

Manual de NodeJS



Alejandro Morales
Miguel Angel Alvarez
Jonathan Mircha



desarrolloweb.com/manuales/manual-nodejs.html

Introducción: Manual de NodeJS

Manual de NodeJS, donde a lo largo de diversos capítulos iremos viendo como trabajar esta una plataforma para el desarrollo de aplicaciones de propósito general, con Javascript como lenguaje.

NodeJS es una tecnología que se apoya en el motor de Javascript V8 para permitir la ejecución de programas hechos en Javascript en un ámbito independiente del navegador. A veces se hace referencia a NodeJS como Javascript del lado del servidor, pero es mucho más.

La característica más importante de NodeJS, y que ahora otra serie de lenguajes están aplicando, es la de no ser bloqueante. Es decir, si durante la ejecución de un programa hay partes que necesitan un tiempo para producirse la respuesta, NodeJS no detiene el hilo de ejecución del programa, esperando que esa parte acabe, sino que continúa procesando las siguientes instrucciones. Cuando el proceso lento termina, entonces realiza las instrucciones que fueran definidas para realizar con los resultados recibidos.

Esa característica, y otras que veremos en el Manual de NodeJS, hace que el lenguaje sea capaz de atender muchas peticiones, rápidamente, con pocos recursos. Del mismo modo, hace muy apropiado su uso en entornos de servidores web e Internet en general.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-nodejs.html>

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



Alejandro Morales Gámez

Desarrollador web, especialista en JavaScript, Node.js, lua y Ruby.



Jonathan MirCha

Jonathan es apasionado del desarrollo web, especialista en Javascript, HTML5 o NodeJS. Fundador de Bextlan e instructor en EscuelaIT y otras organizaciones. Comparte su afición al mundo de la web con la carrera como maratonista.



Introducción y primeros pasos en NodeJS

En los primeros artículos de este manual ofrecemos una introducción a la plataforma Node, revisando su historia y algunas de sus características que lo hacen especial en relación a otros lenguajes de programación. Además veremos unos ejemplos sencillos con los que comenzar a experimentar el desarrollo con NodeJS.

Introducción a NodeJS

Una inmersión teórica en NodeJS, plataforma para el desarrollo con Javascript del lado del servidor. Qué es node, quiénes lo están usando y por qué es una buena idea aprender NodeJS.

Con este artículo comienza el [Manual de NodeJS](#) de DesarrolloWeb.com, que os llevará a lo largo de diversas entregas en vuestros primeros pasos con esta plataforma para desarrollo. Será una serie de artículos que estamos preparando en base a la transcripción de los webcasts #nodeIO, emitidos en DesarrolloWeb.com.

De momento, en los primeros pasos nos encontramos en la [Introducción a NodeJS realizada en el primero de los programas de la serie](#), transmitidos en directo, por *hangout*, en el canal de Youtube de Desarrolloweb.com. Tenemos que agradecer la presentación a Alejandro Morales @_alejandromg desarrollador web, entusiasta del *Open Source* y experimentado programador en NodeJS.



NodeJS, conocido habitualmente también con la palabra "node" a secas, surge en 2009 como respuesta a algunas necesidades encontradas a la hora de desarrollar sitios web, específicamente el caso de la concurrencia y la velocidad.

NodeJS es un plataforma super-rápida, especialmente diseñada para realizar operaciones de entrada / salida (Input / Output o simplemente I/O en inglés) en redes informáticas por medio de distintos protocolos, apegada a la [filosofía UNIX](#). Es además uno de los actores que ha provocado, junto con HTML5, que Javascript gane gran relevancia en los últimos tiempos,

pues ha conseguido llevar al lenguaje a nuevas fronteras como es el trabajo del lado del servidor.

En este artículo pretendemos explicar qué es Node, para qué se utiliza, por qué es bueno aprenderlo ya y algunos de los proyectos más relevantes creados con esta tecnología, y que muchos de nosotros conocemos.

¿Qué es NodeJS?

"Node Yei es", tal como se pronuncia NodeJS en inglés, es básicamente un *framework* para implementar operaciones de entrada y salida, como decíamos anteriormente. Está basado en eventos, *streams* y construido encima del motor de Javascript V8, que es con el que funciona el Javascript de Google Chrome. A lo largo de este artículo daremos más detalles, pero de momento nos interesa abrir la mente a un concepto diferente a lo que podemos conocer, pues NodeJS nos trae una nueva manera de entender Javascript.

Si queremos entender esta plataforma, lo primero que debemos de hacer es desprendernos de varias ideas que los desarrolladores de Javascript hemos cristalizado a lo largo de los años que llevamos usando ese lenguaje. Para empezar, NodeJS se programa del lado del servidor, lo que indica que los procesos para el desarrollo de software en "Node" se realizan de una manera muy diferente que los de Javascript del lado del cliente.

De entre alguno de los conceptos que cambian al estar Node.JS del lado del servidor, está el asunto del "Cross Browser", que indica la necesidad en el lado del cliente de hacer código que se interprete bien en todos los navegadores. Cuando trabajamos con Node solamente necesitamos preocuparnos de que el código que escribas se ejecute correctamente en tu servidor. El problema mayor que quizás podamos encontrarnos a la hora de escribir código es hacerlo de calidad, pues con Javascript existe el habitual problema de producir lo que se llama "código espagueti", o código de mala calidad que luego es muy difícil de entender a simple vista y de mantener en el futuro.

Otras de las cosas que deberías tener en cuenta cuando trabajas con NodeJS, que veremos con detalle más adelante, son la programación asíncrona y la programación orientada a eventos, con la particularidad que los eventos en esta plataforma son orientados a cosas que suceden del lado del servidor y no del lado del cliente como los que conocemos anteriormente en Javascript "común".

Además, NodeJS implementa los protocolos de comunicaciones en redes más habituales, de los usados en Internet, como puede ser el HTTP, DNS, TLS, SSL, etc. Mención especial al protocolo SPDY, fácilmente implementado en Node, que ha sido desarrollado mayoritariamente por Google y que pretende modernizar el protocolo HTTP, creando un sistema de comunicaciones que es sensiblemente más rápido que el antiguo HTTP (apuntan un rendimiento 64% superior).

Otro aspecto sobre el que está basada nodeJS son los "streams", que son flujos de datos que están entrando en un proceso. Lo veremos con detalle más adelante.

¿Quién usa NodeJS?

Existen varios ejemplos de sitios y empresas que ya están usando Node en sitios en producción y algunos casos de éxito que son realmente representativos. Quizás el más comentado sea el de LinkedIn, la plataforma de contacto entre profesionales a modo de red social. Al pasar a NodeJS, LinkedIn ha reducido sensiblemente el número de servidores que tenían en funcionamiento para dar servicio a sus usuarios, específicamente de 30 servidores a 3.

