**Curso de Fundamentos de NodeJS.**

**Instalación de Node.js**

Si vas 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>), 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 (mi recomendación 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.

Como consejo, asegúrate de tener una buena conexión a internet cuando lo instales, para que tarde poco tiempo.

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.

Con esto, ya tenemos instalado Node.js y NPM, que es todo lo que necesitamos para empezar con nuestro curso de Fundamentos de Node.js.

# Node: orígenes y filosofía

**¿Qué es?**

* Entorno de ejecución de JS fuera del navegador
* Fuera del navegador
* Su uso en Servidores, IoT, herramientas

**Bases**

* Concurrente  
  – Entradas y salidas asíncronas  
  – Un proceso por cada núcleo del procesador
* Motor V8  
  – JS en código máquina, en vez de interpretarlo.
* Modular  
  – Todo lo que no sea sintaxis, son módulos
* Orientado a eventos  
  – Bucle de eventos que se ejecutan constantemente  
  – Reactivo

Actualmente Node se ha convertido en una de las tecnologías más relevantes en la industria de la programación, debido que nos permite correr código de una manera rápida y escalable de grandes maneras desde un servidor.

Node en si es un entorno de desarrollo que nos permite ejecutar JavaScript fuera del navegador, fue creado en el 2009 y corre con el motor V8 creado con C++, lo que nos ofrece una gran potencia a la hora de desarrollar.

Como características de Node principales, es concurrente, gracias a su procesador que es realmente monohilo, este nos permite realizar procesos de forma asíncrona, haciendo así que nunca se quede bloqueado.

# 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:** Se encarga de resolver los eventos ultra rápidos que llegan desde el Event Queue. En caso de no poder resolverse rápido, enviá 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.

# PROCESO DE NODE

# 1.- Va a abrirse un proceso, ese proceso es un proceso de node

# 2.- Interpreta todo el archivo

# 3.- Convertirlo a código maquina

# 4.- Prepara todo lo que necesita para ejecutarse

# 5.- Se ejecuta

# 6.- Se cierra el proceso, y termina

# DESVENTAJAS MONOHILO

# - Si no se manejan bien los errores y uno truena, ya no continua con los procesos posteriores

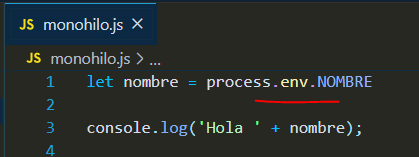
# - Debes estar pendiente de todo el código

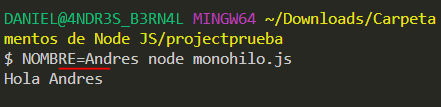
# Importante: Cuando ocurre un error dentro de alguno de los hilos y no se controla apropiadamente (catch); Node detiene todos los hilos de ejecución. Esto puede ser muy peligroso, debido a que es difícil determinar cuál fue el origen del problema y en que punto de ejecución se encontraba cada hilo cuando fue detenido.

# Variables de entorno.

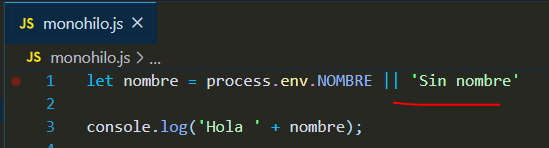
Las variables de entorno son una forma de llamar información de afuera a nuestro software, sirve para definir parámetros sencillos de configuración de los programas de modo que puedan ejecutarse en diferentes ambiente sin necesidad de modificar el código fuente de un script.

El objeto **process** nos da información sobre el procesos que está ejecutando este script.  
La propiedad **env** es la que nos da acceso a las variables de entorno de manera sencilla.

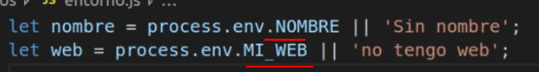




Si queremos dejar un valor por defecto. Quedaría de las siguiente manera:



Detalle importante como buena practica colocar las variables de entorno en MAYUSCULA.



# Herramientas para ser más felices: Nodemon y PM2.

* **Nodemon** (***Desarrollo***): Es un gestor que nos ayuda a detectar los cambios, compilarlo y ejecutarlos.

<https://nodemon.io/>

**Para su instalación:**



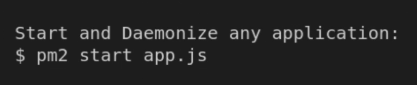
**Para su uso:**



**PM2** (***Producción***): Es parecida a nodemon simplemente más avanzada y mas compleja. No se debe utilizar en desarrollo por que dará más problemas que soluciones. Nos ayudara a ver los datos de nuestra aplicación en producción, como el uso del CPU, memoria, cuantas veces se ha reiniciado.

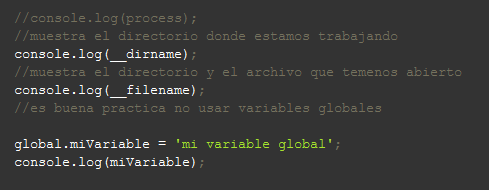
<https://pm2.keymetrics.io/>





# Globals.

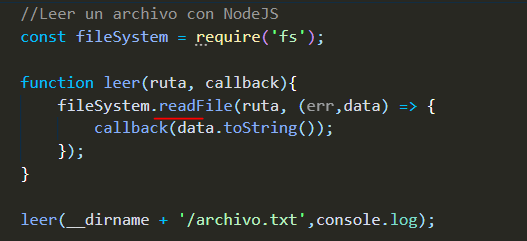
# 

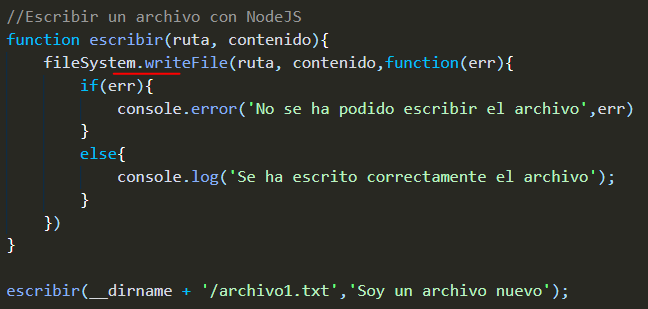


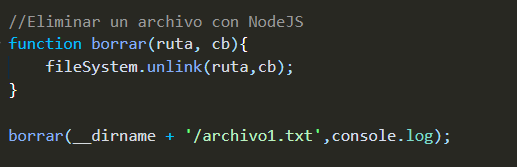
# File System.

En node podemos **crear, editar y eliminar archivos** desde el servidor. La mayoría de estas acciones tienen métodos síncronos y asíncronos, por ejemplo readFile y readFileSync.

“Use readFile siempre que sea posible, readFileSync bloquea el hilo mientras la solicitud es resuelta.”



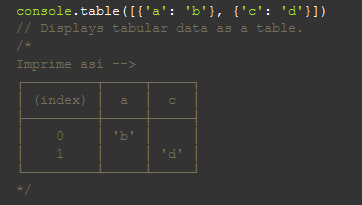


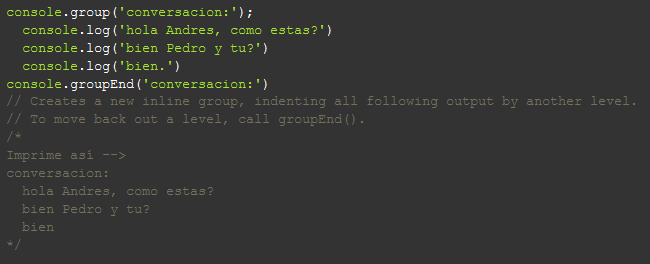


## **Console.**

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

* **console.log**: recibe cualquier tipo y lo muestra en el consola.
* [**console.info**](http://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 identación.
* **console.groupEnd**: finaliza la agrupación.
* **console.clear**: Limpia la consola.



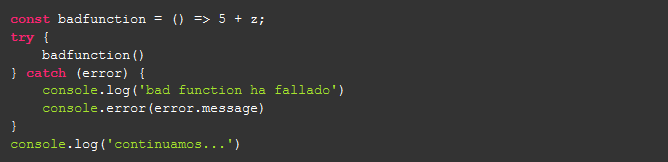


## **Errores (Try/Catch)**

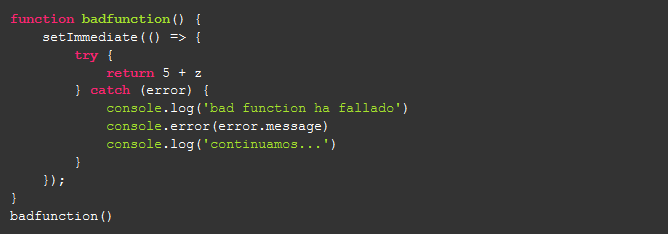
Cuando se genera un error, node propaga el error hacia arriba, hasta que esta es capturado. Si el error no se captura node se detiene.

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

Try/Catch nos permite capturar los errores:



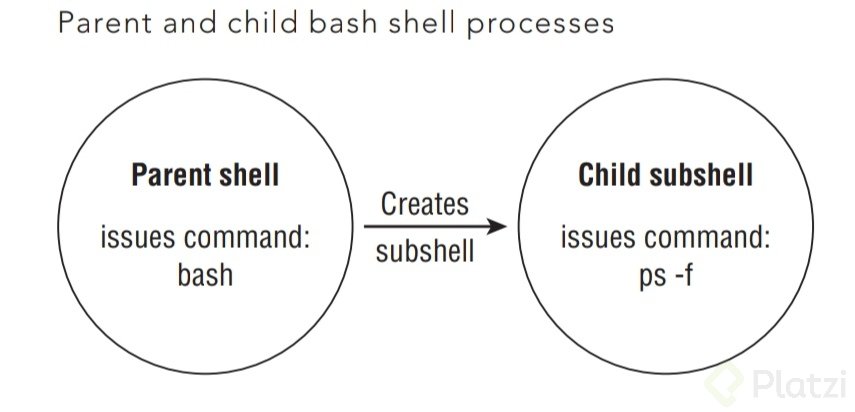
Debemos tener bien presente que hay funciones (en este caso setImmediate) que no se van a ejecutar sobre el hilo principal por lo que tendremos que controlar los errores directamente en la funcion si deseamos manejar errores asíncronos:



## **Child Process.**

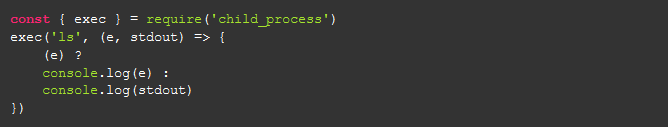
**Funcionamiento de los child process**  
Un proceso es la ejecución de algo (programa, commando, etc) que terminara en un tiempo finito.

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

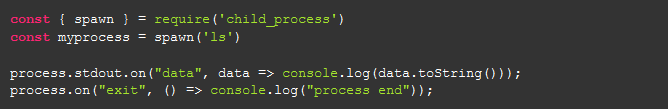


En node podemos crear procesos hijos que ejecuten cualquier acción de nuestro sistema operativo, como si nos encontráramos en la línea de comandos.

