Índice

[**Instalación de Node.js** 1](#_Toc101644585)

[**Node: orígenes y filosofía** 1](#_Toc101644586)

[**EventLoop: asíncrona por diseño** 2](#_Toc101644587)

[**Monohilo: implicaciones en diseño y seguridad** 3](#_Toc101644588)

[**Herramientas para ser más felices: Nodemon y PM2** 4](#_Toc101644589)

[**Producción** 5](#_Toc101644590)

[**Aclaración** 5](#_Toc101644591)

[**Callbacks** 5](#_Toc101644592)

[**Callback Hell: refactorizar o sufrir** 6](#_Toc101644593)

[**Promesas** 6](#_Toc101644594)

[**Async/await** 8](#_Toc101644595)

[**Globals** 10](#_Toc101644596)

[**Buena práctica** 10](#_Toc101644597)

[**File system** 12](#_Toc101644598)

[**Console** 14](#_Toc101644599)

[**Algunas funciones útiles** 14](#_Toc101644600)

[**Errores (try / catch)** 14](#_Toc101644601)

[**Procesos hijo** 16](#_Toc101644602)

[**Cuando usar spawn** 16](#_Toc101644603)

[**Cuando usar exec** 16](#_Toc101644604)

[**¿Como funciona un child process?** 17](#_Toc101644605)

[**Módulos nativos en C++** 18](#_Toc101644606)

[**HTTP** 19](#_Toc101644607)

[**OS** 21](#_Toc101644608)

[**Process** 23](#_Toc101644609)

[**Gestión de paquetes: NPM y package.json** 24](#_Toc101644610)

[**Advertencia** 24](#_Toc101644611)

[**Construyendo módulos: Require e Import** 24](#_Toc101644612)

[**Exportar módulos en versiones 12 y anteriores de nodejs** 26](#_Toc101644613)

[**¿Qué son los módulos ES?** 27](#_Toc101644614)

[**Formas de exportar código en ES6** 27](#_Toc101644615)

[**Formas de importar código en ES6** 28](#_Toc101644616)

[**Import estático vs dinámico** 28](#_Toc101644617)

[**Diferencias entre import estáticos y dinámicos** 29](#_Toc101644618)

[**Módulos útiles** 30](#_Toc101644619)

[**Datos almacenados vs en memoria** 33](#_Toc101644620)

[**Características de los discos y memorias** 33](#_Toc101644621)

[**Diferencia entre Buffer y Stream** 33](#_Toc101644622)

[**Buffers** 33](#_Toc101644623)

[**Creación de un buffer básico** 34](#_Toc101644624)

[**Otras formas de crear un buffer** 34](#_Toc101644625)

[**Streams** 36](#_Toc101644626)

[**Tipo de Streams** 36](#_Toc101644627)

[**Buffer** 36](#_Toc101644628)

[**Benchmarking (console time y timeEnd)** 37](#_Toc101644629)

[**¿Qué es más rápido?** 37](#_Toc101644630)

[**Debugger** 38](#_Toc101644631)

[**Error First Callbacks** 40](#_Toc101644632)

[**Advertencia** 41](#_Toc101644633)

[**Scraping** 42](#_Toc101644634)

# **Instalación de Node.js**

Si vamos a trabajar con **Node.js**, lo primero que tienes que hacer, es instalarlo en tu máquina. Ya sea con **Windows**, **Linux** o **Mac**, si vas a la web de [Node.js](https://nodejs.org/en/), la web detectará tu sistema operativo, y te ofrecerá un paquete con el que instalarlo.

Simplemente pulsa en el botón verde de la versión que quieras (lo recomendable es siempre usar las **versiones LTS**) y completar el proceso de instalación. Dependiendo del sistema operativo, te hará más o menos preguntas, pero con las opciones por defecto se instalará bien.

Una vez lo hayas instalado, para comprobar que todo funciona correctamente, abre una terminal (en windows, CMD o PowerShell valen perfectamente) y escribe:

**node -v**

Ese comando te devolverá la versión de Node.js que se ha instalado.

También nos habrá instalado NPM, el gestor de paquetes. Para asegurarte de que está instalado, puedes ejecutar:

**npm -v**

Y te devolverá la versión de NPM que hay instalada.

# **Node: orígenes y filosofía**

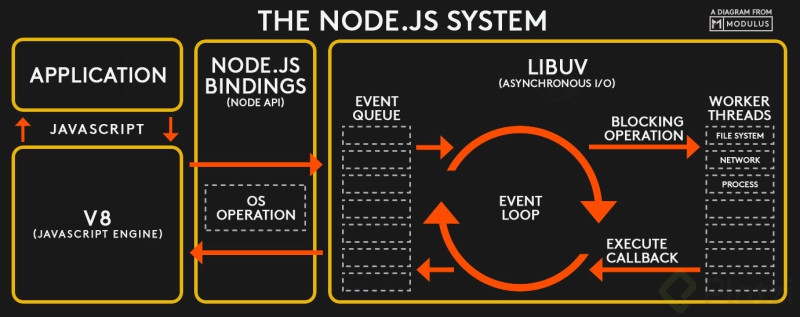
**NodeJS** es un entorno de ejecución de JavaScript **fuera** del navegador. Se crea en 2009, **orientado a servidores**. Es muy importante que esté fuera del navegador debido a que ya no es necesario un navegador web para ejecutar código JavaScript.

Características principales de JavaScript:

* **Concurrencia**: Es monohilo, con entradas y salidas asíncronas.
* **Motor V8**: Creado por Google en 2008 para Chrome. Escrito en C++. Convierte JS en código máquina en lugar de interpretarlo en tiempo real.
* **Todo funciona en base a Módulos**, que son piezas de código muy pequeñas que modularizan nuestros sistemas y ayudan a entender mejor el código.
* **Orientación a Eventos**, existe un bucle de eventos que se ejecuta constantemente. Lo que nos permite programar de forma reactiva, esto quiere decir, que podemos programar con la lógica de “**Cuando sucede algo, se ejecuta esta parte de mi código y eso a su vez dispara otra parte**”.