Lo que sí queda claro es que NodeJS tiene un *footprint* de memoria menor. Es decir, los procesos de NodeJS ocupan niveles de memoria sensiblemente menores que los de otros lenguajes, por lo que los requisitos de servidor para atender al mismo número de usuarios son menores. Por aproximar algo, podríamos llegar a tener 1.000 usuarios conectados a la vez y el proceso de NodeJS ocuparía solamente 5 MB de memoria. Al final, todo esto se traduce en que empresas grandes pueden tener un ahorro importante en costes de infraestructura.

Otros ejemplos, además de LinkedIn son eBay, Microsoft, empresas dedicadas a *hosting* como Nodester o Nodejitsu, redes sociales como Geekli.st, y muchos más. Podemos obtener más referencias acerca de casos de uso y empresas que implementan NodeJS en el enlace nodeknockout.com que es un *hackaton* donde se realizaron aplicaciones en Node.

Por qué Node.JS es una tecnología que se puede usar ya mismo

NodeJS es una plataforma reciente y que ha sufrido muchos cambios a lo largo de su creación. De hecho, en el momento de escribir este artículo aún no se ha presentado la *release* 1.0, por lo que muchos desarrolladores la han tomado en cuenta con cierta distancia. Actualmente se encuentra a disposición la versión 0.8.15.

Inicialmente, es cierto que ha sufrido bastantes modificaciones, un tanto radicales, en su API, lo que ha obligado a diversos profesionales que apostaron por Node desde un principio a reciclar sus conocimientos rápidamente y rehacer su código en alguna ocasión. Sin embargo, desde hace tiempo han adquirido el compromiso desde NodeJS a no cambiar el API y continuar con la misma arquitectura, realizando solo cambios a nivel interno.

Esto nos hace entender que es un buen momento para aprender NodeJS sin temor a que lo que aprendamos acabe rápidamente en desuso.

Más tecnologías y frameworks basados en NodeJS

No todo termina con NodeJS, en la actualidad existen diversos proyectos interesantes que basan su funcionamiento en Node y que nos dan una idea de la madurez que está adquiriendo esta plataforma. Es el caso de proyectos como:

Meteor JS: Un framework Open Source para crear aplicaciones web rápidamente, basado en programación con "Javascript puro" que se ejecuta sobre el motor de NodeJS.

Grunt: Un conjunto de herramientas que te ayudan como desarrollador web Javascript. Minifica archivos, los verifica, los organiza, etc. Todo basado en línea de comandos.

Yeoman: Otra herramienta, esta vez basada en Grunt, que todavía ofrece más utilidades que ayudan a simplificar diversas tareas en la creación de proyectos, basados en muchas otras librerías y frameworks habituales como Bootstrap, BackboneJS...

Estos son algunos ejemplos que destacó Alejandro, entre muchos otros que hay en Internet. Son programas basados en Node que nos facilitan labores de desarrollo de aplicaciones web.

Conclusión

En este artículo nos hemos podido encontrar una base teórica de NodeJS, que viene muy bien para saber un poco mejor qué es esta tecnología y cuáles son las ventajas que nos puede traer su uso. Está claro que hay muchas otras cosas que querréis saber sobre Node y que te invitamos a seguir descubriendo con nosotros.

En el siguiente artículo explicaremos dónde se obtiene NodeJS y cómo lo instalas. Todo ello a partir del sitio web del propio [NodeJS](#).

Además, te invitamos a seguir viendo el [vídeo #nodeIO introducción a Node.JS](#) del que este texto es una mera transcripción. En dicho vídeo hemos recogido en este texto solamente hasta el minuto 22, por lo que nos queda mucho por ver.

Este artículo es obra de *Alejandro Morales Gámez*
Fue publicado por primera vez en 13/12/2012
Disponible online en <http://desarrolloweb.com/articulos/intro-nodejs.html>

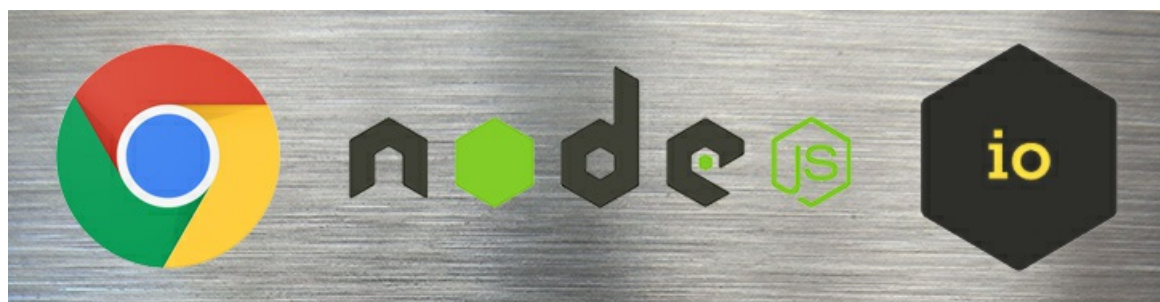
NodeJS 4

El interesante futuro de JavaScript del lado del servidor: NodeJS 4.0.0, versión que une los proyectos Node de Joyent e io.js de la comunidad.

Por si no lo sabes, Node.JS (o simplemente Node) es el entorno de programación JavaScript del lado del servidor, gracias a él desde el 2009 podemos hacer aplicaciones FullStack con un solo lenguaje de programación tanto en el cliente como en el servidor.

Node fue creado en 2009 por Ryan Dahl programador en ese entonces de la empresa Joyent (dedicada a ofrecer servicios de cómputo en la nube) que a su vez se convirtió en la propietaria de la marca Node.js™ y la que le daría patrocinio y difusión desde el momento de su creación.

Joyent puso todo su empeño para el desarrollo de Node, sin embargo, al ser una empresa del sector privado y no una comunidad o fundación, los avances de Node comenzaron a ser muy lentos, en comparación de lo que la comunidad solicitaba y que también quería contribuir.



Con el paso del tiempo y durante los siguientes 5 años (2009 a 2014) se fueron entregando versiones de Node a paso de tortuga pues nunca se llegó como tal a una versión 1.0.0 estable, la última versión de Node bajo la gobernabilidad de Joyent fue la 0.12.7.

La comunidad al ver el brillante futuro que podría tener Node en el ecosistema web y la insuficiencia (o incompetencia) de Joyent para sacarlo adelante, decide tomar cartas en el asunto e intenta contribuir al core de Node, sin embargo Joyent se opone a reconocer cualquier cambio no supervisado por ellos bajo el nombre comercial de Node.js™, por lo que la comunidad decide lanzar un fork amigable de Node compatible con npm y liderado bajo un modelo de gobernabilidad abierta con el nombre de io.js.