Podemos llamar a exec para ejecuciones sencillas:

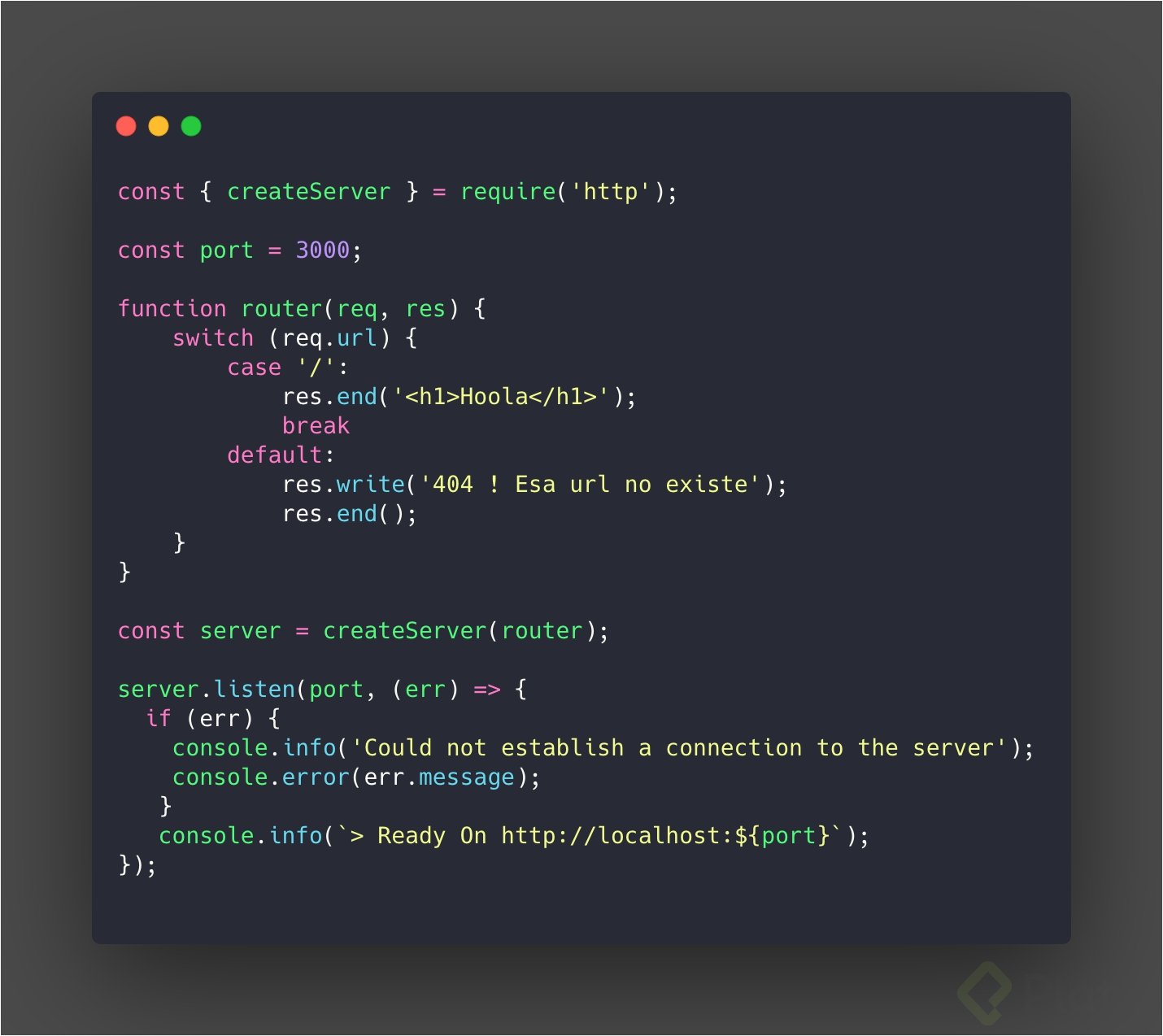


Podemos llamar a spawn para obtener el proceso: La ventaja de este enfoque es que obtienes mayor control del proceso, y del estado en el que se encuentra.



**HTTP.**

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

* **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.



# Aquí las opciones de process: <https://nodejs.org/dist/latest-v12.x/docs/api/process.html>

# Gestión de paquetes: NPM.

**npm** (Node Package Manager) es un administrador de paquetes que permiten ejecutar funciones ya realizadas y validadas y de esta manera acelerar y asegurar la calidad de nuestro proceso de desarrollo.

Podemos buscar módulos para casi todo en:

<https://www.npmjs.com>

**// Para instalar un módulo de npm en nuestro proyecto**

$ npm install is-odd

**// Para requerir el modulo**

*const isOdd = require('is-odd');*

*console.log(isOdd(3)); // true*

**// Para revisar que los paquetes no están actualizados a nivel global dentro de nuestro proyecto**

npm outdated -g --depth=0

Va a imprimir algo así:

Package Current Wanted Latest Location

firebase-tools 8.0.1 8.0.2 8.0.2 global

npm 6.13.7 6.14.4 6.14.4 global

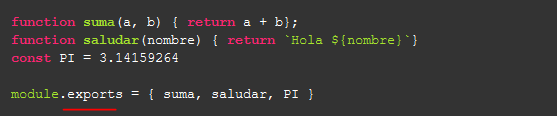
**// Para actualizar todos los paquetes a nivel global dentro del proyecto**

npm update -g

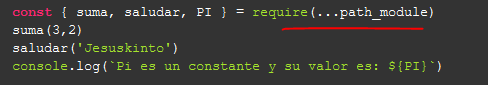
# Construyendo módulos: Require e Import

En Node tenemos una forma de importar módulos la cual es con el método **require**, el cual es la forma por defecto de importar módulos, ya sean nuestros propios módulos como los de otras personas en nuestros proyectos JS, pero suele haber mucha confusión debido al import.

**// Modulo**



**//Importación**



**Import** es la forma de importar módulos en Ecmascript, el cual es un estándar de JavaScript para la web, esta forma de importar en teoría Node no la acepta oficialmente, a no ser que usemos su modo de .mjs “***index.mjs***”.



# Módulos útiles.

# El módulo de cifrado de ****bcrypt**** nos permite construir una plataforma de seguridad utilizando contraseñas encriptadas con Salt.

# <https://www.npmjs.com/package/bcrypt>

# 

# 

# Moment. js es una librería que nos permite solventar problemas con formateo de fechas e implementa un sistema de manejo de fechas mucho más cómodo.

# <https://momentjs.com/>

# 

# 

# Sharp puede convertir imágenes grandes en imágenes JPEG, PNG más pequeñas y compatibles con la web de diferentes dimensiones.

# <https://sharp.pixelplumbing.com/>

# 

# Datos almacenados vs en memoria.

**Memoria RAM**: Es la memoria de corto plazo del computador. Su función principal es recordar la información que tienes en cada una de las aplicaciones abiertas en el computador, mientras este se encuentre encendido. Sin embargo cuando escribimos en memoria es **ultra** **rápido** porque la CPU y la MEMORIA RAM a nivel interno están muy cerca y sus protocolos de conexión son muy rápidos.

**Disco Duro**: El disco duro guarda y protege los datos a largo plazo, lo que significa que quedarán guardados incluso si se apaga el computador. Sin embargo cuando queremos escribir en disco entramos en procesos mucho más lentos, porque el disco esta pensado para almacenar mucha información (1tb, 2tb, etc) en comparación de una Memoria Ram que su uso es exclusivo para la data que se esté manipulando dentro de los procesos corriendo por nuestro procesador y que corren en los thread de los núcleos.

# Buffers.

# **Los buffers son conjuntos de datos en crudo, datos binarios, que podemos tratar en NodeJS para realizar diversos tipos de acciones.**

# **Buffer es la forma en que podemos leer los datos es su forma más sencilla, en este caso Node transmite estos datos Buffer en binario.**

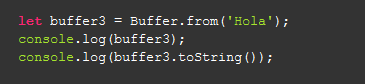
# **Poder convertir datos a Buffer nos ayudara a la velocidad y mejorar el rendimiento de la lectura y escritura de datos, debido que se hace en el lenguaje de más bajo nivel para las maquinas.**

# 

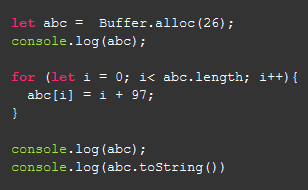
# **Para crear un buffer, con 4 espacios por ejemplo, podemos hacerlo con la siguiente sintaxis.**

# 

**Otras formas de crear un buffer.**



**Guardar el abecedario en un buffer.**



# Streams.

# Los streams son puntos de lectura de Buffers, con estos podemos declarar que acciones ejecutar cuando se recibe un buffer. Existen tres tipos de streams, stream de lectura, de escritura y de lectura y escritura.

# Un gran ejemplo de uso de los streams es el procesamiento de archivos grandes, como imágenes o videos, ya que podemos transformar estos a buffer y a través de los streams cargarlos o guardarlos parte por parte para mejorar el rendimiento de nuestro código.

# 

# Benchmarking (console time y timeEnd).

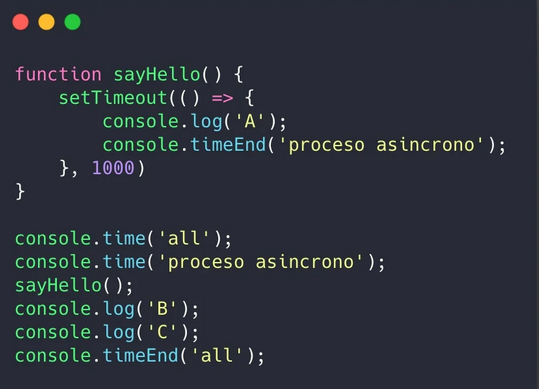
# La función console.time(‘nombre’) inicia un temporizador que se puede usar para rastrear cuánto tiempo dura una operación. El temporizador será identificado por el nombre dado a la consola. Ese mismo nombre se utilizará cuando se llame a console.timeEnd(‘nombre’) para detener el temporizador y obtener el tiempo demorado durante el proceso.

# 

# Su salida sera:



Benchmark = prueba de rendimiento o comparativa en inglés.



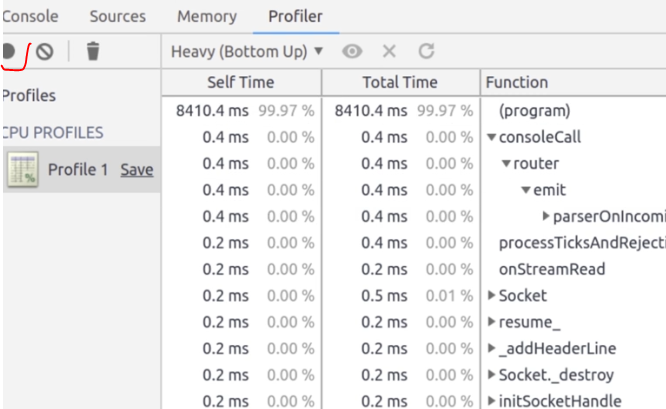
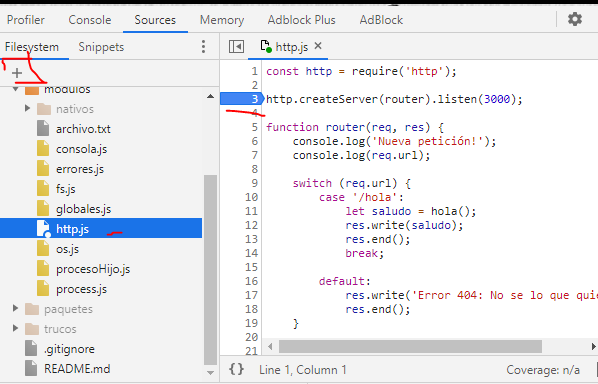
# Debugger.

# La característica de hacer debug de aplicaciones de **Node** en el navegador solo es soportada por Chrome. En **Firefox** no es posible realizarlo debido a que usa un **JavaScript Engine** diferente a **V8**.

# Para debuggear lo que tendremos que hacer es iniciar node con el flag *–inspect.*

# 

# Ingresamos a chrome://inspect y desde allí ya nos debe aparecer lo que estamos corriendo y abrirlo dando click sobre inspect.



# Error First Callbacks.

# *Si el V8 es el corazón de Nodejs, los callbacks son sus venas. 💚*

# Dentro de nuestro código JS siempre vamos a hacer uso de callbacks (funciones que pasamos como parámetros), debemos siempre tener muy presente de que si algo puede fallar , eventualmente fallará, asi que como buena practica debemos controlar los errores dentro de la asincronía. Lanzar una excepcion con Throw dentro de un callback asincrono no va a funcionar, solo nos va a funcionar en un bloque de código síncrono. Es por esto que aplicaremos Error First Callbacks.

# Los Error First Callbacks se utilizan para pasar primero el error y los datos posteriormente. Entonces, puedes verificar el primer argumento, es decir, el objeto de error para ver si algo salió mal y puedes manejarlo. En caso de que no haya ningún error, puedes utilizar los argumentos posteriores y seguir adelante.

# 

# Esto lo que nos permitirá es tener control de errores sobre la asincronía. (Callbacks)

# Scraping.

# Web Scraping es una técnica utilizada mediante programas de software para extraer información de sitios web. Usualmente, estos programas simulan la navegación de un humano en la World Wide Web ya sea utilizando el protocolo HTTP manualmente, o incrustando un navegador en una aplicación.

# 

# Automatización de procesos.

# Gulp es una herramienta que nos va a permitir automatizar tareas o procesos de la siguiente forma. Antes debemos instalarlo previamente con: *npm i gulp gulp-server-livereload*.