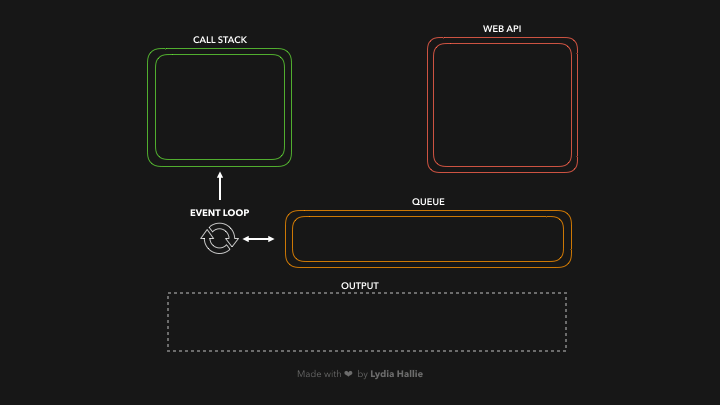
# **EventLoop: asíncrona por diseño**

* **Event Queue**: Contiene todos los eventos que se generan por nuestro código (Funciones, peticiones, etc.), estos eventos quedan en una cola que van pasando uno a uno al Event Loop.
* **Event Loop**: Es un bugle infinito que se encarga de resolver los eventos ultra rápidos que llegan desde el Event Queue. En caso de no poder resolverse rápido, envía el evento al Thread Pool.
* **Thread Pool**: Se encarga de gestionar los eventos de forma asíncrona. Una vez terminado lo devuelve al Event Loop. El Event Loop vera si lo pasa a Event Queue o no.



# **Monohilo: implicaciones en diseño y seguridad**

El hecho de que sea monohilo lo hace delicado en el sentido de que puede ejecutarse algo que corte el código y detenga el programa, como la ausencia de sintaxis o una variable pendiente por definir.



# **Herramientas para ser más felices: Nodemon y PM2**

[**Nodemon**](https://nodemon.io/). Demons en linux, puedes tener procesos que ves ejecutándose.

**Nodemon** + **archivo** al que quiero acceder detecta cambios, y ejecuta automáticamente el código.

**sudo npm install -g nodemon**

## **Producción**

**sudo npm install -g pm2**

[**PM2**](https://pm2.keymetrics.io/) Es un demonio administrador de procesos que me puede ayudar a administrar y mantener mi aplicación 24/7.

* Voy a poner monitorizar el código para saber si algo se rompe.
* Me permite ver dashboard de mi código, puedo ver que está corriendo.
* Puedo ver el rendimiento de mi cpu
* Con: pm2 stop + id —> me detiene el proceso que está en ejecución con ese ID.

## **Aclaración**

* **Nodemon**: Para usar en desarrollo
* **PM2**: Para usar en producción.

# **Callbacks**

Una funcion **callback** es una funcion que es **pasada** como **argumento** a **otra funcion**, para **ser** **llamada** (called back) en **otro momento**.

La funcion que recibe como argumento otras funciones es denominada **funcion de orden superior** (higher-order function), esta contiene la lógica correspondiente para ejecutar adecuadamente la funcion callback.



* **setTimeout** es una higher-order function.
* **console.log** es una callback function.

# **Callback Hell: refactorizar o sufrir**

Los **callback Hell** se dan cuando empiezo a pasar una función como parámetro que a su vez llama a otra función como parámetro, y así hasta **n**.

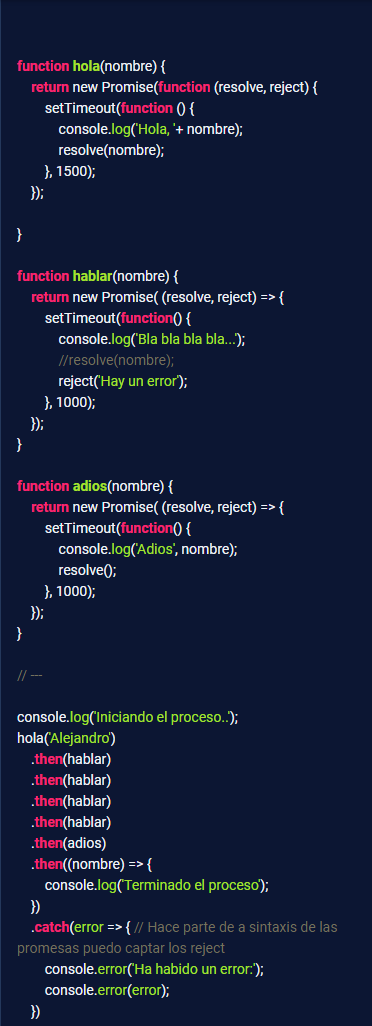
Una estrategia para trabajar con estas estructuras lógicas tan monolíticas es usar **estructuras de control** y **funciones recursivas**.

# **Promesas**

Las promesas vienen de los callbacks, pero las promesas lo que hacen es dar un estado.

Las promesas son una “clase” global que podemos llamar de donde sea, nuestras funciones devuelvan promesas **Promise()**, la diferencia entre promises y callbacks es la capacidad de anidar promesas. Formando una cadena de promesas.