Con io.js el entorno de Node ganó mejoras que se venían esperando desde hace muchísimo tiempo tales como:

- Soporte para las últimas versiones de V8 y ES6
- Desarrollo activo haciendo liberaciones semanales
- Integración continua y ejecución de pruebas al 100%
- Vinculación con la comunidad
- Gobernabilidad abierta por la comunidad y no por empresas
- Hoja de ruta predecible
- Versiones compatibles con SemVer
- Comienza a incrementar su actividad nuevamente

La reacción de Joyent, propietario de Node.js™ fue lanzar la versión 0.12.1 que la comunidad estaba esperando desde hacía 2 años, sin embargo y por la premura y sorpresa de io.js salió con un par de errores por lo que tuvieron que lanzar una versión 0.12.2 en cuestión de días. Con ello Joyent logra mantener la paridad de características con io.js. Es importante mencionar que todo esto sucede en diciembre del 2014 y io.js se libera oficialmente el 13 de enero del 2015. Después de un par de meses de la liberación de io.js, ambas partes, comunidad y Joyent manifiestan su intención de reconciliar los proyectos en uno sólo, la fundación Linux se ofrece como mediador de la situación y entonces el 8 de mayo del 2015 se crea la **Node.js Foundation** encargada de reunificar las tecnologías en una sola, dicho proyecto lo llamaron **Node.js Convergence**.

Los objetivos de dicha fundación eran claros:

- Crear un consejo de fundación encargado de la parte legal, marca, mercadotecnia y fondos gestionado por Joyent.
- Crear un comité técnico que actúe de forma independiente y permita el crecimiento del ecosistema JS en el servidor gestionado por la comunidad.
- Normalizar y unificar las versiones actuales de Node.js™ (0.12.X) y io.js (3.X.X) en una sola tecnología.

Finalmente el 8 de septiembre del 2015 sale a la luz esa convergencia en Node 4.0.0 (estable) y con ello un futuro interesante para JavaScript en el servidor.

¿Por qué saltaron de la versión 0.12.X a la 4.0.0?

Desde su nacimiento de io.js se fueron liberando versiones semanales hasta llegar a la versión

3.3.0, en consideración a esta evolución y al fuerte compromiso de la comunidad activa que se matuvo viva y fuerte por hacer crecer el ecosistema de JavaScript en el servidor es que se decide relanzar Node en la versión 4.0.0.

A continuación les dejo una cronología de los hechos más importantes de este suceso:

- Enero 2015: Se libera io.js.
- Febrero 2015: Joyent anuncia la formación de la Fundación Node.js.
- Marzo 2015: Se distribuyen borradores de las reglas y normas de la Fundación Node.js.
- Mayo 2015: io.js obtiene los votos técnicos requeridos para entrar en la Fundación Node.js y comenzar el proceso de convergencia.
- Junio 2015: Lanzamiento oficial de la Fundación Node.js con miembros fundadores.
- Julio 2015: Se establece el comité directivo de la Fundación Node.js.
- Agosto 2015: Se celebra la primer cumbre de colaboración Node.js en San Francisco.
- Septiembre 2015: Se libre de forma estable Node versión 4.0.0.
- Diciembre 2015: La dominación del mundo :).

En las siguientes entregas estaremos revisando y analizando las nuevas características que nos ofrece Node 4.0.0.

Este artículo es obra de *Jonathan MirCha*
Fue publicado por primera vez en 11/09/2015
Disponible online en <http://desarrolloweb.com/articulos/el-futuro-de-nodejs4.html>

Instalar NodeJS

Guía para la instalación del framework para Javascript del lado del servidor, NodeJS.

En este artículo vamos a proceder a la instalación de Node, para tenerlo disponible en nuestro sistema. Dado que NodeJS se ha convertido en una herramienta esencial para desarrolladores, es muy posible que ya lo tengas instalado, ya que mucho de los complementos de desarrollo frontend trabajan con NodeJS. La mejor manera de saber si ya tenemos Node instalado es lanzar el comando en la consola "node -v" y si está instalado nos debería informar la versión que tenemos.

Actualizado en julio de 2016: Realmente instalar NodeJS es un proceso bien sencillo, ya que la plataforma ha evolucionado bastante desde la redacción original de este artículo. Generalmente el proceso es tan elemental como ir a la página de NodeJS donde encontrarás los instaladores para tu sistema operativo. Será solo descargar y ejecutar el instalador en nuestro sistema. Con ello obtendremos directamente "npm" que es el gestor de paquetes que se usa en Node y del que hablaremos más adelante.

Quizás ahora la principal duda que podremos tener es qué versión de NodeJS instalar, ya que en el sitio web nos ofrecen dos alternativas. La recomendación es instalar la última versión, ya que tiene soporte para más cosas y muchas herramientas la requieren.

Si ya tienes Node instalado en tu sistema puedes saltar la lectura de este artículo. También te recomendamos que simplemente accedas al sitio de [Nodejs](https://nodejs.org) y que descargues ese instalador y lo instales por tu cuenta en unos minutos. Aquí podrás ver notas adicionales y alternativas de instalación vía gestores de paquetes para sistemas como Linu o Mac.

Nota: El resto de este artículo es una transcripción del primer *hangout* sobre NodeJS emitido en DesarrolloWeb.com, #nodeIO. En el primer artículo estuvimos conociendo una [introducción teórica a Node.JS](#).



Si no tienes instalado todavía Node.JS el proceso es bastante fácil. Por supuesto, todo comienza por dirigirse a la página de inicio de NodeJS:

nodejs.org

Allí encontrarás el botón para instalarlo "Install" que pulsas y simplemente sigues las instrucciones.

Los procesos de instalación son sencillos y ahora los describimos. Aunque son ligeramente distintos de instalar dependiendo del sistema operativo, una vez lo tienes instalado, el modo de trabajo con NodeJS es independiente de la plataforma y teóricamente no existe una preferencia dada por uno u otro sistema, Windows, Linux, Mac, etc. Sin embargo, dependiendo de tu sistema operativo sí puede haber unos módulos diferentes que otros, ésto es, unos pueden funcionar en Linux y no así en otros sistemas, y viceversa.

Nota: Los módulos no se han nombrado todavía, pero en la práctica se usan constantemente durante el desarrollo en NodeJS. Los entenderás como paquetes de software o componentes desarrollados y que puedes incluir en tu programa para tener soporte a distintas necesidades, como, por ejemplo, comunicaciones mediante un protocolo. A la hora de hacer tu programa puedes incluir esos módulos para acceder a funcionalidades adicionales de Node.