# 

# 

# 

# Aplicaciones de escritorio.

# Con NodeJS vamos a poder convertir nuestras aplicaciones Web en aplicaciones de escritorio con ayuda de Electron. Para ello debemos instalar electron:

# *npm i electron*

# Luego importaremos electron y realizaremos las siguientes configuraciones:

# 

# Para finalizar configuramos en el Package.json y ejecutamos:

# 

# 

# Importants Questions and Answers.

# 

# 

# 

# 

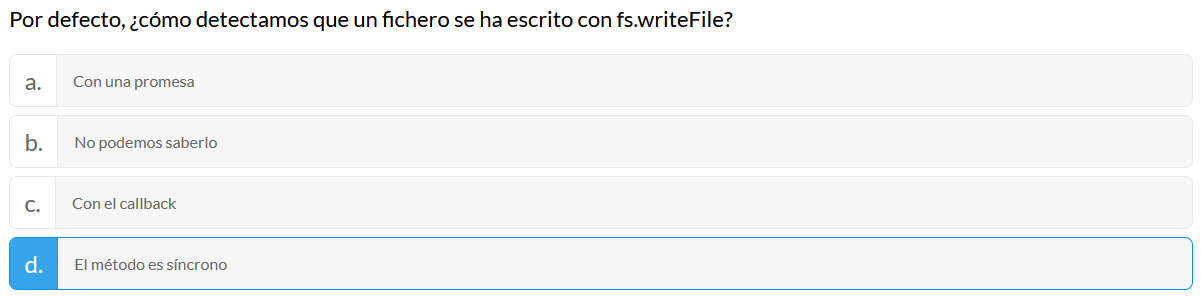
# 

# 

# 

# Malas.

# 



**Curso de Backend con NodeJS.**

**¿Qué es Node.js?**

Node.js es un entorno de ejecución para JavaScript construido con el motor JavaScript V8 de Chrome. JavaScript es un lenguaje interpretado pero en Node.js tenemos algo llamado el JIT Compiler que es una especie de monitor que optimiza fragmentos de código que son ejecutados frecuentemente.

**¿Qué es node.js ?**  
Node.js es un entorno de ejecución para javascript

**¿Qué es un entorno de ejecución?**

Un entorno de ejecución es una capa encima del sistema operativo que ejecuta una pieza de software.

**¿Qué nos permite?**

Usar javascript para el servidor.

## **¿Qué es Node.js y para qué sirve?**

La definición formal de **nodejs es:** un entorno de ejecución para javascript construido con el motor v8.

**El entorno de ejecución:** es la capa que corre por el sistema operativo que ejecuta software, básicamente se encarga de cómo se consume memoria, como acceder a las variables y como corre el [garbage collector](https://es.wikipedia.org/wiki/Recolector_de_basura).

[Chrome V8](https://es.wikipedia.org/wiki/Chrome_V8) Es un engine de Javascript por de **chromiund-project** para Chrome y chromium. Además de Chrome existen **2 proyectos** que son **chromium** que es la versión open source y luego **Chrome canary**, chrome canary se llama así por una analogía donde antiguamente los mineros iban a la mina y para detectar si había gases o algún peligro, ponían a un canario en una pequeña jaula, si había un gas y pasaba algo, el canario lastimosamente moría y es la manera en cómo se daban cuenta si había algún error, lo mismo pasa con chrome canary, es la manera como detectan errores y si todo sale bien, lo pasan a chrome.

[Chrome V8](https://es.wikipedia.org/wiki/Chrome_V8) lo que hace es compilar javascript a código máquina. Recordemos que los **lenguajes interpretados** se ejecutan muy rápido, pero cuando hay un **loop** de código muy seguido **se demoran**, porque cada vez que pasan por esa linea de código tendrán que **volver a interpretar** a diferencia de los **lenguajes compilados** que se demoran **mucho en cargar**, porque tienen que pasar precisamente por este proceso de compilación, pero luego **se ejecutan muy rápido** porque ya está compilada esa linea, por eso cada vez que vuelven a pasar por ese loop, ya está perfectamente compilado.

**Javascript solía ser interpretado** y **ahora es compilado** con una tecnología llamada **Just in time Compiler** o [compilación en tiempo de ejecución](https://es.wikipedia.org/wiki/Compilaci%C3%B3n_en_tiempo_de_ejecuci%C3%B3n), está tecnología lo que tiene es un monitor que se encarga de revisar cada cuanto se ejecuta nuestro código, si el código se ejecuta mucho pone un estado warm y lo que hace es que ese código lo compila, si ese código compilado se **ejecuta muchas veces**, lo coloca en un estado **HOT** y es básicamente hacerle una **optimización** a ese compilado, para que cuando se llame, ya llame a la versión optimizada.

Nodejs fue tomar el engine de JS chrome V8 para crear un entorno de ejecución y poder usar javascript del lado del servidor, recordemos que tenemos otros engine de JS como: [SpiderMonkey](https://es.wikipedia.org/wiki/SpiderMonkey), [JavascriptCore](https://es.wikipedia.org/wiki/JavaScriptCore) y [Chakra](https://es.wikipedia.org/wiki/Chakra_(int%C3%A9rprete_de_JScript)). Pero como recientemente van a renovar la versión de Edge van a empezar a implementar el motor V8 como Js engine.

## **Fechas importantes de NodeJS**

* En **2009** por primera vez [Ryan Dahl](https://en.wikipedia.org/wiki/Ryan_Dahl) mostró al mundo nodejs.
* En **2011** por primera vez LinkedIn usa nodejs en producción.
* En **2013** se saca Gust que es una Plataforma de plugin.
* A la vez Paypal saca un framework de nodejs llamado [Krakenjs](https://github.com/krakenjs/kraken-js).
* En **2015** sale la competencia de nodejs llamada IOJS, pero afortunadamente se reconcilian y crean lo que hoy es [La Nodejs Foundation](https://foundation.nodejs.org/).
* En **2017** Nodejs Se vuelve Messing con un 8.8 millones de instancias funcionando.

**Diferencias entre Node.js y JavaScript**

En ***JavaScript*** del lado del cliente tenemos el DOM y el CSSOM así como el objeto window para manipular los elementos de nuestra página además una serie de APIs, aquí unos ejemplos:

* fetch
* SessionStorage y LocalStorage
* canvas
* bluetooth
* audio
* web authentication

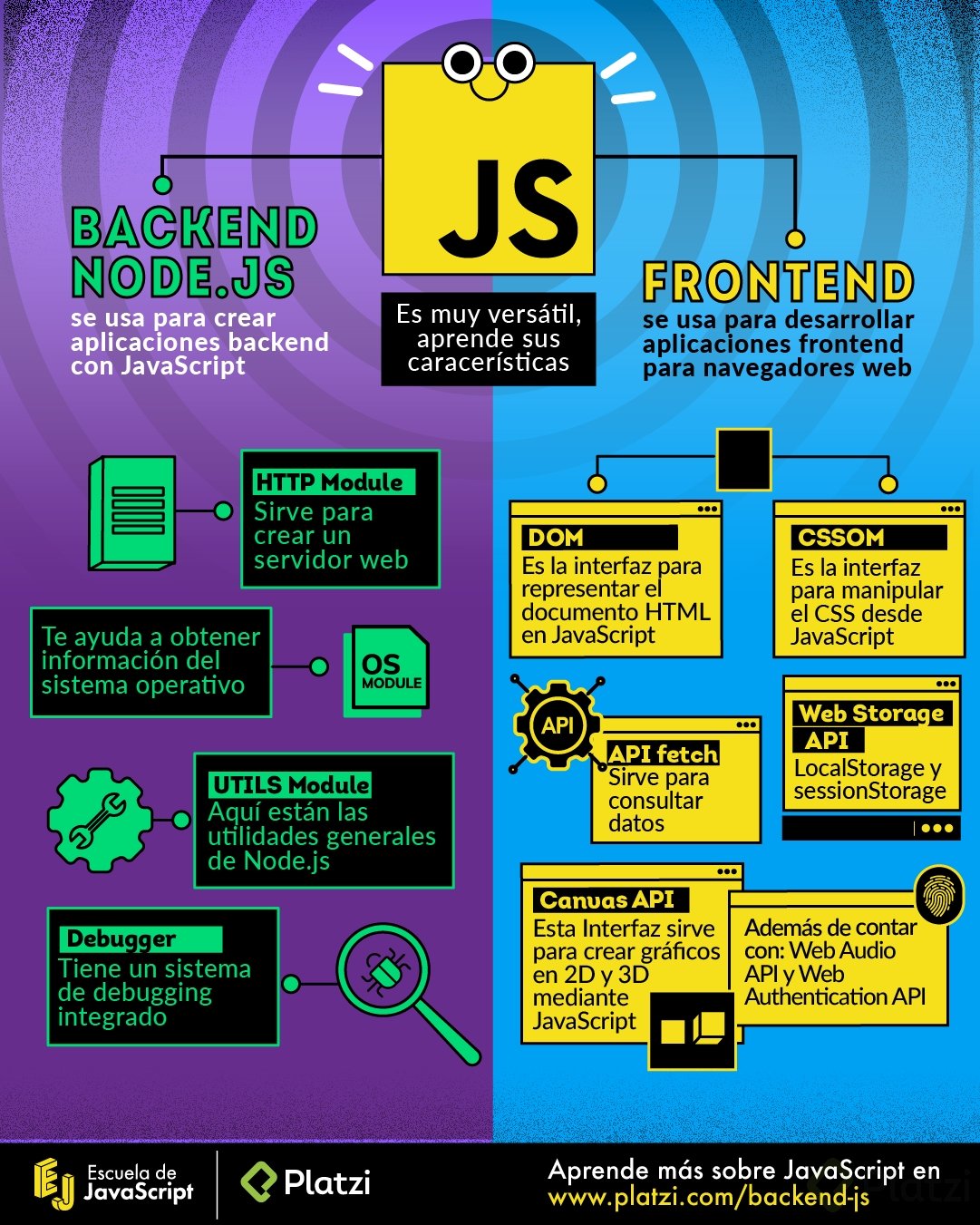
Mientras que en ***Node.js*** no tenemos un DOM ni un objeto window, lo que sí tenemos son una serie de módulos que nos permiten interactuar con los recursos de la máquina como el sistema operativo y el sistema de archivos, por ejemplo:

* **OS (Sistema operativo):** nos permite comunicarnos entre el sistema operativo y darnos información sobre él.
* Fs (**File System**)
* **Http**: permite crear servidores
* **Util:** que son una serie de utilidades exclusivas para nodejs.
* **Debugger** (manejado directamente en el navegador)
* **Stream:** nos permiten manejar grandes colecciones de datos.
* events

## **Instalación de Node.js**

Para instalar Node.js tienes que dirigirte a [nodejs.org](https://nodejs.org/es/) y elegir entre la última versión o la version LTS.

Por defecto Node.js detecta tu sistema operativo y descarga el archivo indicado para la instalación, si no es tu caso puedes dirigirte al enlace de otras [descargas](https://nodejs.org/es/download/).



**Configurar .config en Node.**

Existe una libreria en **npmjs** que se llama [dotenv-flow](https://www.npmjs.com/package/dotenv-flow) que permite manejar estas configuraciones.

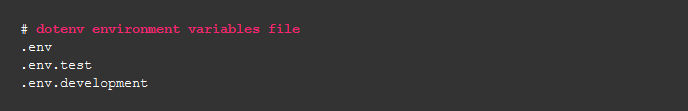
* Instalar el paquete



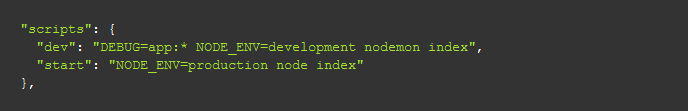
* Creé un nuevo archivo de configuración **.env.development** de copia del **.env** y le cambié los parámetros
* En el archivo /config/index.js cambié la libreria a usar por dotenv-flow



* Agregue al .gitignore el archivo .env.development para que no sea cargada mis credenciales de desarrollo al repositorio



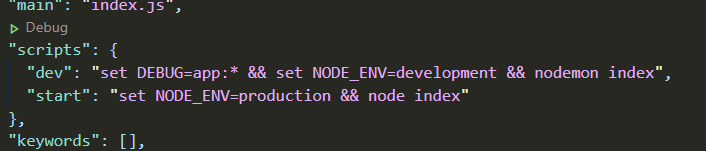
* En el package.json modifiqué el script dev agregando a la variable de NODE\_ENV=development

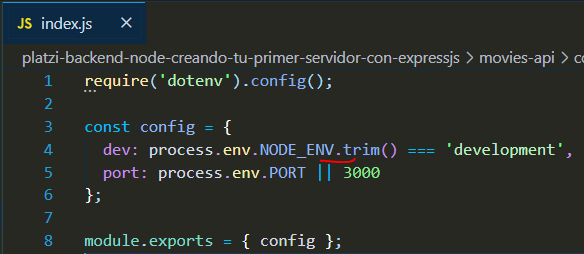


Con estos cambios se logra separar las configuraciones del ambiente de producción al ambiente de desarrollo. El paquete de dotenv-flow se encarga basado en NODE\_ENV cargar el archivo de configuración respectivo. Este paquete tiene más opciones, aquí esta el enlace del repositorio [dotenv-flow](https://www.npmjs.com/package/dotenv-flow)

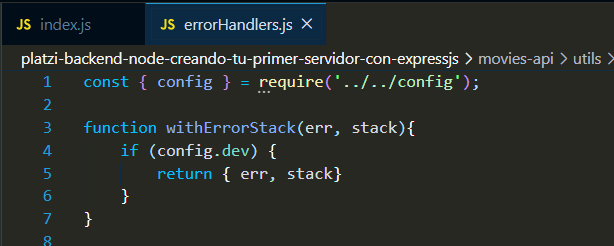
**Variables de entorno.**

Cuando configuremos nuestros scripts dentro del ***package.json*** podemos hacer uso de las variables de entorno con **set VARIABLE\_ENTORNO=VALUE && next..** y de esta manera acceder a esas variables con **process.env.VARIABLE\_ENTORNO** y configurar en nuestro archivo de **config** si estamos o no en modo de desarrollo





Y de este manera cambiar la lógica de nuestro programa según sea el caso.