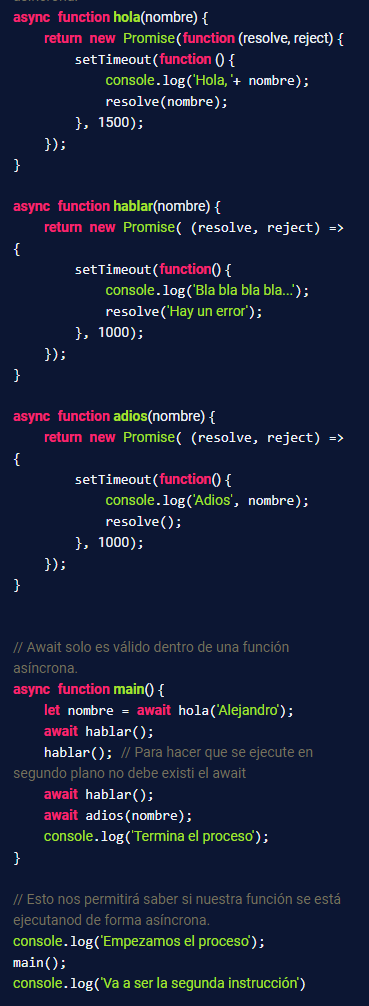
Es muy útil para visualizar código asíncrono de manera síncrona.



# **Async/await**

Async y Await nos permite definir una función de forma explícita como asíncrona y esperar a que la función termine.

**Esto no bloqueará el hilo principal**, estará esperando a que se **resuelva** con el **event loop**.



# **Globals**

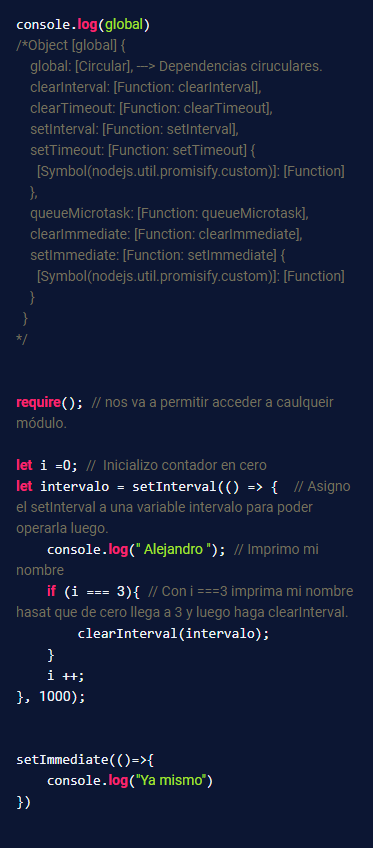
Node funciona a base de **módulos**, los módulos son el código que permite a Node **tener ciertas funcionalidades**.

Para que Node funcione correctamente siempre ha de tener sus módulos **globales**, son aquellos módulos que nos permiten usar la mayoría de funcionalidades básicas y complejas que conocemos de Node, como **setTimeout**, **setInerval**, etc.

Estos módulos los podemos usar sin necesidad de importarlos explícitamente en nuestro código, pueden ser usados en cualquier archivo de Node.

## **Buena práctica**

Si no tengo que usar variables globales, por favor, no usarlas, esto es un foco de problemas.

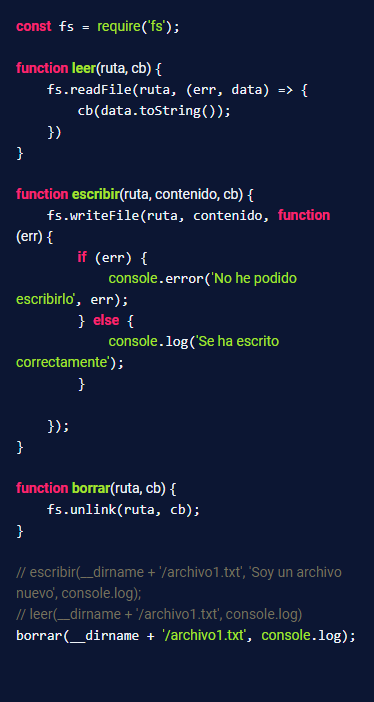


# **File system**

El **file system** provee una API para interactuar con el sistema de archivos cerca del estándar **POSIX**.

**POSIX** es el estándar para interfaces de comando y shell, las siglas significan: “**Interfaz de sistema operativo portátil**” la **X** de POSIX es por **UNIX**.

El **file system** nos permite acceder archivo del sistema, **leer**, **modificar** y **escribirlos**, es muy útil para precompiladores, para lo que requiera hacer **grabados de disco**, o **bases de datos**. Todo lo que hagamos con módulos por **buenas prácticas son asíncronos**, pero tienen una version **síncrona** no recomendada pues podría **bloquear** el event loop con más facilidad.



# **Console**

Con console podemos imprimir todo tipo de valores por nuestra terminal.

## **Algunas funciones útiles**

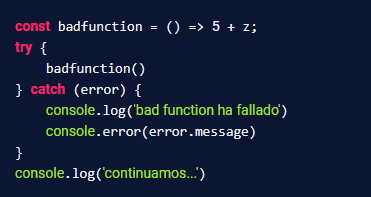
* **console.log**: recibe cualquier tipo y lo muestra en el consola.
* **console.info**: es equivalente a log pero es usado para informar.
* **console.error**: es equivalente a log pero es usado para errores.
* **console.warn**: es equivalente a log pero es usado para warning.
* **console.table**: muestra una tabla a partir de un objeto.
* **console.count**: inicia un contador autoincremental.
* **console.countReset**: reinicia el contador a 0.
* **console.time**: inicia un cronometro en ms.
* **console.timeEnd**: Finaliza el cronometro.
* **console.group**: permite agrupar errores mediante indentación.
* **console.groupEnd**: finaliza la agrupación.
* **console.clear**: Limpia la consola.

Más info [aquí](https://nodejs.org/dist/latest-v16.x/docs/api/console.html).

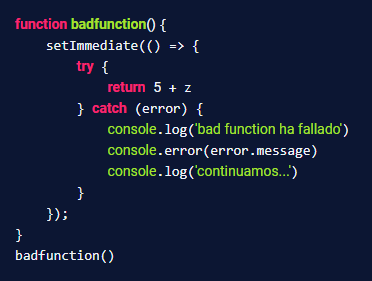
# **Errores (try / catch)**

**Siempre que sea posible debemos capturar todos los errores que se puedan generar en nuestros hilos**.

Cuando se genera un error, node propaga el **error hacia arriba**, es decir, sí tenemos una función **dentro** de **otra** y el error se produce en la que está en el **interior**, node mostrará el error de igual forma, si este esté es captado, si el error no se captura node se detiene. **Try/Catch** nos permite capturas los errores.