Realmente poco hay que hablar de la instalación, pues es muy sencilla. Debido a ello, nuestro amigo Alejandro Morales @_alejandromg, como ponente de #nodeIO no consideró necesario agrega mucha más información. Sin embargo, para aquellos que necesitan una ayuda adicional

y para agregar algo más de contenido que pueda completar el [Manual de NodeJS](#), ofreceré alguna guía adicional, ya fuera del guión de la charla.

Instalación de NodeJS en Windows

Si estás en Windows, al pulsar sobre Install te descargará el instalador para este sistema, un archivo con extensión "msi" que como ya sabes, te mostrará el típico asistente de instalación de software.

Una vez descargado, ejecutas el instalador y ¡ya lo tienes!

A partir de ahora, para ejecutar "Node" tienes que irte a la línea de comandos de Windows e introducir el comando "node".

Nota: Ya debes saberlo, pero a la línea de comandos de Windows se accede desde el menú de inicio, buscas "cmd" y encuentras el cmd.exe, que abre la línea de comandos. En algunos sistemas Windows anteriores a 7 accedes también desde el menú de inicio, utilizas la opción "Ejecutar" del menú de inicio y escribes "cmd".

Entonces entrarás en la línea de comandos del propio NodeJS donde puedes ya escribir tus comandos Node, que luego veremos.

Instalar NodeJS en Linux

Actualizado: En la página de instalación de NodeJS te ofrecen los comandos para instalar Node en Linux. Son un par de comandos sencillos, pero depende de tu distro, así que es recomendable que te documentes allí. En mi caso quería instalar NodeJS 6 en Ubuntu 16.04. Para ello he usado los siguientes comandos, a ejecutar uno después del otro.

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

Con el primer comando obtienes el instalador y con el segundo comando haces la instalación propiamente dicha. Luego puedes verificar la instalación con el comando "node -v" y te debería salir la versión que se ha instalado en tu máquina.

Notas de instalación sobre Linux desactualizadas: Puedes instalarlo de muchas maneras. Lo más interesante sería bajártelo de los repositorios de tu distribución para que se actualice automáticamente cuando suban nuevas versiones. Pero en mi caso, el centro de software de Ubuntu me ofrece hoy la versión 0.6.12 y a mi me gustaría contar con la 0.8.15, que es la que ofrecen en el sitio oficial de Node.JS.

Para solucionar esta situación podemos usar otros repositorios y para ello te ofrecen en la siguiente referencia instrucciones para instalar Node.JS desde diferentes gestores de paquetes, para las distribuciones más habituales.

github.com

Siguiendo esa referencia, en mi ubuntu he ejecutado los siguientes comandos para instalar la última versión:

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs npm
```

También si lo deseas, desde la página de descargas de Node accederás a los binarios de Linux o al código fuente para compilarlo.

Instalar NodeJS en Mac

Actualizado: Ahora para instalar NodeJS sobre Mac es tan sencillo como acceder a la página de Node y descargar el instalador. Se instala como cualquier otra aplicación, tanto el propio NodeJS como npm.

Alternativa con Homebrew: La instalación en Mac es muy sencilla si cuentas con el gestor de paquetes "homebrew". Es tan fácil como lanzar el comando:

```
brew install nodejs
```

Durante la instalación es posible que te solicite incluir en tu sistema un paquete de utilidades por línea de comandos de xcode, si es que no lo tienes ya instalado en tu OS X. Si se produce un error durante la instalación prueba a hacer un update de homebrew.

```
brew update
```

Probando los primeros comandos NodeJS

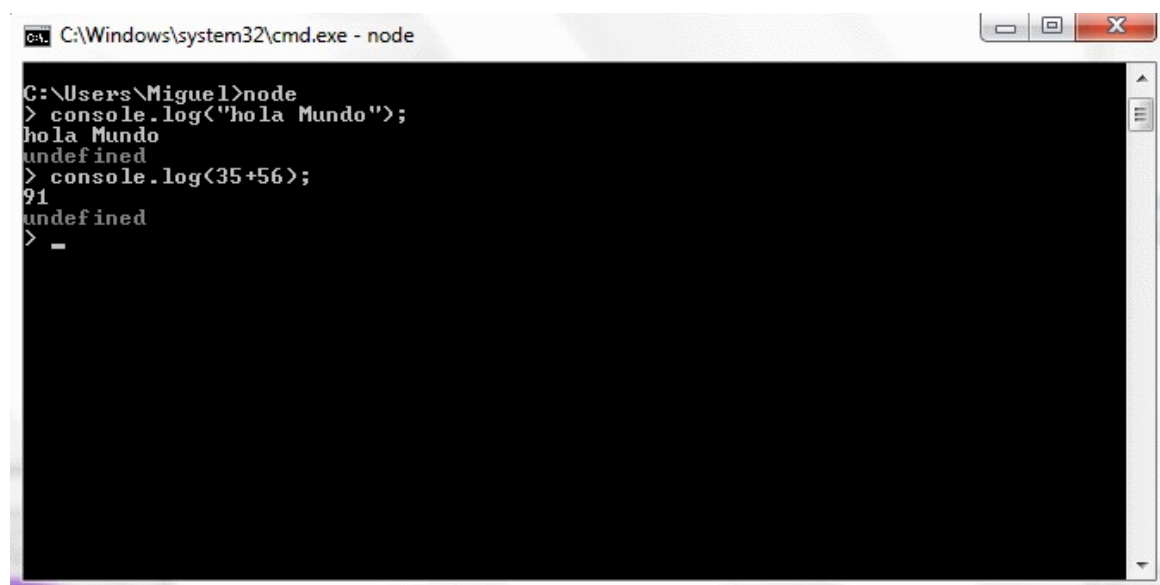
En NodeJS la consola de Node puedes escribir instrucciones Javascript. Si lo deseas, puedes mandar mensajes a la consola con `console.log()` por lo que ésta podría ser una bonita instrucción para comenzar con node:

```
$ node

console.log("hola mundo");
```

Te mostrará el mensaje "hola mundo" en la consola.

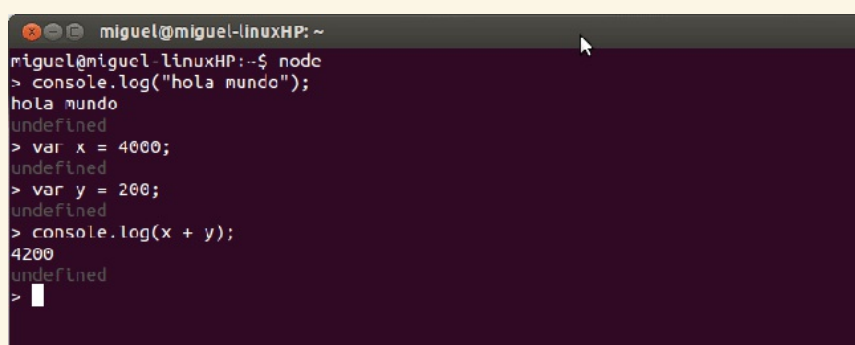
Podemos ver en la siguiente imagen un par de mensajes a la consola de Node que podemos utilizar para comenzar nuestro camino.