# Arquitectura orientada a eventos.

Uno de los paradigmas de programación en nodejs es la arquitectura orientada a eventos, los eventos nos permiten manipular el código asincrono de una mejor manera. Sabemos que este código asincrono lo podemos manipular a través de Callback, Promesas y Async/Await, sin embargo hay una forma más poderosa: **Event Emiter.**

## **Event Emiter**

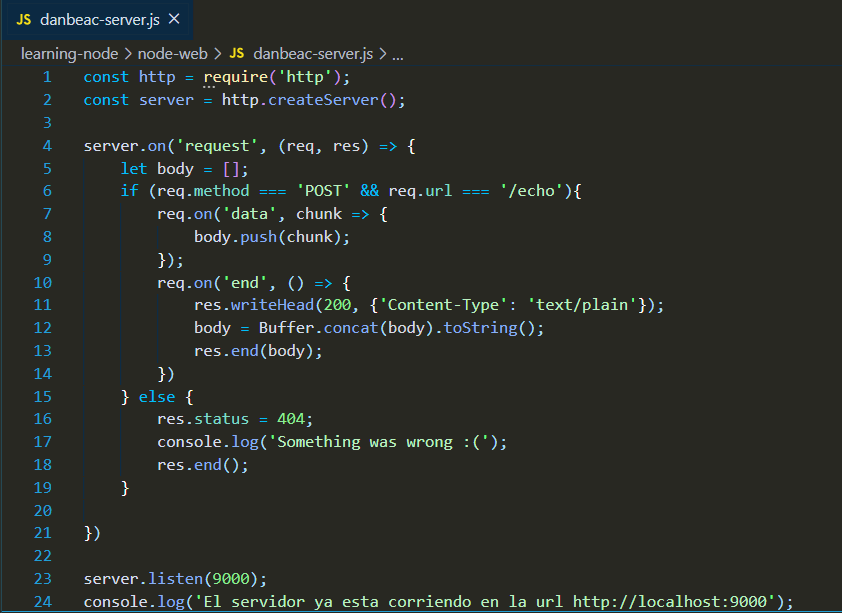
Hay una forma aún más poderosa de ejecutar el código anterior y es con la clase event-emitter. [EventEmitter](https://nodejs.org/dist/latest-v10.x/docs/api/events.html#events_class_eventemitter) no es exclusivo, podemos usar promesas y código asincrono, pero ya vamos a ver cuáles son sus ventajas:



[***https://nodejs.org/api/events.html***](https://nodejs.org/api/events.html)

# Node.js para la web

Con Node vamos a poder iniciar servidores, para ello vamos a crear el servidor importando inicialmente http y crearemos el server con el método createServer del mismo. Este servidor va tener un **event emiter** es decir que va estar escuchando eventos entre los cuales vamos a escuchar un ‘request’. Dentro de este vamos a poder llevar a cabo toda la configuración necesaria.

Es importante mencionar también que nuestro request (en este caso **req**) también va a tener un event emiter por lo que va a estar escuchando eventos tales como ***‘data’***  para extraer la información enviada (chunk) o el ***‘end’*** para cuando finalice el request (**req**)

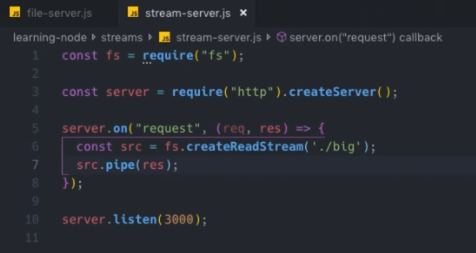
# Introducción a streams.

Los Streams son una colección de datos como los arrays o strings sólo que se van procesando pedazo por pedazo, esta capacidad los hace muy poderosos porque podemos manejar una gran cantidad de datos de manera óptima.

Este es un ejemplo en el que el consumo de memoria en nuestro equipo es excesivo por estar leyendo un archivo demasiado grande.



Podemos optimizarlo con el uso de Streams mejorando el uso de memoria de la siguiente manera:y7



# Readable y Writable streams

Los Readable y Writable streams tienen los siguientes eventos y funciones respectivamente:

## ***Readable***

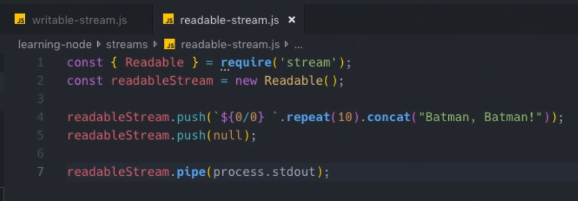
### **Eventos**

* **data**. Se dispara cuando recibe datos.
* **end**. Se dispara cuando termina de recibir datos.
* **error**. Se dispara cuando hay un error.

### **Funciones**

* pipe
* unpipe
* read
* push

En este ejemplo de uso de readableStreams, lee unos datos que pasamos con el método **push** y que se repetirán 10 veces. Para indicar que ya no recibe más datos pasamos **null.**

+



En este otro ejemplo imprimiremos todo el abecedario.

## ***Writeable***

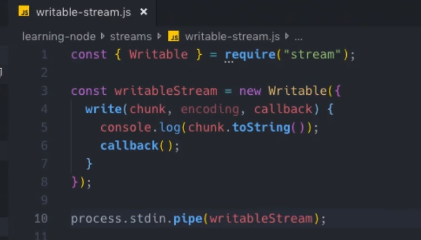
### **Eventos**

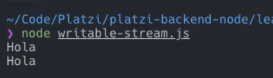
* **drain**. Se dispara cuando emite datos.
* **finish**. Se dispara cuando termina de emitir.
* **error**. Se dispara cuando hay un error.

### **Funciones**

* write
* end

Recuerda que tienen estos eventos porque los heredan de la clase **EventEmitter**.

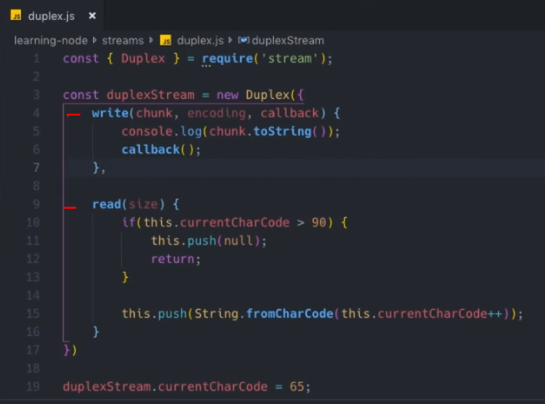
Este es un ejemplo muy básico del uso de writableStreams, en el que recibe un dato y lo imprime.



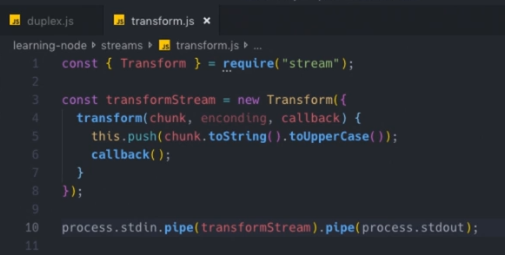
**Duplex y Transforms streams**

Ambos sirven para simplificar nuestro código:

* **Duplex:** implementa los métodos write y read a la vez.

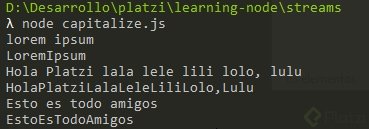


* **Transform:** es similar a Duplex pero con una sintaxis más corta.



Este es otro ejemplo en el que hacemos un camelCase a la cadena de texto que reciba como entrada.





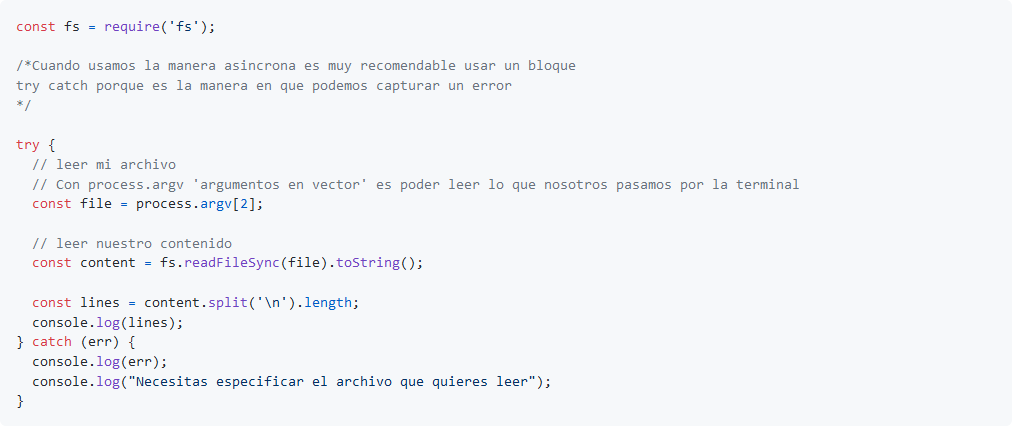
**Sistema operativo y sistema de archivos**

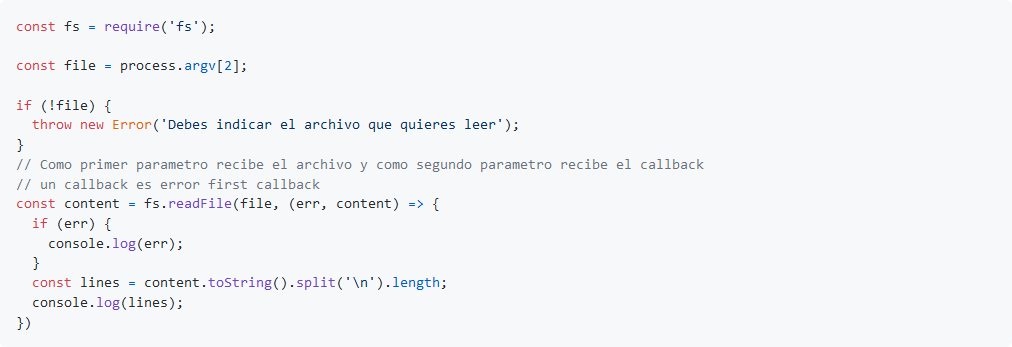
En esta clase vemos dos módulos básicos:

* **os**. Sirve para consultar y manejar los recursos del sistema operativo.



* **fs**. Sirve para administrar (copiar, crear, borrar etc.) archivos y directorios.

En node las utilidades, la mayoría pueden funcionar de manera síncrona y asíncrona, de manera **síncrona**, quiere decir que él **va a esperar a hasta que se ejecuta ese proceso** y hasta que no dé una respuesta no va a continuar con la siguiente linea.

Vamos a hacer lo mismo de manera asíncrona, pero esta vez no tenemos que especificar que es readFileSync sino que es simplemente **readFile**, porque **por defecto Nodejs trata de ser asincrono**.