Si deseamos manejar errores asíncronos.



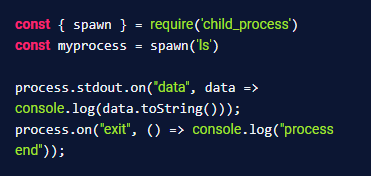
# **Procesos hijo**

El módulo de procesos secundarios de Node.js (**child\_process**) tiene dos funciones **spawn** y **exec**, mediante las cuales podemos **iniciar un proceso secundario** para ejecutar otros programas en el sistema.

La diferencia más significativa entre **child\_process.spawn** y **child\_process.exec** está en lo que **spawn** devuelve un **stream** y **exec** devuelve un **buffer**.

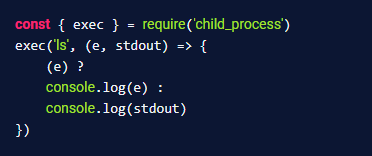
## **Cuando usar spawn**

* Usa **spawn** cuando quieras que el proceso hijo devuelva datos binarios enormes a Node.
* Cuando quieras recibir datos desde que el proceso arranca.



## **Cuando usar exec**

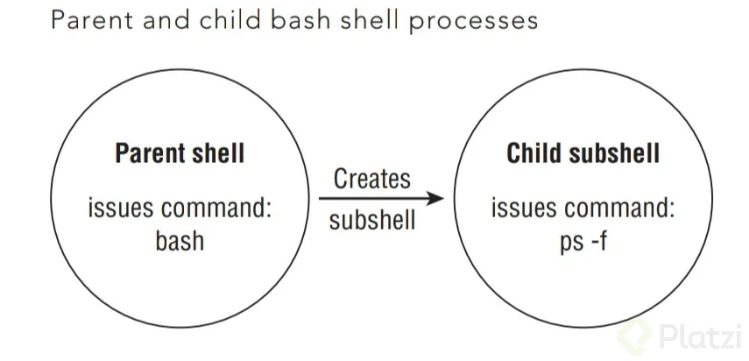
* Usa **exec** cuando quieras que el proceso hijo devuelva mensajes de estado simples.
* Cuando solo quieras recibir datos al final de la ejecución.



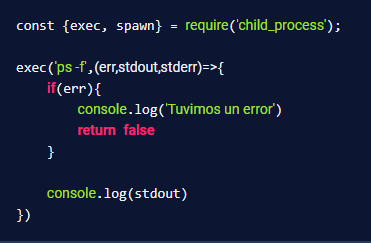
## **¿Como funciona un child process?**

Un proceso es la ejecución de algo (programa, commando, etc) que terminará en un tiempo finito.

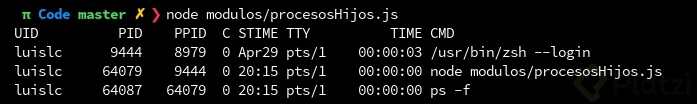
Cuando se ejecuta un programa estos corren sobre la **terminal** y cuando necesitan crear un nuevo proceso crean una nueva terminal (**Parent shell**).



Entonces al utilizar esto creamos un **Chils** **shell** que ejecutará cualquier instrucción que le pasemos. Para corroborar esto podemos ejecutar.



Y nos mostrará algo así donde PID es **Proces ID** se liga con **Process Parent ID**.

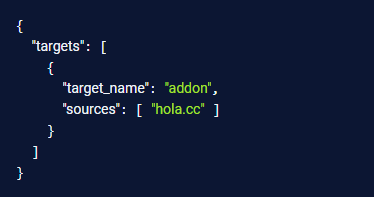


# **Módulos nativos en C++**

JavaScript permite hacer uso de módulos nativos de **c++**. Para lograr esto debemos instalar **sudo npm i -g node-gyp**, este módulo de **npm** nos permite compilar módulos nativos de c++ en node.

Luego debemos tener listo nuestro archivo de código fuente en c++ junto a otro archivo **.gyp**, que nos ayudara hacer la compilación a JavaScript.

En este archivo **.gyp** le indicamos que va compilar, como se va llamar el archivo resultante y de donde va a tomar la info a convertir, todo esto lo dejamos como un **json**.



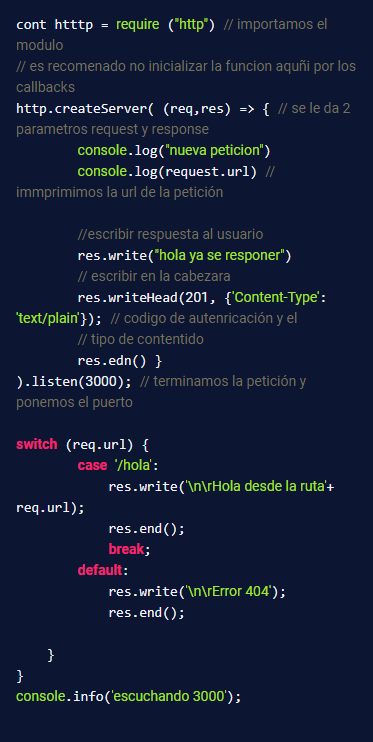
Luego le decimos a node que configure este módulo, con el comando **node-gyp configure**, como resultado tendremos en un directorio nuevo donde se encontraran diferentes archivos de código nativo, para finalizar con **node-gyp build** creamos nuestro modulo y estará listo para ser usado.

# **HTTP**

Node nos ofrece el módulo HTTP el cual nos permite principalmente crear un servidor en nuestro computador.

En este módulo encontraremos todo lo necesario que necesitamos para crear un sistema de rutas, que responderá cada ruta, los header que podrá mandar, etc.

Uno de los métodos principales de este módulo es **createServer**, el cual nos permitirá abrir un puerto para crear el servidor.