```
C:\Windows\system32\cmd.exe - node
C:\Users\Miguel>node
> console.log("hola Mundo");
hola Mundo
undefined
> console.log(35+56);
91
undefined
> -
>
```

Nota: Observarás que te salen unos mensajes "undefined" en la consola y es porque las instrucciones que estamos ejecutando como "console.log()" no devuelven ningún valor.

O en su correspondiente consola de Linux:



```
miguel@miguel-linuxHP: ~
miguel@miguel-linuxHP:~$ node
> console.log("hola mundo");
hola mundo
undefined
> var x = 4000;
undefined
> var y = 200;
undefined
> console.log(x + y);
4200
undefined
> 
>
```

Nota: Para salir de la línea de comandos de Node.JS pulsas CTRL + D o dos veces CTRL + c.

Conclusión

Hay una referencia que el propio @_alejandromg comentó en el primero de los webcast #nodeIO que es Node.js Hispano, donde tienen diversos artículos sobre NodeJS, incluidos varios sobre la [instalación de NodeJS en diversas plataformas](#).

Cualquier información adicional que tengamos sobre la instalación la iremos publicando aquí. Aunque no sea un proceso muy complicado, si tienes otros datos que pienses pueden ayudar, te agradecemos el contacto para completar esta información.

En el siguiente artículo comenzaremos a explicar otras características destacables de NodeJS, con algunos conceptos que te tienen que quedar claros antes de dar tus primeros pasos con la programación.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado por primera vez en 28/12/2012

Disponible online en <http://desarrolloweb.com/articulos/instalar-node-js.html>

Hola Mundo en NodeJS

Mostramos el Hola Mundo en NodeJS y vemos cómo ejecutar algunas instrucciones básicas que están en el core del lenguaje, junto con el flujo de trabajo que usarás para lanzar programas Node.

En el [Manual de NodeJS](#) hemos conocido aspectos básicos sobre el lenguaje. Ahora que lo conocemos un poco mejor y que ya lo tenemos instalado, podemos escribir nuestro primer programa, usando instrucciones básicas del lenguaje.

Otra cosa importante es aprender a trabajar en el flujo habitual de ejecución de aplicaciones Node. Todo esto es bastante elemental, por lo que si ya tienes idea, quizás te interese ya acceder al siguiente artículo con [características de la programación de Node](#), como la asincronía o la programación dirigida por eventos.

Escribir un programa con NodeJS

Los programas en NodeJS se escriben mediante cualquier editor de texto plano. Cualquier editor básico es suficiente, así que cada persona elegirá aquel que prefiera.



Generalmente a los archivos Javascript les pondrás extensión ".js", aunque la verdad podrían tener cualquier extensión que quieras, ya que lo que importa es que tengan texto plano. Lo colocarás en cualquier carpeta de tu sistema y de momento no necesitamos ninguna estructura de directorios, con tener el archivo suelto será suficiente, aunque más adelante cuando tengamos nuestras aplicaciones complejas, con sus módulos de dependencias sí comenzarás a tener una estructura de carpetas bien definida.

El código de nuestro primer programa será tan sencillo como esto:

```
console.log('hola mundo Node!');
```

Lo guardamos en un archivo, por ejemplo "holamundo.js". Es indiferente el nombre del archivo, lo importante es que sepas dónde lo has dejado, porque para ejecutarlo tendrás que situarte en esa carpeta desde tu terminal.

Ejecutar un programa Node

Abrimos el terminal de línea de comandos. Esta operación será dependiente de tu sistema

operativo.

Nota: Aunque imaginamos que debes controlar este aspecto, decir que nos sirve cualquier terminal. Linux y Mac tienen terminales suficientemente buenos para nuestro día a día, pero en Windows muchas veces el terminal que nos viene se quedará pequeño. En Windows, depende de la versión, tenemos un terminal (ejecutable escribiendo "cmd" desde el menú de iniciar aplicaciones) básico que responde a comandos del antiguo MS-DOS. Pero ahora muchas versiones ya tienen instalado "power shell" que está bastante mejor. Yo por mi parte prefiero "git bash" que se instala cuando instalas "Git" en tu sistema. Otras personas prefieren el terminal "cmd".

Nos situamos en la carpeta donde hemos colocado el archivo "holamundo.js".

```
cd mi/proyecto/esta/ruta/debes/saberla/tu
```

Luego lanzamos el proceso de NodeJS indicando el nombre del archivo que vas a ejecutar:

```
node holamundo.js
```

Obviamente, si llamaste al archivo con otro nombre, colocarás ese nombre después de la palabra "node".

Entonces deberías de ver la salida del programa por la misma terminal "hola mundo Node!".

Otras instrucciones básicas del core de Node

No vamos a entrar en muchos detalles de sintaxis en este manual, pues es la misma sintaxis que conoces ya de Javascript. Si no es tu caso y comienzas en NodeJS sin conocer Javascript te recomendamos pasarte por el [Manual de Javascript](#). En la primera parte de ese manual encontrarás todo lo relacionado a la sintaxis, variables, operadores, estructuras de control, etc.

Ahora vamos a ver qué otras cosas simples de lo que ya conoces de Javascript puedes aplicar en Node.

Por ejemplo, podríamos complicar nuestro ejercicio para incluir un bucle:

```
for (var i=0; i<10; i++) {  
  console.log('hola mundo ' + i);  
}
```

Ahora vamos a ver cómo trabajar con alguna de las funciones básicas del API de Javascript, como es el caso del `setTimeout()`, que nos sirve para ejecutar instrucciones con un retardo expresado en milisegundos.

```
setTimeout(function() {  
  console.log('Hola NodeJS');  
}, 3000);
```

Nota: Si te extraña ver una función como parámetro enviado a otra función, lo que se conoce habitualmente como una función "callback", es que quizás te falta un poco de base de Javascript para entrar en NodeJS. Esa función anónima (porque no tiene nombre) se ejecutará solo cuando toque. De nuevo te recomendamos la lectura del Manual de Javascript para ver estas cosas básicas del lenguaje.

Si lo ejecutas verás que tarda 3 segundos antes de mostrar el mensaje "Hola NodeJS".

Parecido a `setTimeout()` tenemos `setInterval()` que ejecuta un código cada un intervalo de tiempo, también en milisegundos.