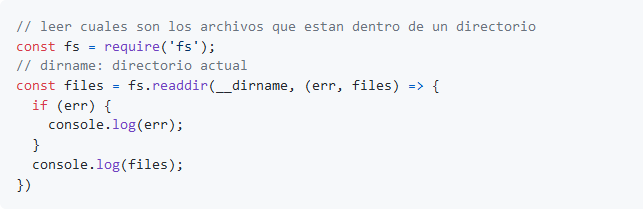
Los métodos contenidos en estos módulos (y en todo Node.js) funcionan de forma asíncrona por default, pero también se pueden ejecutar de forma síncrona, por ejemplo el método readFile() tiene su versión síncrona readFileSync().

El **módulo file system** no solo nos permite leer archivos sino que en el también podemos **crear carpetas, leer directorios, crear archivos, eliminar archivos, etc**. Es todo lo que un usuario puede hacer con archivos y carpetas.

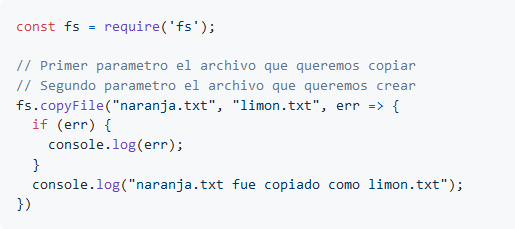
También podemos visitar la documentación de Nodejs 12.0 sobre [FileSystem](https://nodejs.org/api/fs.html#fs_file_system)

## **Administrar directorios y archivos.**

Leer el nombre de los archivos que están dentro de un directorio

Creamos directorios o carpetas con **mkdir desde node**.

**Copiar archivos con Nodejs**.



# Consola, utilidades y debugging.

# En nodejs tenemos la utilidad consola, en ella podemos usar el **[console.info]**, **[console.log]**, **[console.error]**, estamos bastante familiarizados con esta utilidad, sin embargo todo por defecto lo que imprimimos por consola se va por el **stdout**, y todo lo que imprimimos en el console.error se va por el **stderror**. Mediante la **clase Consola** que es diferente a la consola, podemos personalizarlo para decir que en vez se valla por el stdout o stderror, hagamos una cosa completamente diferente.

# 

No solo podemos jugar con la clase de la consola para crear nuestra consola personalizada, sino que también vamos a explorar diferentes utilidades de consola.

**console.log por debajo trabaja con una utilidad llamada util format**:

* **%s**: **String**
* **%n**: **Número**
* **%j**: **Json**

Estos son pequeños **placeholders** para formatear nuestros datos. console.log("Un %s y un %s", "Perrito", "Gatito");

Si accedemos a la consola de node exactamente esto hace la utilidad **util.format()** es decir la consola se alimenta del paquete **[util.format]** y funciona exactamente igual.

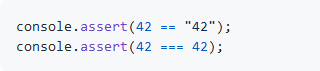
Node ocupa este paquete de utilidades para otros paquetes que el expone, pero nos deja la posibilidad de alimentarnos de estas utilidades, es decir si yo por alguna razón quiero hacer uso del **[util.format]** lo podemos hacer.

Alias de console.log:

* console.info

Alias de de console.error:

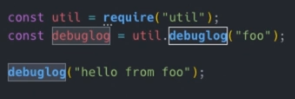
* console.warm
* **console.assert:** Si hay un error nos muestra que existe un error en un assert, en un booleano o verificación:



* **console.trace**: nos indica la linea donde está ocurriendo el error que es mucho más específico.

Una utilidad bastante interesante es una llamada de [debuglog](https://nodejs.org/dist/latest-v8.x/docs/api/util.html#util_util_debuglog_section), lo que tenemos que hacer es obtener la utilidad, por la cual node nos la deja abierta para que la hagamos, e invocamos el debuglog, llamando **util.debuglog**. Esto es muy parecido a cómo funciona el paquete debug de express, pero nosotros básicamente lo que decimos es crear un debugging.

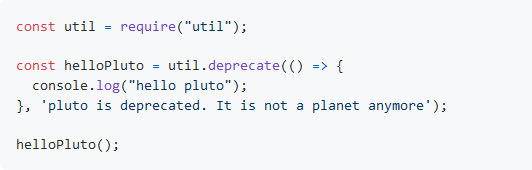
Vamos a crear un nuevo debugging que va a exponer un namespace que va a ser foo Nosotros podemos imprimir nuestro mensaje de debug "hello" y esto solo solo se imprime si pasamos la variable de entorno NODE\_DEBUG con el namespace.





## **Deprecate**

Cuando hacemos util.deprecate hacemos un wrap de una función que ya está obsoleta y queremos hacer saber a nuestros usuarios que ya no debería de usar, lo interesante de nuestra función deprecate es que nos permite imprimir un mensaje.



Esto es bastante util cuando nosotros estamos haciendo refactory y nosotros queremos hacerle saber al usuario que hay ciertas funcionalidades que quizás en una versión más adelante va a desaparecer por completo.

## **Debugging en node**

Para hacer debugging en node lo que debemos usar es el flag node --inspect y luego especificar al archivo que queremos hacer debugging. En versiones anteriores de node es decir < 12, hacer uso de node debug, genera un warning o deprecation warning que es exactamente igual al util-deprecate que aprendimos con anterioridad, esto quiero decir que en futuras versiones de node, esto va a desaparecer. Por lo que la recomendación es empezar a usar el node --inspect desde ahora.

Cuando hacemos el node inspect, el abre un puerto en el localhost en nuestro navegador. Si nos damos cuenta es exactamente la utilidad debugging que tiene js del lado del cliente.

# <https://nodejs.org/api/console.html>

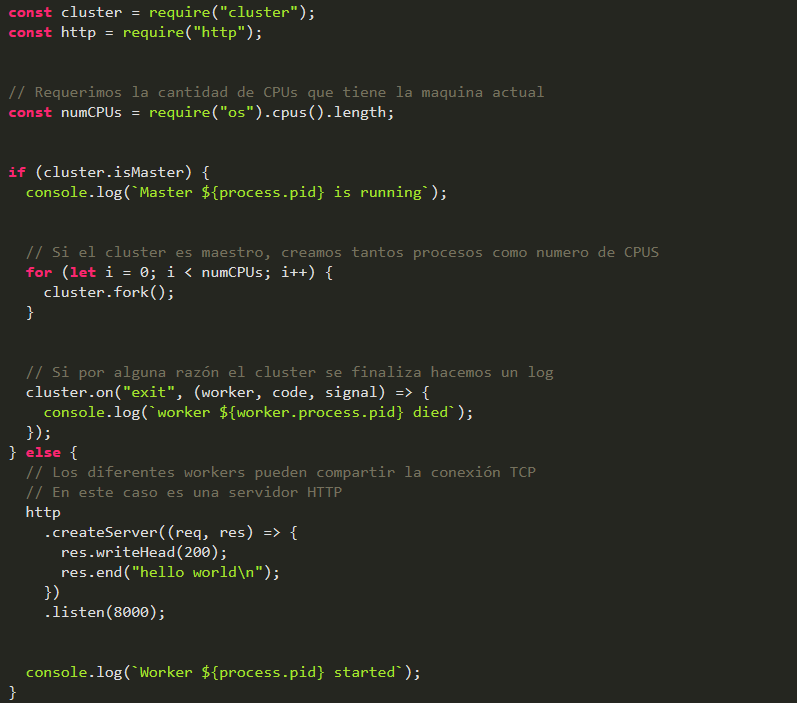
# <https://nodejs.org/api/debugger.html>

# <https://nodejs.org/api/util.html>

# <https://github.com/workshopper/learnyounode>

# Clusters y procesos hijos.

Una sola instancia de Node.js corre un solo hilo de ejecución. Para tomar ventaja de los sistemas con múltiples core, necesitamos lanzar un cluster de procesos de Node.js para manejar la carga.

El módulo cluster nos permite la creación fácil de procesos hijos que comparten el mismo puerto del servidor. Veamos un ejemplo en código:

Si corremos nuestro archivo de Node.js ahora compartirá el puerto 8000 con los diferentes workers:

En Windows, todavía no es posible establecer un nombre de proceso server en un worker.

[***https://nodejs.org/api/cluster.html***](https://nodejs.org/api/cluster.html)

# ¿Qué es Express.js y para qué sirve?

# 

Express.js es un framework para crear **Web Apps**, **Web APIs** o cualquier tipo de **Web Services**, es libre bajo la licencia MIT.

Express es muy liviano y minimalista además de ser extensible a través de Middlewares.

Los Middlewares interceptan el request y el response para ejecutar una acción en medio.

# Creando tu primer servidor con Express.js

# 

# 

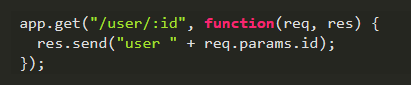
# Request y Response Objects.

# Request Object.

El objeto req (Request) en Express representa el llamado HTTP y tiene diferentes propiedades del llamado, como la cadena de texto *query* (Query params), los parámetros de la URL (URL params), el cuerpo (Body), los encabezados (HTTP headers), etc.

Para acceder al req basta con acceder al primer parámetro de nuestros router handlers (router middleware) o middleware.

Como por ejemplo así lo hemos visto siempre:



Pero también funcionaria sin problemas:



## **Exploremos las propiedades más importantes**

### **req.body**

Contiene los pares de llave-valor de los datos enviados en el cuerpo (body) del llamado (request). Por defecto es undefined pero es establecido cuando se usa algún “body-parser” middleware como body-parser y multer.

En **Postman** cuando hacemos un request y enviamos datos en la pestaña **Body**, estos middlewares son los que nos ayudan a entender el tipo de datos que vamos a recibir en el req.body.

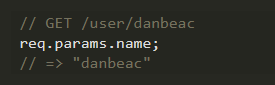
Aquí podemos ver como se pueden usar estos middlewares para establecer el valor del req.body:

# 

Más información sobre los diferentes formatos que puede tener el body: <https://developer.mozilla.org/es/docs/Web/HTTP/Methods/POST>

### **req.params**

Esta propiedad contiene un objeto con las propiedades equivalentes a los parámetros nombrados en la ruta. Por ejemplo, si tenemos una ruta de la forma /user/:name entonces la propiedad name está disponible como req.params.name y allí podremos ver su valor. Supongamos que llamaremos a la ruta con /user/danbeac, entonces el valor de req.params.name sería danbeac. Este objeto por defecto tiene el valor de un objeto vacío {}.



### **req.query**

Esta propiedad contiene un objeto con las propiedades equivalentes a las cadenas de texto query de la ruta. Si no hay ninguna cadena de texto query tendrá como valor por defecto un objeto vacío {}.



Más información sobre los query strings en: <https://es.wikipedia.org/wiki/Query_string> y <https://tools.ietf.org/html/rfc3986#section-3.4>

# Response Object.

El objeto res representa la respuesta HTTP que envía una aplicación en Express.

Para acceder al res basta con acceder al segundo parámetro de nuestros *router handlers* (router middleware) o *middleware*.

Como por ejemplo así lo hemos visto siempre:



# Pero también funcionaría sin problemas:

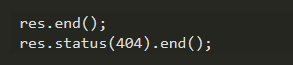
# 

## **Exploremos las propiedades más importantes**

### **res.end()**

Finaliza el proceso de respuesta. Este método viene realmente del core de Node.js, específicamente del método response.end() de http.ServerResponse.

Se usa para finalizar el request rápidamente sin ningún dato. Si necesitas enviar datos se debe usar res.send() y res.json().



### **res.** **json()**

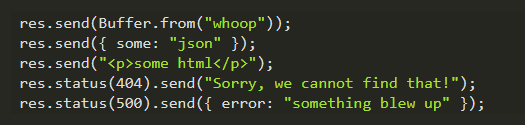
Envía una respuesta JSON. Este método envía una respuesta (con el content-type correcto) y convierte el parámetro enviado a una cadena de texto JSON haciendo uso de JSON.stringify().

El parámetro puede ser cualquier tipo de JSON, incluido un objeto, un arreglo, una cadena de texto, un *boolean*, número, *null* y también puede ser usado para convertir otros valores a JSON.



### **res.** **send()**

Envía una respuesta HTTP. El parámetro body puede ser un objeto tipo Buffer, una cadena de texto, un objeto, o un arreglo. Por ejemplo:



**Anatomía de una API Restful**

**REST (Representational State Transfer)** es un estilo de arquitectura para construir web services, no es un estándar pero si una especificación muy usada.

Las peticiones HTTP van acompañadas de un “verbo” que define el tipo de petición:

* **GET**. Lectura de datos.
* **PUT**. Reemplazar datos.
* **PATCH**. Actualizar datos en un recurso específico.
* **POST**. Creación de datos.
* **DELETE**. Eliminación de datos.