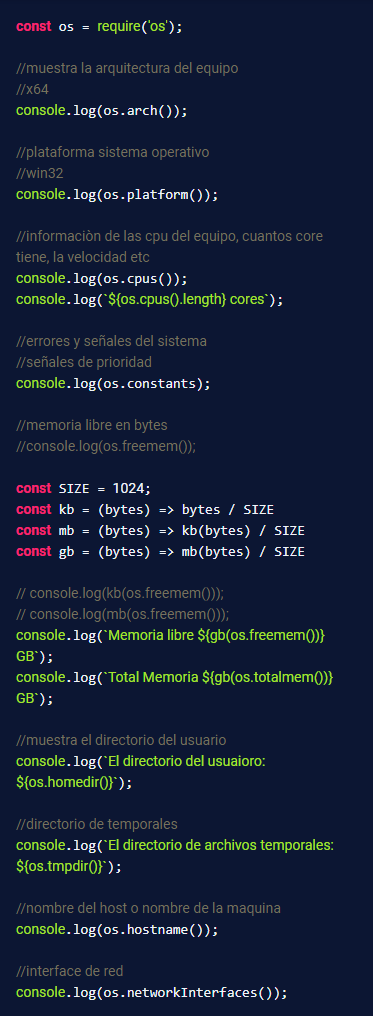


# **OS**

El módulo de **OS**, **Operative System**, nos permite ejecutar acciones de más bajo nivel en nuestro sistema, permitiéndonos conocer una gran variedad de detalles del mismo.

Como la memoria disponible que tiene, el total de la memoria, la interfaz de red, etc.

Esto nos será de gran ayuda a la hora de ejecutar o crear proyectos que necesiten información de una máquina para ejecutar una operación.



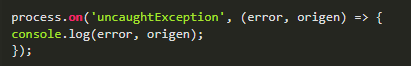
# **Process**

El objecto process es una instancia de EventEmitter, podemos suscribirnos a él para escuchar eventos de node.

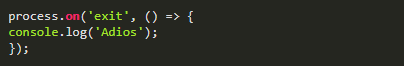
Podemos hacer require para obtener process.



* **beforeExit**: Es para enviar algo antes que pare un proceso.
* **UncaughtException**: Permite capturar cualquier error que no fue capturado previamente. Esto evita que Node cierre todos los hijos al encontrar un error no manejado.



* **exit**: Se ejecuta cuando node detiene el eventloop y cierra su proceso principal.



* **uncaughtRejection**: Permite capturar cualquier error de promesas que se han rechazado.

# **Gestión de paquetes: NPM y package.json**

**npm** (**N**ode **P**ackage **M**anager) **es un administrador de paquetes** que permiten ejecutar funciones ya realizadas, validadas y de esta manera acelerar y asegurar la calidad de nuestro proceso de desarrollo.

Podemos buscar módulos para casi todo en:

* [npmjs](https://www.npmjs.com/)

## **Advertencia**

Debemos tener cuidado a la hora de elegir un paquete, pues vamos a depender de éste, así que es mejor que esta dependencia tenga **buen soporte**, **buenas actualizaciones** y ya **haya sido ampliamente testeado** por muchos usuarios.

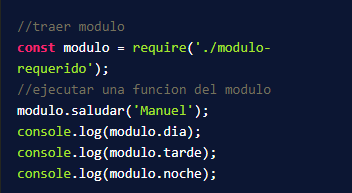
# **Construyendo módulos: Require e Import**

Cuando tengamos un proyecto robusto es posible que tengamos que trabajar con módulos creados por nosotros mismos.

Cuando hablamos de construir módulos es de modularizar nuestro código, es decir, tenemos un archivo que al ser llamado ejecuta ciertas funciones. Por ej:

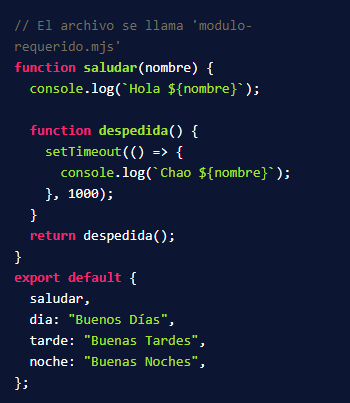


Podemos requerir este archivo de la siguiente manera.

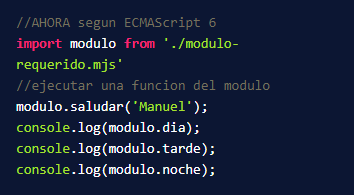


Para **ECMAScript** **6** la forma de **exportar** y **requerir** módulos ha cambiado, pero es una forma experimental en las versiones anteriores a la 12 de nodejs.

## **Exportar módulos en versiones 12 y anteriores de nodejs**



**index.mjs** es donde se importa el módulo anterior.



Para ejecutar un modulo experimental se debe utilizar el comando **node --experimental-module [name\_file.js]**, de esta forma podremos ejecutar nuestro código.

## **¿Qué son los módulos ES?**

En ECMAScript 2015 se introduce una característica nativa denominada Módulos **ES** o **ESM**, que permite la **importación** y **exportación** de datos entre diferentes ficheros Javascript. Esto nos permite trabajar de forma más flexible en nuestro código Javascript.

