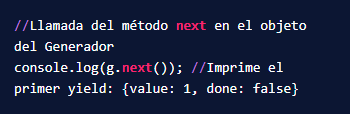
* La estructura del Generador consta con la palabra **function** seguido de un asterisco **\* : function\*** ésta es una función generadora heredada.
* El resultado que se quiere obtener se coloca al lado derecho de **yield**, puede ser de cualquier tipo (string, numérico, objetos, etc) y se puede tener tantos **yield** que se desee.



Para poder iterar con el generador, se puede inicializar un valor con la función generadora:



Entre las propiedades del iterador está **next**():



**next**() permite acceder a la función del generador y obtener con **yield** dos valores: **value** y el estado de **done**, es decir, si tenemos **yield 1;** y mandamos a imprimir el resultado con **next**() obtenemos `**{value: 1, done: false}**’:

* El 1 por el valor al lado **derecho** del primer **yield**.
* Y **done** es **false** porque mientras haya otro yield por operar será falso.
* Será **true** cuando se ejecute cuatro veces **next**() y la salida mostrará {value: undefined, done: true}. Esto se debe a que ya no hay más nada que mostrar, porque se mandó a imprimir un cuarto elemento y el generador solo tiene 3 yield.