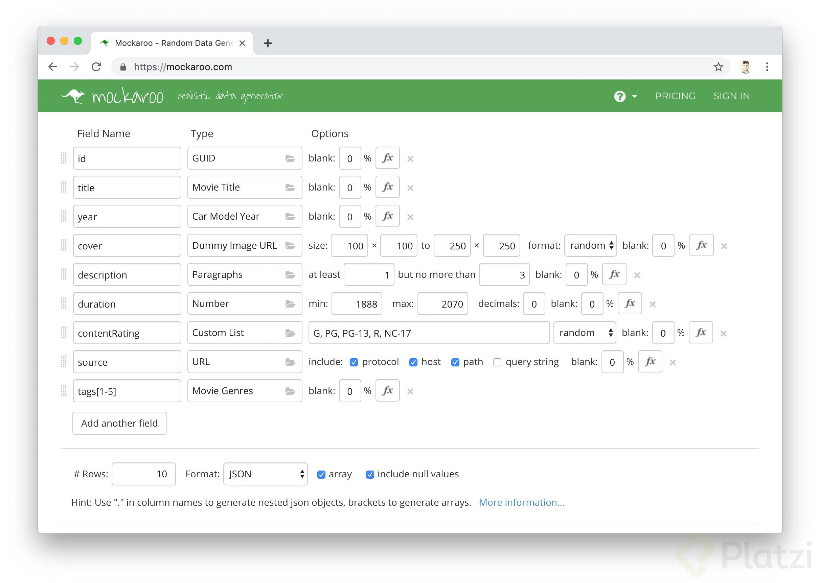
No es recomendable habilitar un endpoint de tipo PUT y DELETE para toda nuestra colección de datos, sólo hacerlos para recursos específicos, ya que no queremos que por error se puedan borrar todos nuestros datos.

# Estructura de una película con Moockaru.

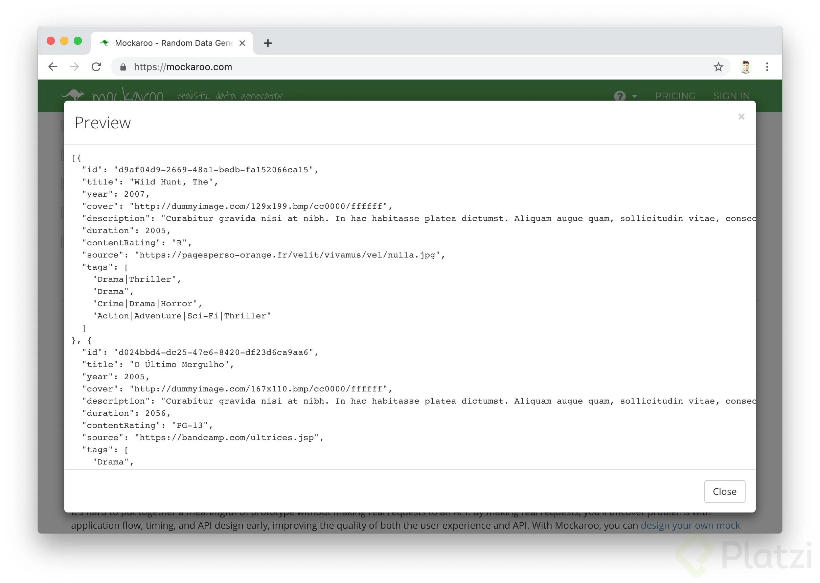
# Mockaroo\* es un servicio que nos permite crear datos simulados a partir de una estructura, por ejemplo para generar la estructura de una película:

# 

Lo que podemos hacer en Mockaroo es seleccionar la siguiente estructura:



Luego seleccionamos el número de filas (rows) que queremos generar y elegimos el formato, en este caso será de tipo JSON. Podemos hacer clic en preview para ver cómo queda y finalmente para descargar los datos hacemos clic en **Download Data**.



**Implementando un CRUD en Express.js**

Las siglas CRUD vienen de las palabras en inglés:

* Create - crear
* Read - leer
* Update - actualizar
* Delete – eliminar



Ahora vamos a implementar un crud en nuestro código:

Para crear una ruta necesitamos de express pues es quien nos define el router, luego vamos a usar en esté caso un archivo de mocks, los **mocks son archivos falsos, de datos falsos**, pero más adelante vamos a aprender cuando nos conectemos con servicios y como conectarnos a la base de datos para traer archivos reales, en este ejemplo lo estamos haciendo, porque lo que nos interesa ahora es entender cómo se definen las rutas y esos archivos de mocks nos van a servir más adelante para escribir test y verificar.



**Pagina para generar .gitignore**

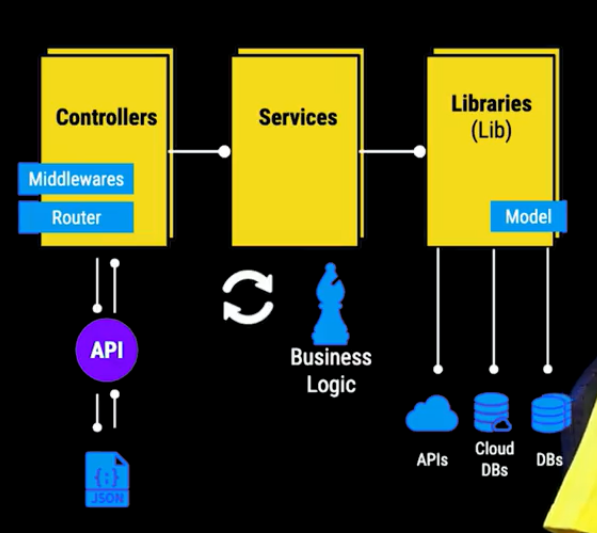


**Consultar Notas del curso:**

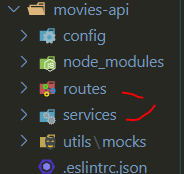
[**https://github.com/JasanHdz/backendnodejs/tree/master/notes**](https://github.com/JasanHdz/backendnodejs/tree/master/notes)

# Implementando una capa de servicios.

La arquitectura tradicional MVC se queda corta en aplicaciones modernas, por eso necesitamos una arquitectura diferente cómo la **Clean Arquitecture** que tiene una capa de servicios para manejar la lógica de negocio.



**¿Por qué la comparto o la recomiendo?**

Porque MVC que es la arquitectura tradicional a la que estamos acostumbrados, se queda corta en las aplicaciones modernas, **no nos basta** solo con tener: **modelo, vista y controlador**. Entonces lo que nosotros definimos en una aplicación en express: **los controladores que son toda la capa de middlewares y el router** que se comunican con la API y reciben o envían JSON, luego tenemos una **capa de servicios**, esta capa es muy importante porque aquí está el corazón de nuestra aplicación, aquí es donde está toda la lógica de negocio y es importante saber que **los controladores NO llaman a otros controladores** los controladores **solo llaman servicios**. Pero **los servicios si pueden llamar otros servicios o llamar librerías**, las librerías **son** la capa que esta adjunta a **librerías externas**, como por ejemplo: **bases de datos, bases de datos que están en la** **nube o incluso otras API**.

Diferentes razones y opiniones sobre porque dejar de usar MVC:

* [Clean coder](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)
* [Ryan Florence](https://www.youtube.com/watch?v=kp-NOggyz54)
* [Laravel no es mvc](https://styde.net/porque-laravel-no-es-mvc-y-tu-deberias-olvidarte-de-mvc/)
* [Twitt sobre mvc por founder de Laravel](https://twitter.com/taylorotwell/status/262290285499936768)

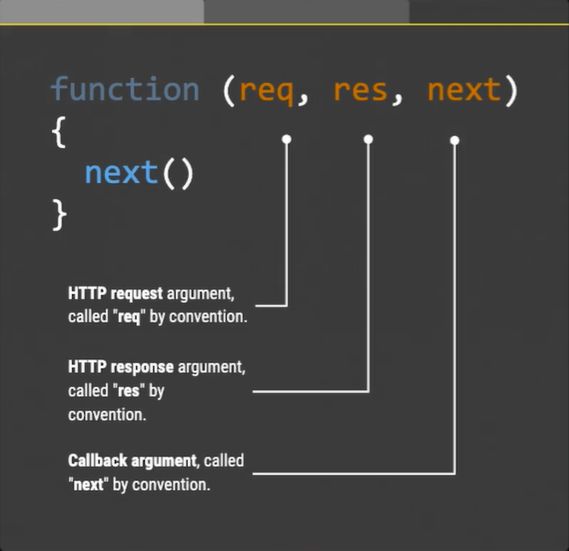
Ejemplo de Implementación de la capa de servicios en un proyecto:



# ¿Qué es un middleware? Capa de manejo de errores usando un middleware.

Un **middleware** es una pieza de software que está en medio de otras 2, se le suele describir como **software glue**, es decir pegamento de software y es porque nos ayuda a conectar otras piezas de software, pensemos por un momento en la población y en el agua, que es un recurso natural, si queremos que esté recurso natural llegue a la población deberíamos insertar en el medio, en este caso un middleware que sería un sistema de tuberías, el sistema de tuberías nos permite conectar el agua a la población, pero nosotros podemos seguir agregando middlewares, podemos agregar un middleware que se encargue de purificar el agua y luego podríamos poner otro middleware que se encargue de contar el consumo del agua.

**En express** específicamente, la manera en cómo funcionan los **middlewares** es mediante la firma del: **request-object, response-object y la funcionalidad next**.



# 

**El orden de los middleware importa**

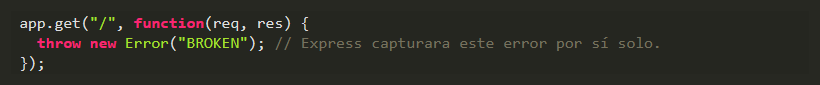
# 

# 

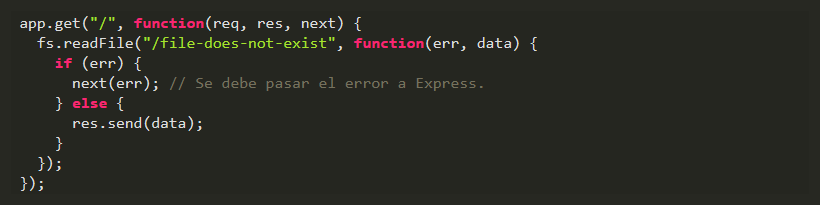
# Manejo de errores asíncronos y síncronos en Express .

El manejo de errores en Express es el proceso de capturar un error de manera asíncrona como síncrona . Por defecto Express viene con un manejador de errores por defecto, así que no es necesario escribir uno para empezar a usarlo.

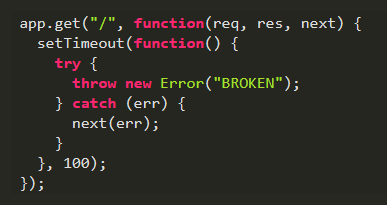
Los errores que ocurren de manera síncrona dentro un manejador de rutas o un *middleware* no requieren trabajo extra. Si un código síncrono lanza un error Express automáticamente capturará el error. Por ejemplo:



Para errores que se retornan desde funciones asíncronas invocadas desde un manejador de ruta o un middleware, es necesario pasar el error como argumento a la función next(), de esta manera Express capturará el error y lo procesará. Por ejemplo:

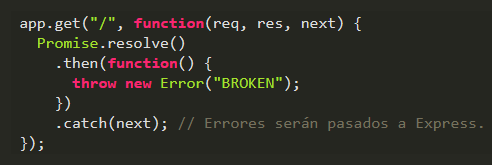


Es responsabilidad de nosotros capturar errores que puedan ocurrir en código asíncrono invocado desde un manejador de ruta o middleware para que Express lo procese. Por ejemplo:



El ejemplo de arriba usa un bloque try...catch para capturar los errores en el código asíncrono y pasarlo a Express. Si el bloque try...catch fuese omitido, Express no podría capturar el error debido a que no es parte de un manejador síncrono de código.