Para trabajar con módulos tenemos las siguientes palabras clave.

|  |  |
| --- | --- |
| Declaración | Descripción |
| export | Exporta datos (variables, funciones, clases...) del fichero actual hacia otros que los importen. |
| import | Importa datos (variables, funciones, clases...) desde otro fichero **.js** al actual. |

## **Formas de exportar código en ES6**

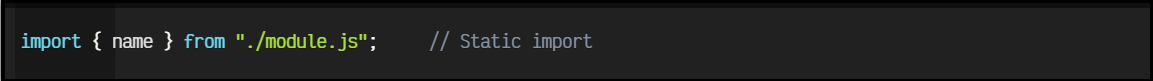
|  |  |
| --- | --- |
| Forma | Descripción |
| export ... | Declara un elemento o dato, añadiéndolo además al módulo de exportación. |
| export { name } | Añade el elemento **name** al módulo de exportación. |
| export { n1, n2, n3... } | Añade los elementos indicados (n1, n2, n3 ...) al módulo de exportación. |
| export \* from './file.js' | Añade todos los elementos del módulo de **file.js** al módulo de exportación. |
| export default ... | Declara un elemento y lo añade como módulo de exportación por defecto. |

## **Formas de importar código en ES6**

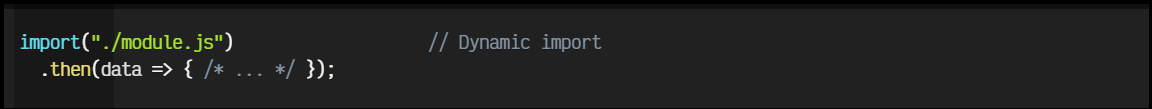
|  |  |
| --- | --- |
| Forma | Descripción |
| import nombre from './file.js' | Importa el elemento por defecto de **file.js** en **nombre**. |
| import { nombre } from './file.js' | Importa el elemento **nombre** de **file.js**. |
| import { n1, n2... } from './file.js' | Importa los elementos indicados desde **file.js**. |
| import \* as name from './file.js' | Importa todos los elementos de **file.js** en el objeto **name**. |
| import './file.js' | No importa elementos, sólo ejecuta el código de **file.js**. |

## **Import estático vs dinámico**

La palabra clave **import** que es lo que se conoce como un **import estático**, una forma de importar módulos de ficheros externos. Se suelen colocar en la parte superior del fichero Javascript y son algo similar a lo siguiente:



Sin embargo, existe otro tipo de importación en Javascript, popularmente conocida como **import dinámico** (dynamic import), que tiene el siguiente aspecto, ligeramente diferente al anterior:



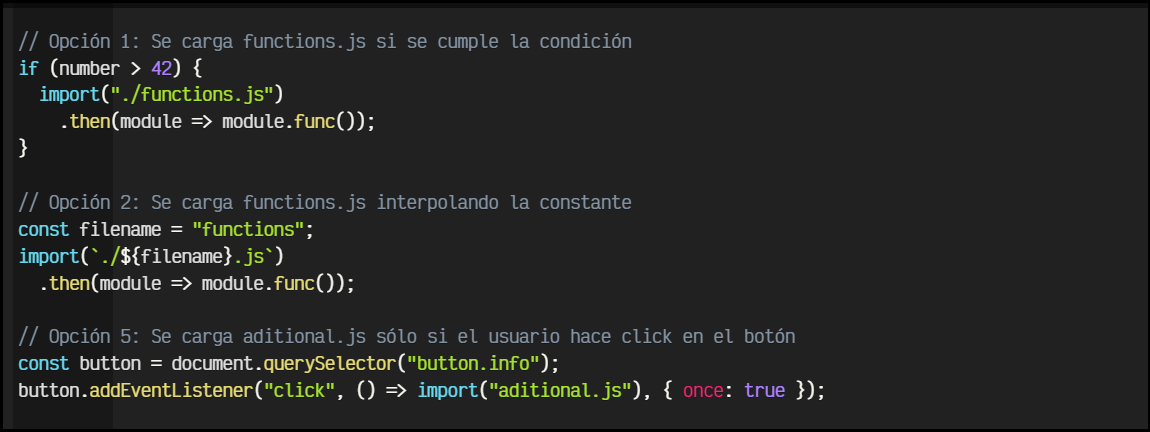
En este segundo caso, el **import()** tiene unos paréntesis que lo diferencian del anterior, y por la existencia del **.then()** sabemos que nos devuelve una **promesa**, por lo que se trata de código asíncrono.

## **Diferencias entre import estáticos y dinámicos**

Los **import estáticos** son muy útiles, pero tienen algunas desventajas si se presentan ciertos casos específicos. Los más frecuentes suelen ser los siguientes:

* Queremos importar un módulo si se cumple una determinada condición.
* Queremos importar un módulo interpolando variables o constantes.
* Queremos importar un módulo dentro de un ámbito específico.
* Queremos importar un módulo desde un script normal (sin **type="module"**).
* Queremos importar un fichero javascript (sin módulo) y ejecutarlo bajo demanda.

En cada uno de estos casos, no se puede utilizar el import estático, pero si el import dinámico:

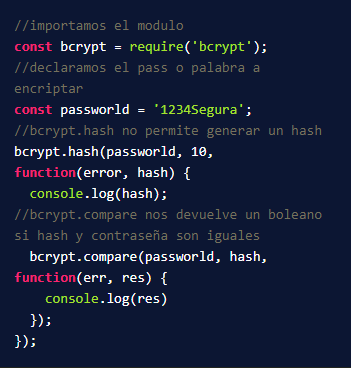


El **import dinámico** nos permite indicar entre los paréntesis del **import** el nombre del archivo Javascript. A diferencia del **import estático**, este fichero no se cargará siempre y desde el principio, sino que sólo lo hará cuando se llegue a esta parte del código, siendo posible incluirla dentro de **condicionales**, **funciones** o **lógica diversa**.

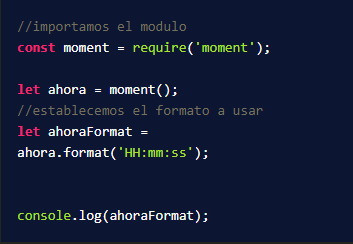
# **Módulos útiles**

Módulos que utilizaremos frecuentemente en nuestras aplicaciones como:

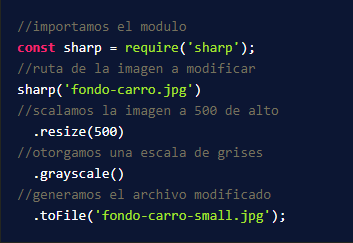
* **bycript**: Permite encriptar un string generando un hash.
* **bycript.hash**: Genera el hash del string, este método recibe por parámetros el string a encriptar, las veces que debe ejecutarse el script para generar un hash aleatorio y por ultimo una función que se encarga de capturar hash o el error que se presente.
* **bycript.compare**: Permite comprar el hash con el string a encriptado y nos devuelve un **true** o **false**. **bycript.compare** recibe por parámetros el string a encriptado, luego el hash y por ultimo una función que captura la información o el error que se presente.