Cuando se usan funciones que retornan promesas, puedes simplemente proveer la funcionalidad **next** al final del manejador catch de la promesa y Express automáticamente capturará el error. Por ejemplo:

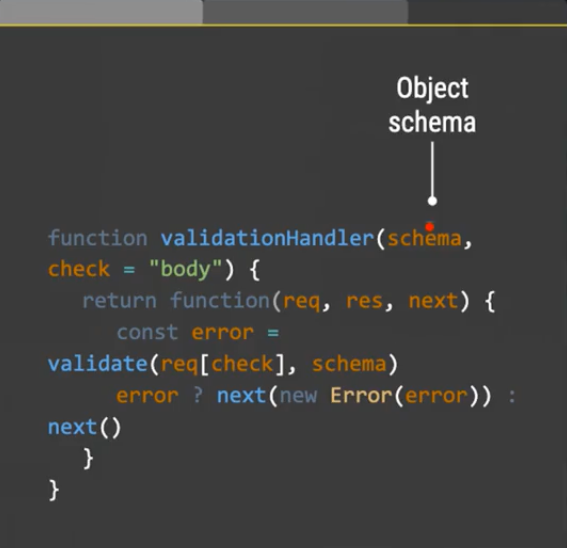


## **Capa de validación de datos a través de un middleware.**

La capa de validación es necesaria porque cuando estamos **consumiendo** nuestros **endpoints**, los endpoints no saben que datos le estamos enviando, si estamos enviando en el caso de una película con título o sin título, o a la hora de recibir las películas el formato del id es correcto, entonces mediante este **middleware podemos validar o podemos especificar unos schemas**, de tal manera que podemos especificar que: la pelicula sea de tipo string, queremos que la numeración sea de tipo número, etc. De está manera si por alguna razón el cliente o el usuario ingrese los datos que no son, podemos especificar un error mostrando que partes son requeridas o que partes no tienen el formato, esto le va a ayudar al cliente o al usuario a que se dé cuenta como debe consumir nuestras API.

Para ello vamos a crear una funcionalidad que se va a llamar **validationHandler**, está funcionalidad va a **recibir** un **schema**, el esquema va a determinar cuál es el formato del esquema que vamos a enviar, **como segundo parámetro** le vamos a informar **donde queremos checar este esquema**, por defecto vamos a checar el **body del request**, pero también podríamos checar los parámetros o el query, etc. Cuando tenemos estos datos en la función retornamos una función que tiene la firma de un middleware, cogemos una utilidad que vamos a llamar validate, en este caso le vamos a pasar en el req **[check]** checando el body, y le vamos a pasar un schema y esto puede que devuelva o no un error de validación, si devuelve un error, lo que vamos a hacer es llamar a nuestra funcionalidad next con el error.

Acordarse que cuando llamamos a nuestra funcionalidad **next** con un parámetro, en este caso el error, van a empezar a ejecutarse los middlewares de error, sino simplemente llamamos a nuestra funcionalidad next y **como no le estamos pasando ningún parámetro va a llamar al siguiente middleware**.



## **¿Qué es Join y Boom?**

Join y Boom son tecnologías del ecosistema de [hapijs](https://hapi.dev/) **pero son tan buenas, que las vamos a integrar como middleware en nuestra aplicación de express**.

**Join**: Object schema validation, es la librería que nos va a ayudar a validar los schemas. **Boom**: HTPP-friendly error objects. Es un manejador de errores de una manera mucho más comprensible.

La mejor manera de entenderlos es visitando su documentación oficial.

* [Join](https://hapi.dev/family/joi/)
* [Boom](https://hapi.dev/family/boom/?v=8.0.1)

## **Implementando Boom**

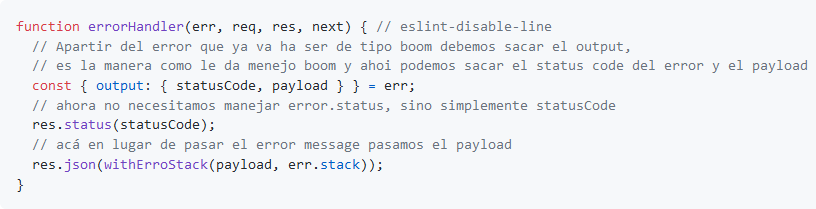
En esta sección aprenderemos a cómo podemos implementar Boom en nuestro código de express. Lo que haremos será usar boom en nuestros errorsHandlers que definimos con anterioridad, también aprovecharemos para crear nuestro error 404 de tal manera que cuando hagan un ***request*** a un ***endpoint*** que no existe, respondamos correctamente, te mostraré como hacerlo en el código.

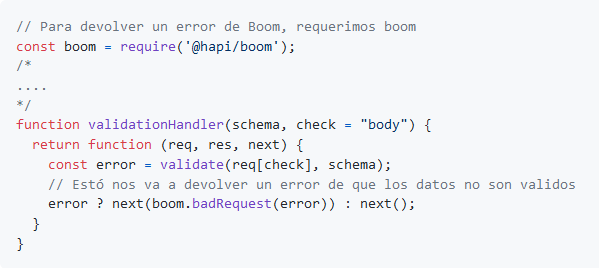
1. primero necesitamos instalar Boom **npm i @hapi/boom**
2. Nos disponemos a ir a nuestro middleware manejador de errores, en el vamos a incluir la dependencia de boom.



1. Como ya nos va a llegar un error boom, vamos a hacer un [spread-operator](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Spread_operator) al error porque ahora el error no solo trae el mensaje, si no que trae unas propiedades como vimos, el **error, statusCode y message**, entonces es necesario hacer esté pequeño cambio
2. Crearemos un tercer middleware que se llamará **wrapErrors()** porque es posible que en algún punto el error que nos llegué no sea de tipo boom y nosotros queremos que apartir de ahí todos los errores tengan la estructura boom.



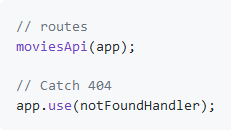
1. Ahora apartir del error que ya va a ser de tipo boom, debemos sacar el output:
2. Ahora tenemos que exportar wrapErrors
3. Ahora lo que debemos hacer es actualizar nuestro archivo index el middleware, en este caso lo que debemos hacer es ponerlo antes del errorHandler.
4. Otra cosa que debemos hacer es ir al validationHandler y donde estabamos devolviendo un error, es devolver un error de Boom



También vamos a crear un middleware para manejar los errores 404, el cual vamos a llamar notFoundHandler.js.

Está función es un **middleware** pero no recibe el next, porque para que pueda funcionar, lo más importante es que esto debe ir al final de las rutas, el notFound lo que hace es que se ejecuta cuando ya paso por todas las rutas

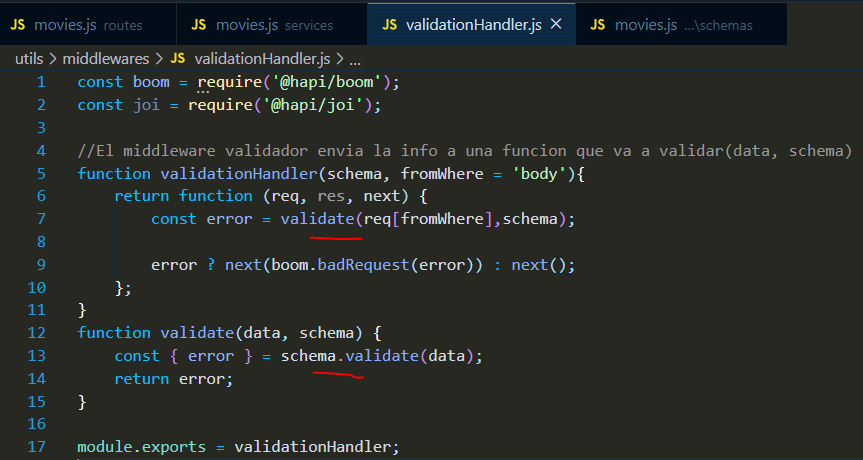
Ahora solo lo requerimos en nuestro archivo index y lo agregamos al final de las rutas.



Ahora si intentamos hacer una llamada con una ruta inexistente nos marcará el error 404 de Boom.

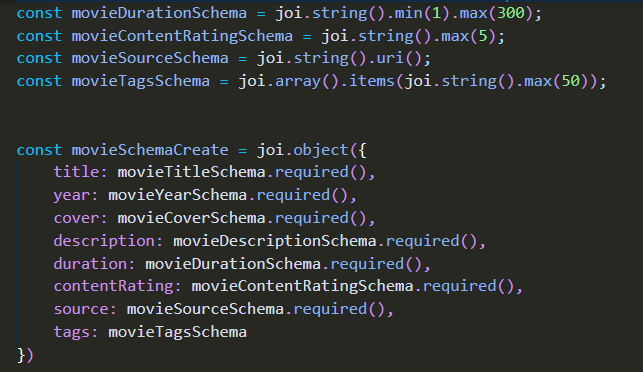
## **Implementando Joi**

En esté modulo vamos a implementar joi con nuestra capa de validación de datos.

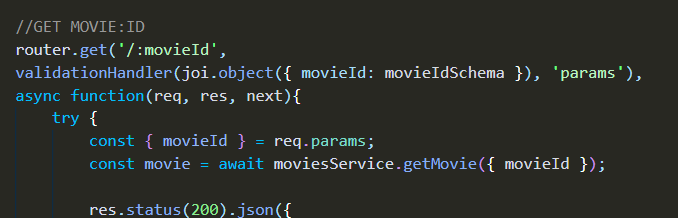
1. Necesitamos instalar **joi** como una dependencia npm i @hapi/joi
2. Vamos a ir a nuestro middleware de validationHandler.
3. Vamos a requerir joi.
4. Ahora lo que debemos de hacer es crear el schema para nuestros datos, en esté caso vamos a crear un schema que valide la estructura de unas películas. Entonces lo que debemos de hacer es crear un schema que por ejemplo:

* A la hora de escribir un string tenga cierto tamaño o cierto mínimo de tamaño y lo mismo para el resto de atributos. **(joi.string().max(80))**
* Si vamos a ocupar en la duración números, tenemos que asegurarnos que sean números. **(.joi.number())**
* Si vamos a usar una url debemos asegurarnos que sea una url. **(joi.string().uri())**

1. Para ello en nuestras utilidades vamos a crear una nueva carpeta que se a llamar schemas, para guardar nuestros esquemas.



Ahora lo que debemos hacer es aplicar estos esquemas en nuestras rutas, para ello debemos ir a nuestras routes movies, y lo primero que debemos hacer es importar los schemas.





Recordar que nuestro **validationHandler** va a **recibir** un **schema** y también **va a recibir de donde va a sacar los datos**, por defecto va a sacarlo del body.

# Middlewares populares en Express.js

## **Body parser**

Analiza los cuerpos de las solicitudes entrantes en un middleware antes que los manejadores de ruta disponibles bajo la propiedad req.body. <http://expressjs.com/en/resources/middleware/body-parser.html>

## **CORS**

Middleware para habilitar CORS (Cross-origin resource sharing) en nuestras rutas o aplicación. <http://expressjs.com/en/resources/middleware/cors.html>

## **Morgan**

Un logger de solicitudes HTTP para Node.js. <http://expressjs.com/en/resources/middleware/morgan.html>

## **Helmet**

Helmet nos ayuda a proteger nuestras aplicaciones Express configurando varios encabezados HTTP. ¡No es a prueba de balas de plata, pero puede ayudar! <https://github.com/helmetjs/helmet>

## **Express Debug**

Nos permite hacer debugging de nuestras aplicaciones en Express mediante el uso de un toolbar en la página cuando las estamos desarrollando. <https://github.com/devoidfury/express-debug>

## **Express Slash**

Este middleware nos permite evitar preocuparnos por escribir las rutas con o sin slash al final de ellas. <https://github.com/ericf/express-slash>

## **Passport**

Passport es un middleware que nos permite establecer diferentes estrategias de autenticación a nuestras aplicaciones. <https://github.com/jaredhanson/passport>

Puedes encontrar más middlewares populares en el siguiente enlace: <http://expressjs.com/en/resources/middleware.html>

## **Considerando las mejores prácticas para el despliegue.**

En este punto ya tu aplicación está lista para el despliegue, pero lo que debemos considerar unas muy buenas prácticas para el lanzamiento a producción.

**Buenas prácticas**:

* Remover contraseñas quemadas
* Encapsular código spaghetti
* Revisar la estructura del proyecto
* Configurar los scripts de build
* Agregar soporte de cache
* Añadir HTTPS y CORS
* Revisar otras prácticas de seguridad.

Aquí hay un lectura para remover los datos confidenciales del historial si te ha pasado: <https://help.github.com/es/articles/removing-sensitive-data-from-a-repository>

El **código spaghetti** es ese código que es muy difícil de leer o crece mucho, lo más recomendable es mover porciones de código a diferentes funciones que tengan mucho significado en esa porción de código en específico.

Scripts de **build**: si tu proyecto tienen algún paso de build, como de pronto minificar o transpilar código también es necesario que esté para que se vayan en código óptimo para producción.