* **moment**: Logramos manipular fechas de manera eficiente.
* **moment.format**: Podemos darle un formato adecuado a nuestra fecha.



* **sharp**: Permite trabajar con imágenes.



# **Datos almacenados vs en memoria**

Todo esto funciona por **tiempos**, cuando vas a escribir en **memoria** todo funciona **súper rápido**, sin embargo, cuando tienes que escribir en **disco** el proceso es **más lento**.

Especialmente los discos duros antiguos (HDD) son súper lentos. Aunque los SSD sea más rápidos, siguen siendo más lentos que la memoria RAM.

## **Características de los discos y memorias**

**Datos en memoria**

* Es más rápido, debido a la distancia más cercana entre la CPU y la memoria RAM.
* Leer y escribir archivos de manera rápida.

**Datos almacenados en discos**

* Permite almacenar mayor información, pero se gestiona de forma más lenta que la RAM.
* Los tiempos de lectura y escritura son más lentos que escribir y leer en memoria.

## **Diferencia entre Buffer y Stream**

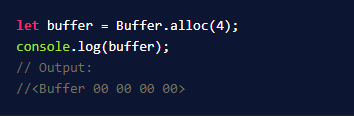
* **Buffer**: Es un montón de datos.
* **Stream**: Es un proceso en el que pasan un montón de datos.

# **Buffers**

* Un buffer es un espacio de memoria (en la memoria RAM), en el que se almacenan datos de manera temporal.
* Es la forma más cruda en la que se pueden almacenar los datos. (Se guardan en bytes y no se especifica el tipo de dato)
* En la consola, los datos se muestran en formato hexadecimal.

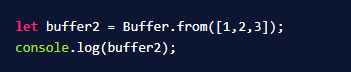
## **Creación de un buffer básico**

Para crear un buffer, con 4 espacios, podemos hacerlo con la siguiente sintaxis.

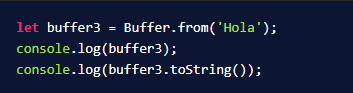


## **Otras formas de crear un buffer**

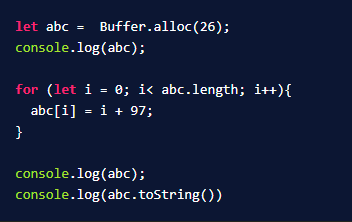
Datos en un arreglo.



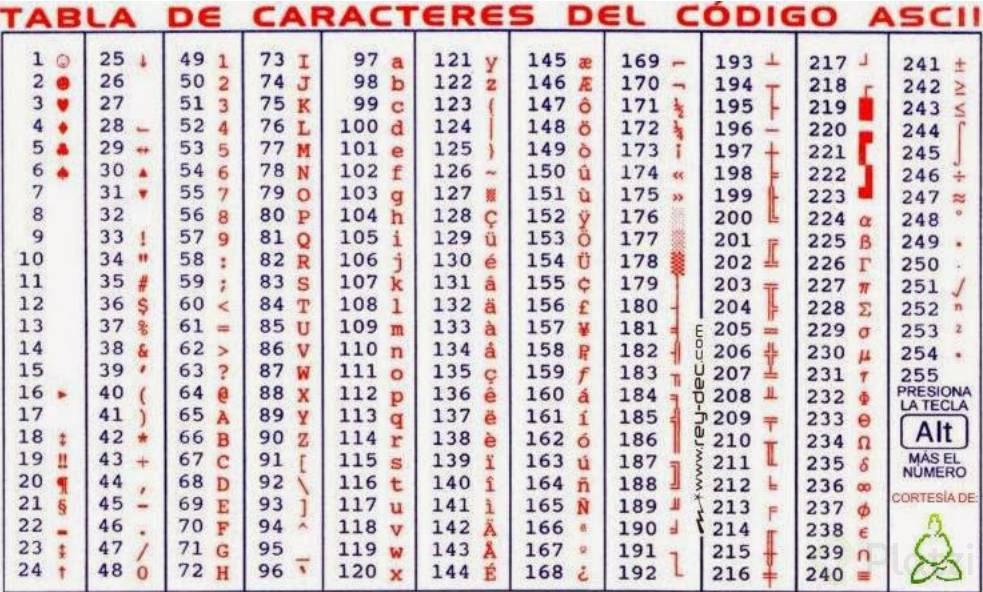
Datos de tipo string.



Guardar el abecedario en un buffer.



**¿Porque 97?**



# **Streams**

Un stream es el proceso de ir consumiendo datos al tiempo que se están recibiendo.

## **Tipo de Streams**

* **Streams de lectura**: Desde un stream de origen va soltando datos hacia otro stream de destino, el cual el de destino lee los datos enviados por el stream de origen.
* **Streams de escritura**: Tenemos un stream con un destino, le metemos datos a ese destino.
* **Streams de ambos sentidos**: Metemos y escribimos datos en ambos sentidos.

Podría decirse que un **Stream** **es el proceso de ir consumiendo datos al tiempo en que se reciben**. Por ejemplo, cuando vemos un video en YouTube estamos consumiendo datos por medio de streaming (**read stream**, porque solo podemos ver los videos, pero no editarlos) ya que lo vemos al mismo tiempo en que este se está descargando, de lo contrario habría que esperar a que se descargue el video por completo para poder verlo.

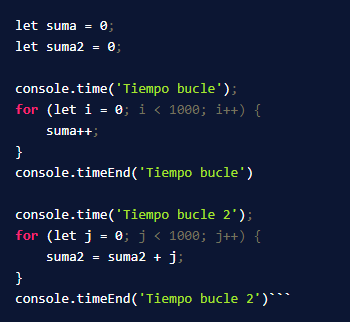
## **Buffer**

Si en el caso anterior, mientras vemos el video, fallara el internet, así sea por un segundo, la reproducción se pararía instantáneamente. Pero sabemos que en realidad no es así, el video continúa reproduciéndose por un tiempo más. Esto es gracias a la implementación de un buffer el cuál **es un espacio en memoria RAM** en donde la información proveniente del servidor llega por fragmentos (chunks), para luego ser consumido, y como ese almacenamiento de datos en el buffer se hace a bajo nivel, de forma binaria, el proceso es mucho más rápido de lo que se consume. Es por eso que cuando reproducimos un video en YouTube vemos que este se carga más rápido. (dependiendo del ancho de banda claro está.)

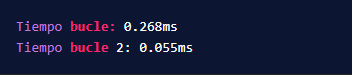
# **Benchmarking (console time y timeEnd)**

* **console.time(‘nombre’)**: Inicia un temporizador que se puede usar para rastrear cuánto tiempo dura una operación. El temporizador será identificado por el nombre dado a la consola.
* **console.timeEnd(‘nombre’)**: Para detener el temporizador y obtener el tiempo demorado durante el proceso.

## **¿Qué es más rápido?**



Resultado



El bugle con la sintaxis **suma2 = suma2 + j** resulto ser más eficiente a nivel de memoria. Pero dependiendo de que tipo de sistema se esta desarrollando, es preferible **legibilidad** antes que **eficiencia**.

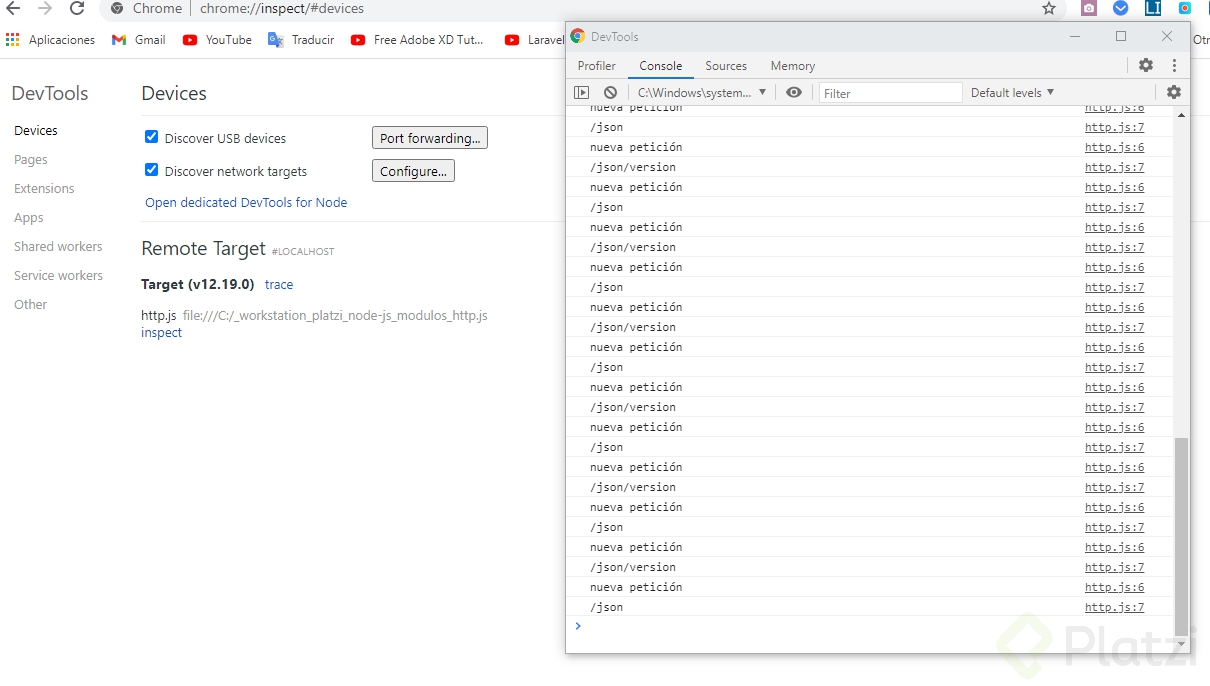
# **Debugger**

Node.js viene integrado con un modo de debug para poder conectarnos desde cualquier herramienta de inspección de código a nuestro código de node.js.

Podemos utilizar en la terminal el flag de **--inspect** con nodemon.

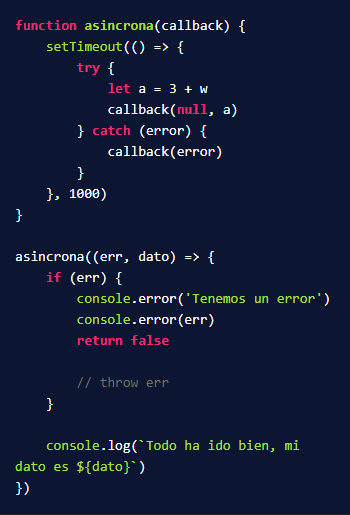


Para poder acceder a debugger de Chrome vamos a la url **chrome://inspect/#devices** y le dan a inspect en el remote target que quieres inspeccionar.



# **Error First Callbacks**

* Un patrón que se sigue siempre en cualquier lenguaje y programa de desarrollos es **Error First Callbacks**, esto quiere decir que siempre que tengamos un **callback** el primer parámetro debería ser el **error**.
* Esto se usa por la convención de que todo puede fallar.
* Otro patrón típico es tener en el callback como la última función que se pasa. Aunque depende del caso.



## **Advertencia**

**throw** en funciones asíncronas no funciona.

# **Scraping**