Soporte de cache: muchas veces cuando requerimos las películas, ellas no cambian porque no todo el tiempo se están agregando películas nuevas, entonces es importante tener una capa de cache tipo: en 15 minutos siempre envíame el mismo resultado, de esta manera no tenemos que ir hasta base de datos a traer las películas, sino que las traemos del cache del navegador.

**Https**: la conexiones por https son encriptadas y seguras, si por alguna vez un usuario y contraseña por una conexión que no es https, fácilmente podría agarrar tu usuario y contraseña porque están en texto plano.

**Cors**: sirve para que no todos los clientes se conecten a nuestro backend y no nos hagan cosas maliciosas, lo otro que es importante es que esté revisando las buenas prácticas de seguridad, que explores librerías como Helmet o que revises open web application security project (owasp).

# Variables de entorno, CORS y HTTPS

## **Como usar las variables de entorno para diferente ambientes**

Ya vimos cómo en nuestro ambiente local podemos hacer uso de las variables de entorno usando el archivo .env y la librería dotenv. Generalmente lo que se recomienda es usar el mismo para los diferentes ambientes como **Staging (Pruebas)** y **Producción**.

Para ello se debe acceder al servidor remoto:

1. Duplicar el archivo .env.example y renombrarlo por .env.
2. Cargar las respectivos valores de las variables de entorno.
3. Usar valores y servicios diferentes para cada ambiente, esto quiere decir que las credenciales de desarrollo, staging y producción deben ser completamente diferente.
4. Si se quiere tener un backup de estos valores se recomienda usar las notas seguras de aplicaciones como [1Password](https://1password.com/) o [LastPass](https://www.lastpass.com/).

Como lo hemos dicho antes no se debe hacer commit del archivo .env y este debe estar en el .gitignore, además se recomienda manejar solo un archivo .env. Más información: <https://github.com/motdotla/dotenv#faq>

### **Cuando no es posible acceder al servidor remoto**

Algunos servicios como [Heroku](https://www.heroku.com/) o [Now](https://zeit.co/now) no nos permiten acceder a un servidor remoto pues la administración del servidor es controlada por los mismos servicios, sin embargo cada servicio tiene sus mecanismos para establecer las variables de entorno:

* [Configuración de variables de entorno en Heroku](https://devcenter.heroku.com/articles/config-vars)
* [Configuración de variables de entorno en Now](https://zeit.co/docs/v2/development/environment-variables)

### **Variables de entorno de forma nativa**

El uso del archivo .env junto con la biblioteca dotenv es un mecanismo que nos facilita la configuración de variables de entorno pero si por alguna razón las quisiéramos cargar de manera nativa, es decir desde el sistema operativo recomiendo este tutorial de [Digital Ocean](https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps)

## **Habilitando CORS en producción**

El Intercambio de Recursos de Origen Cruzado (Cross-Origin Resource Sharing) es un mecanismo que agrega unos encabezados (Headers) adicionales HTTP para permitir que un user agent (generalmente un navegador) obtenga permisos para acceder a los recursos de un servidor en un origin distinto (dominio) del que pertenece.

**Por ejemplo una solicitud de origen cruzado seria** hacer una petición AJAX desde una aplicación que se encuentra en <https://dominio-a.com> para cargar el recurso <https://api.dominio-b.com/data.json>.

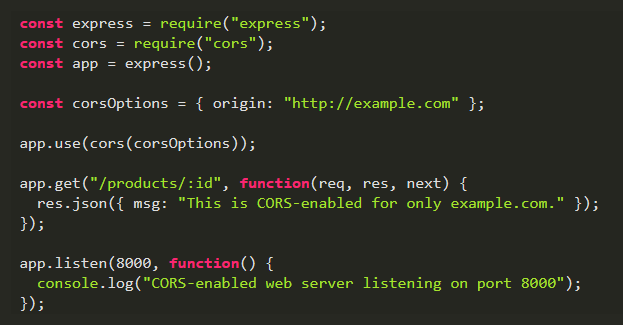
Por razones de seguridad, los navegadores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.

Si necesitamos permitir request desde un dominio diferente al del servidor podemos usar el middleware cors para permitirlo, pero es importante no dejarlo expuesto a todos los dominios.

### **Habilitar CORS para todos los** request **(No recomendado en producción)**

### 

### **Habilitar CORS para los** request **específicos de un cliente (Recomendado para producción)**



Debemos tener en cuenta que para aplicaciones server-side es poco probable que necesiten el uso de CORS debido a que las aplicaciones conviven en el mismo dominio. Sin embargo, es buena práctica habilitarlo para los llamados externos de nuestra API.

Más información sobre el middleware CORS en <https://expressjs.com/en/resources/middleware/cors.html>

## **Cómo funciona y por qué es importante el uso de HTTPS**

El Protocolo Seguro de Transferencia de Hipertexto (HTTPS) es un protocolo HTTP que funciona en el puerto 443 y utiliza un cifrado basado en SSL (Secure Sockets Layer) / TLS (Transmission Layer security) con el fin de crear un canal de comunicación seguro entre el cliente y el servidor.

### **Por qué usar HTTPS**

Una de las razones por la cual siempre debemos usar sitios con HTTPS es que sin este cualquier individuo podría efectuar ataques conocidos como [man-in-the-middle](https://en.wikipedia.org/wiki/Man-in-the-middle_attack) o [eavesdropping](https://en.wikipedia.org/wiki/Eavesdropping) y obtener nuestro usuario y contraseña en el momento en que intentamos acceder a este servicio que no tiene HTTPS establecido.

### **Cómo funciona**

1. El cliente envía un mensaje al servidor y este responde con su certificado público.
2. El cliente comprueba que este certificado sea válido y toma la llave pública.
3. El cliente genera una cadena llamada pre-master secret y la cifra usando la llave pública del servidor y se lo envía.
4. El servidor usa su llave privada para comprobar el pre-master secret.
5. En ese momento tanto el cliente como el servidor usan el pre-master secret para generar un master secret que es usado como una llave simétrica.
6. Teniendo este par de llaves ya se pueden enviar mensajes seguros entre ellos.

### **Cómo habilitar HTTPS en nuestro servidor**

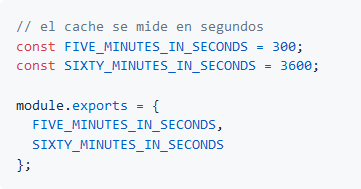
Dependiendo el servicio de hosting que estemos usando puede ofrecernos o no una instalación de certificados de seguridad SSL/TLS que pueden tener algún costo. Sin embargo existen servicios como [Let’s Encrypt](https://letsencrypt.org/) que permiten la instalación de este certificado completamente gratis. Servicios como Now y Heroku ofrecen HTTPS por defecto.

Más información:

* <https://letsencrypt.org/how-it-works/> <https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-on-ubuntu-16-04>
* <https://devcenter.heroku.com/articles/ssl>
* <https://devcenter.heroku.com/articles/automated-certificate-management>
* <https://zeit.co/docs/v1/features/certs>

## **Como implementar una capa de manejo de caché en express**

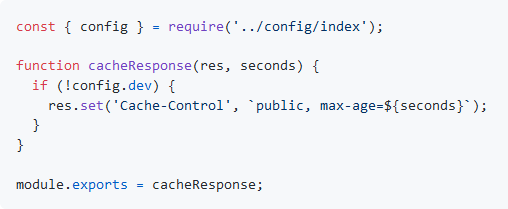
Lo primero que vamos a hacer es crear un archivo en utilidades llamado time.js, donde vamos a establecer unos tiempos.



Teniendo esos tiempos ahora puedo manejar mi utilidad manejo de cache, en la misma carpeta de utilidades, crearemos un archivo que se va llamar **cacheResponse.js**, importamos nuestra configuración porque una de las cosas más molestas de desarrollo es el cache activado, porque muchas veces estamos haciendo modificaciones en nuestro código y notamos que recibimos la misma respuesta y esto es culpa del cache.

Lo más sano para el desarrollador es verificar si no tenemos el cache activado.

Para ello nuestra funcionalidad solo se va a ejecutar response, si no estamos en modo desarrollo. Le vamos a pasar el response porque es el que modificamos para agregar el cache, y los segundos que queremos aplicar.



Ahora debemos ir a nuestras rutas y agregarle cache a las rutas necesarias. **No todas las rutas deben tener cache** las rutas que deben requerir cache deben ser las rutas en las que estamos requiriendo recursos Porque si vamos a crear un recurso, es decir una pelicula nueva, no debería de haber cache porque sucedió en ese momento. Pero si estamos requiriendo películas es posible que agregar películas sea un proceso que no sea inmediato, que lo haga muy de vez en cuando, entonces tiene sentido dejar eso en el cache.

Lo que vamos a hacer es agregar cache solo a la lista de películas y cuando vamos a obtener una lista especifica.

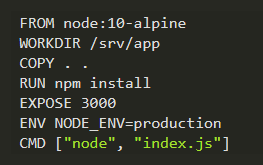


## **¿Cómo contener tu aplicación en Docker?**

Para contener nuestra aplicación en Docker y ejecutarla lo primero es asegurarnos que tenemos instalado Docker.

Podemos seguir las instrucciones para Windows en <https://docs.docker.com/docker-for-windows/install/> o las instrucciones para Mac en <https://docs.docker.com/docker-for-mac/install/>.

Luego lo que debemos hacer es crear un nuevo archivo llamado Dockerfile y en el insertamos el siguiente contenido:



Con el siguiente script creamos una imagen con nuestro de nuestro aplicativo.

docker build -t movies-api

Con el siguiente script podemos ejecutar nuestra imagen en modo detach.

docker run -d movies-api

Si nos dirigimos a http://localhost:3000 deberíamos ver nuestra API funcionando.

Si el contenedor les corre, pero no pueden acceder a la api mediante http://localhost:3000, deténganlo y corran el comando run de la siguiente manera:

docker run -p 3000:3000 -d movies-api

[Instrucciones](https://docs.docker.com/install/linux/docker-ce/ubuntu/) para la instalación de Docker en Ubuntu.

## **Despliegue en now**

En esta sección aprenderás a como desplegar tu aplicación en un servicio llamado now. [now](https://github.com/JasanHdz/backendnodejs/blob/master/notes) es un servicio serverless es decir no tenemos que preocuparnos en infraestructura, ya que a media que tu aplicación tiene cierta demanda, now se encarga de escalar la aplicación por nosotros.

Para hacer uso del servicio de now debemos ir a: <https://zeit.co/download>

Los más recomendable es descargar la aplicación de escritorio pues no solo nos muestra un menú donde podemos ver el status de nuestras descargas, sino que también nos descarga una utilidad de terminal.

La descarga es muy fácil pues NOW CLI se descarga usando npm o yarn. con npm npm i -g now con yarn yarn global add now

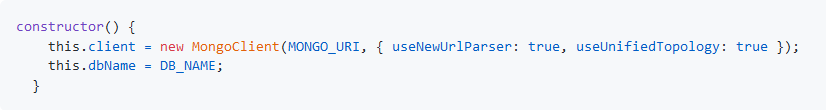
Ahora lo primero que tenemos que hacer es considerar nuestras variables de entorno, pues si instalamos nuestra aplicación si pasárselas al despliegue no van a tener ningún valor, lo que vamos a hacer es que las vamos a sacar del archivo .env.

La manera en cómo now nos permite administrar nuestras variables de entorno, es mediante algo llamado secrets, un secret lo que hace es guardar nuestra variable de entorno y nunca más nos deja acceder a ese resultado, así podemos cuidarnos de que nadie venga nuestra máquina y nos saque el valor de la variable de entorno, la manera en cómo se hace es con now secret add nombreVariableEntorno, y así sucesivamente con todas nuestras variables de entorno.

El archivo now.json que usaremos para el despliegue quedaría de la siguiente manera:



Si al desplegar nuestra aplicación con now dev nos muestra un error cuando se conecta a la base de datos de mongo como:

(node:414) DeprecationWarning: current Server Discovery and Monitoring engine is deprecated, and will be removed in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor. Connected succesfully to mongo Tenemos que agregar el nuevo motor de descubrimiento al monitoreo del servidor. { useUnifiedTopology: true }, lo hacemos de la siguiente manera:

Una vez hecho esto, podemos probar nuestra servidor usando now dev, para que lo ejecute de manera local, y escribimos solo now para enviar el servicio a producción, y cada vez que haya cambios, solo volvemos a hacer now al proyecto.

**Importants Questions and Answers.**