```
setInterval(function() {  
  console.log('Hola NodeJS');  
}, 1000);
```

En este caso verás que tu consola muestra un mensaje cada segundo, dejando el proceso abierto. Solo cerrando la ventana del terminal, o bien presionando la combinación de teclas CTRL+C, o lo que corresponda en tu sistema, se interrumpirá la ejecución del programa.

Como puedes ver, muchas de las cosas que ya sabes de Javascript se pueden usar en NodeJS. Solo tienes que llevar en consideración que no estás dentro de un navegador, por lo que no tendría sentido acceder a objetos del DOM o del navegador en sí, como "document" o "window".

```
console.log(window);
```

Esa línea te dirá que no existe tal variable "window", con un mensaje como este "ReferenceError: window is not defined".

Algo que tienes en NodeJS a nivel global, parecido a la existencia el objeto "window" en un navegador, es un objeto "global", del que cuelgan diversas propiedades e incluso métodos como los que acabamos de hacer uso, `setTimeout()` y `setInterval()` entre otros.

Puedes hacer un `console.log()` del objeto global para ver su contenido.

```
console.log(global);
```

Ahora que ya sabes que existe ese objeto, la operación de `setInterval()` la podrías haber expresado así:

```
global.setInterval(function() {  
  console.log('Hola NodeJS con global');  
}, 1000);
```

De momento es todo. Practica un poco y continua la lectura por el artículo sobre las [características de la programación en la plataforma Node](#).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 21/07/2016
Disponible online en <http://desarrolloweb.com/articulos/hola-mundo-en-nodejs.html>

Características destacables de NodeJS

Para definir Node.JS mejor viene bien observar algunas características de la plataforma y las diferencias de NodeJS con Javascript común y con otros lenguajes de programación.

Este [Manual de Node.JS](#) está creado a través de unas charlas #nodeIO que hemos realizado en DesarrolloWeb.com, de las cuales estos artículos son una transcripción. En este punto estamos todavía en la primera charla, en la que Alejandro Morales nos está aclarando algunos conceptos importantes para poder entender bien qué es NodeJS.



En el [artículo anterior](#) vimos cómo se instalaba esta NodeJS y en el presente texto veremos en detalle algunas de las características más fundamentales de NodeJS. Entre todas ellas nuestro ponente Alejandro Morales destacó: JavaScript sin limitaciones, Programación asíncrona y Programación orientada a eventos. Todos estos puntos los veremos a continuación.

Un Javascript "sin restricciones"

Con NodeJS tenemos un "Javascript sin restricciones", tal como afirma @_alejandromg, ya que todo se ejecuta en el servidor y no tenemos que preocuparnos de si nuestro código será compatible o no con distintos clientes. Todo lo que escribas en Node JS y te funcione en tu servidor, estarás seguro que funcionará bien, sea cual sea el sistema que se conecte, porque toda la ejecución de código del servidor se queda aislada en el servidor.

Nota: Este detalle de fiabilidad y compatibilidad, o lo que ha llamado el autor como Javascript "sin restricciones" (por estar ejecutado en un ambiente seguro, del lado del servidor) no deja de ser una ventaja de todos los lenguajes del lado del servidor, como PHP, ASP.NET, JSP, etc.

Pero no sólo eso, el Javascript original tiene algunas estructuras de control que realmente no se utilizan en el día a día, pero que realmente existen y están disponibles en NodeJS. En algunas ocasiones resulta especialmente útil alguna de las mejoras de Javascript en temas como la herencia.

Otro ejemplo es que en Javascript haces:

```
for(var key in obj){ }
```

Mientras que en las nuevas versiones de Javascript podrías hacer esto otro:

```
Object.keys(obj).forEach()
```

Programación Asíncrona

Éste es un concepto que algunas personas no consiguen entender a la primera y que ahora toma especial importancia, dado que NodeJS fue pensado desde el primer momento para potenciar los beneficios de la programación asíncrona.

Imaginemos que un programa tiene un fragmento de código que tarda cinco segundos en resolverse. En la mayoría de los lenguajes de programación precedentes, durante todo ese tiempo el hilo de ejecución se encuentra ocupado, esperando a que pasen esos cinco segundos, o los que sea, antes de continuar con las siguientes instrucciones. En la programación asíncrona eres capaz de liberar el proceso de modo que los recursos se quedan disponibles para hacer otras cosas durante el tiempo de espera.

Un ejemplo claro de esto es una llamada a un servicio web, o una consulta a la base de datos. Una vez realizada la solicitud generalmente pasará un tiempo hasta que se obtenga la respuesta. Ese tiempo, por corto que sea, dejaría un proceso esperando en la programación tradicional y en la asíncrona simplemente se libera. En NodeJS, o en Javascript en general, cuando esa espera ha terminado y se ha recibido la respuesta, se retomará la ejecución del código. Para definir las acciones a realizar (código a ejecutar) cuando se haya terminado esa espera, se especifica el código mediante funciones llamadas habitualmente "*callbacks*". Esas funciones contendrán las líneas de código que ejecutar al final de esos procesos de espera, y una vez se ha recibido la respuesta.

La filosofía detrás de Node.JS es hacer programas que no bloqueen la línea de ejecución de código con respecto a entradas y salidas, de modo que los ciclos de procesamiento se queden disponibles durante la espera. Por eso todas las APIs de NodeJS usan callbacks de manera

intensiva para definir las acciones a ejecutar después de cada operación I/O, que se procesan cuando las entradas o salidas se han completado.

Nota: Estos callbacks probablemente ya los hayas usado un montón de veces si tienes experiencia con Javascript del lado del cliente, porque se usan en funciones muy habituales como `setTimeout()`, que también está disponible en NodeJS. Librerías Javascript como jQuery también las usan de manera intensiva.

Por ejemplo miremos este código:

```
console.log("hola");
fs.readFile("x.txt", function(error, archivo){
  console.log("archivo");
})
console.log("ya!");
```

Realmente Javascript es primeramente síncrono y ejecuta las líneas de código una detrás de otra. Por ese motivo, como resultado de ejecución del código anterior, primero veremos el mensaje "hola" en la consola, luego el mensaje "ya!" y por último, cuando el fichero terminó su lectura, veremos el mensaje "archivo".

Por la forma de ejecutarse el código se puede entender la programación asíncrona. La segunda instrucción (que hace la lectura del archivo) tarda un rato en ejecutarse y en ella indicamos además una función con un `console.log("archivo")`, esa es la función callback que se ejecutará solamente cuando termine la lectura del archivo. Por ese detalle, lo último que aparecerá en pantalla es el contenido del fichero. Este detalle es de extrema importancia para entender la programación con NodeJS.

Problema del código piramidal

El uso intensivo de callbacks en la programación asíncrona produce el poco deseable efecto de código piramidal, también conocido habitualmente como "código Espagueti" o "callback hell". Al utilizarse los callbacks, se meten unas funciones dentro de otras y se va entrando en niveles de profundidad que hacen un código menos sencillo de entender visualmente y, por tanto de mantener durante la vida de las aplicaciones.

La solución es hacer un esfuerzo adicional por estructurar nuestro código. Básicamente se trata de modularizar el código, escribiendo cada función aparte e indicando solamente su nombre cuando se define el callback. Podrías incluso definir las funciones en archivos aparte y requiriéndolas con `require("./ruta/al/archivo")` en el código de tu aplicación. Todo esto lo veremos con detalle en numerosos ejemplos en el [Manual de NodeJS](#), por lo que no te debes de preocupar mucho si todavía no lo entiendes.

Al conseguir niveles de indentación menos profundos estamos ordenando el código, con lo que será más sencillo de entender y también más fácil de encontrar posibles errores. Además, a la

larga conseguirás que sea más escalable y puedas extenderlo en el futuro o mantenerlo por cualquier cuestión.

Nota: También en [ES6](#) existen otras herramientas como las [promesas](#) que te ayudan a mantener tu código ordenado con la programación asíncrona y el uso de callbacks

Algunos consejos a la hora de escribir código para que éste sea de mayor calidad:

- Escribe código modularizado (un archivo con más de 500 líneas de código puede que esté mal planteado)
- No abuses, no repitas las mismas cosas, mejor reusa.
- Usa librerías que ayuden al control cuando lo veas necesario (como `async` que te ayuda a ordenar ese montón de callbacks)
- Usa promesas y futuros
- Conoce el lenguaje y mantente al día con las novedades de las presentes y futuras versiones de Javascript [ES6](#), ES7...

Programación orientada a eventos (POE)

Conocemos la programación orientada a eventos porque la hemos utilizado en Javascript para escribir aplicaciones del lado del cliente. Estamos acostumbrados al sistema, que en NodeJS es algo distinto, aunque sigue el mismo concepto.

En Javascript del lado del cliente tenemos objetos como "window" o "document" pero en NodeJS no existen, pues estamos en el lado del servidor.

Eventos que podremos captar en el servidor serán diferentes, como "uncaughtError", que se produce cuando se encuentra un error por el cual un proceso ya no pueda continuar. El evento "data" es cuando vienen datos por un *stream*. El evento "request" sobre un servidor también se puede detectar y ejecutar cosas cuando se produzca ese evento.

Volvemos a insistir, NodeJS sería un equivalente a PHP, JSP, ASP.NET y entonces todo lo que sean eventos de Node, serán cosas que ocurran en el lado del servidor, en diferencia con los eventos de Javascript común que son del lado del cliente y ocurren en el navegador.

Nota: La comparación de NodeJS con lenguajes del lado del servidor viene bien para entender cómo pueden sus eventos ser distintos que los de Javascript del lado del cliente. Sin embargo debemos saber que NodeJS es más bien una *plataforma* basada en Javascript. NodeJS viene con muchas utilidades para programación de propósito general y para realizar programas que implementarían servidores en sí mismos, por lo que su equivalente más próximo podría ser algo como Django o Ruby on Rails. Aunque habría que matizar que esa comparación puede ser un poco atrevida, debido que NodeJS estaría en un nivel mucho más bajo (más cercano a la máquina) y aunque lo podrías usar sin más añadidos para el desarrollo de sitios web, se combina con otros frameworks como ExpressJS para dar mayores facilidades para la creación de sitios aplicaciones web.

Conclusión

Poco a poco vamos entendiendo qué es (y que NO es) NodeJS. Son todo conceptos que resultarán nuevos para muchos de los lectores y por ello era importante dejarlos claros. Ahora vamos a pasar a explicar algunas otras cosas de interés que utilizamos en el día a día en esta plataforma, como el [gestor de paquetes npm](#).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 18/01/2013
Disponible online en <http://desarrolloweb.com/articulos/caracteristicas-nodejs.html>

Programación con NodeJS

En los siguientes artículos abordaremos la programación con Node, no tanto la sintaxis de Javascript que es materia de estudio de otros manuales más básicos, sino más bien su API. Node tiene una extensa colección de módulos disponibles para la creación de programas, que resuelven la más variada gama de necesidades de desarrollo. Además en esta parte del manual iremos repasando diversas características del lenguaje, de un modo práctico, con las cuales entender mejor cómo se desarrolla en la plataforma NodeJS.

El proceso de ejecución de NodeJS

Cómo es el proceso de ejecución de NodeJS y qué podemos hacer para controlarlo mediante el objeto global process.

Una de las primeras cosas que vamos a aprender en NodeJS es trabajar y entender el proceso de ejecución del programa, que tiene sus particularidades especiales en este lenguaje. En los capítulos de introducción en el [Manual de NodeJS](#) ya hemos abordado algunos conceptos bastante interesantes, pero ahora lo veremos aplicado a código.

Antes de comenzar queremos volver a recalcar la naturaleza de NodeJS como un lenguaje de programación de propósito general, con el que podemos hacer todo tipo de aplicaciones. Normalmente comenzaremos con programas que se ejecutarán en la consola de comandos, o terminal, de tu sistema operativo, pero a partir de ahí sus aplicaciones son muy extensas.

Piensa entonces que NodeJS se arranca mediante la consola de comandos y entonces el proceso que se encargará de ejecutarlo es el iniciado con la propia consola.



Single Thread (único hilo)

Una de las características de NodeJS es su naturaleza "Single Thread". Cuando pones en marcha un programa escrito en NodeJS se dispone de un único hilo de ejecución.

Esto, en contraposición con otros lenguajes de programación como Java, PHP o Ruby, donde cada solicitud se atiende en un nuevo proceso, tiene sus ventajas. Una de ellas es que permite atender mayor demanda con menos recursos, uno de los puntos a favor de NodeJS.

Para conseguir que la programación Single Thread de NodeJS pueda producir resultados satisfactorios debemos entender su característica no bloqueante, que abordamos al hablar de la programación asíncrona en el artículo de las [características de NodeJS](#). En resumen, todas las operaciones que NodeJS no puede realizar al instante, liberan el proceso se libera para atender otras solicitudes. Pero como hemos dicho en este artículo nos proponemos aterrizar estas ideas en algo de código que nos permita ir aprendiendo cosas nuevas de NodeJS.

Nota: Cabe aclarar que el single thread no implica que Node no pueda disponer de varios hilos de manera interna para resolver sus problemas. Es decir, nuestro hilo principal por ejemplo cuando estamos desarrollando un servidor con Node podrá estar atento a solicitudes, pero una vez que se atiendan, Node podrá levantar de manera interna otros procesos para realizar todo tipo de acciones que se deban producir como respuesta a esas solicitudes. También será posible si lo deseamos disponer de diversos programas ejecutados en procesos distintos, o clones de un mismo proceso para atender diversas solicitudes más rápidamente.

Objeto process

El objeto process es una variable global disponible en NodeJS que nos ofrece diversas informaciones y utilidades acerca del proceso que está ejecutando un script Node. Contiene diversos métodos, eventos y propiedades que nos sirven no solo para obtener datos del proceso actual, sino también para controlarlo.

Nota: El hecho de ser un objeto global quiere decir que lo puedes usar en cualquier localización de tu código NodeJS, sin tener que hacer el correspondiente require().

En la documentación de Node encuentras todo lo que hay en este objeto process:

<https://nodejs.org/docs/latest/api/process.html>

Nota: La documentación de NodeJS está organizada por versiones de la plataforma. El link anterior te dirige a la última versión del API de Node, pero tú deberías consultar los docs de la versión con la que estés trabajando. No obstante, el estado del API para el objeto process está definido desde hace mucho tiempo y difícilmente cambiará.

Consultar datos del proceso

Se pueden hacer muchas cosas con el proceso y consultar diversos datos de utilidad. Por ejemplo, veamos el siguiente código donde podemos examinar información diversa sobre el

proceso actual y la plataforma donde estamos ejecutando este programa:

```
console.log('id del proceso: ', process.pid);
console.log('titulo del proceso: ', process.title);
console.log('versión de node: ', process.version);
console.log('sistema operativo: ', process.platform);
```

Salir de la ejecución de un programa Node

A veces podemos necesitar salir inmediatamente de la ejecución de un programa en NodeJS. Esto lo podemos conseguir mediante la invocación del método `exit()` del objeto `process`.

```
process.exit();
```

Nota: El comportamiento normal de un programa será salir automáticamente cuando haya terminado su trabajo. Técnicamente esto quiere decir, que se haya terminado la secuencia de ejecución de instrucciones de un script y que no haya trabajo pendiente en el "event loop". Sobre el bucle de eventos hablamos en el artículo de los [eventos en Node](#).

Básicamente provocará que el programa acabe, incluso en el caso que haya operaciones asíncronas que no se hayan completado o que se esté escuchando eventos diversos en el programa.

El método `exit` puede recibir opcionalmente un código de salida. Si no indicamos nada se entiende "0" como código de salida.

```
process.exit(3);
```

Evento `exit`

Otra de las cosas básicas que podrás hacer mediante el objeto `process` es definir eventos cuando ocurran cosas diversas, por ejemplo la salida del programa.

Los eventos en Javascript se definen de manera estándar mediante el método `on()`, indicando el tipo de evento que queremos escuchar y una función `callback` que se ejecutará cuando ese evento se dispare. Dado que el evento `exit` pertenece al `process`, lo definiremos a partir de él.

```
process.on('exit', function(codigo) {
  console.log('saliendo del proceso con código de salida', codigo);
})
```

En este caso la función callback asociada al evento exit recibe aquel código de error que se puede generar mediante la invocación del método exit() que conocimos en el anterior punto.

Conclusión

Hemos aprendido algo de conocimiento general sobre NodeJS y algo en particular sobre el proceso de ejecución de un programa escrito en Node, cuya funcionalidad está disponible mediante el objeto global process.

Hemos visto varios ejemplos de propiedades dependientes de process, el método exit(), muy útil para la salida de un programa, así como la definición de uno de los varios eventos disponibles dentro de process, que nos permitirá hacer cosas cuando el programa NodeJS se pare. Con esto tenemos una visión muy general sobre el proceso de ejecución de node, pero comprobarás que hay mucho más en la documentación oficial.

Solo para posibles dudas, dejo aquí un código completo que podrás ejecutar para ver los ejemplos relatados en este artículo:

```
"use strict"

process.on('exit', function(codigo) {
  console.log('saliendo del proceso con código de salida', codigo);
})

console.log('id del proceso: ', process.pid);
console.log('título del proceso: ', process.title);
console.log('versión de node: ', process.version);
console.log('sistema operativo: ', process.platform);

process.exit(3);
console.log('esto no se llegará a ejecutar');
```

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 05/01/2017
Disponible online en <http://desarrolloweb.com/articulos/proceso-ejecucion-nodejs.html>

Módulos y NPM en NodeJS

Qué son los módulos en NodeJS, el gestor de paquetes NPM que permite administrar los módulos y dependencias que necesitamos en un proyecto local.

En el [Manual de NodeJS](#) estamos partiendo como base de las charlas #nodeIO ofrecidas por Alejandro Morales para DesarrolloWeb.com. En este caso todavía nos encontramos viendo contenido de nuestra primera charla, en la que nuestro ponente nos informa de **una de las partes fundamentales en nuestros procesos de desarrollo en NodeJS**, como son los **módulos**. Además, vimos el **gestor de paquetes que viene en el framework**.



En NodeJS el código se organiza por medio de módulos. Son como los paquetes o librerías de otros lenguajes como Java. Por su parte, NPM es el nombre del gestor de paquetes (package manager) que usamos en Node JS.

Si conoces los gestores de paquetes de Linux podrás hacerte una buena idea de lo que es npm, si no, pues simplemente entiéndelo como una forma de administrar módulos que deseas tener instalados, distribuir los paquetes y agregar dependencias a tus programas.

Nota: Por ejemplo, si estás acostumbrado a Ubuntu, el gestor de paquetes que se utiliza es el "apt-get". Por su parte, Fedora o Red Hat usan Yum. En Mac, por poner otro ejemplo, existe un gestor de paquetes llamado Homebrew.

El gestor de paquetes npm, no obstante, es un poquito distinto a otros gestores de paquetes que podemos conocer, porque los instala localmente en los proyectos. Es decir, al descargarse un módulo, se agrega a un proyecto local, que es el que lo tendrá disponible para incluir. Aunque cabe decir que también existe la posibilidad de instalar los paquetes de manera global en nuestro sistema.

Incluir módulos con "require"

Javascript nativo no da soporte a los módulos. Esto es algo que se ha agregado en NodeJS y se realiza con la **sentencia require()**, que está inspirada en la variante propuesta por CommonJS.

La instrucción require() recibe como parámetro el nombre del paquete que queremos incluir e inicia una búsqueda en el sistema de archivos, en la carpeta "node_modules" y sus hijos, que contienen todos los módulos que podrían ser requeridos.

Por ejemplo, si deseamos traernos la librería para hacer un servidor web, que escuche solicitudes http, haríamos lo siguiente:

```
var http = require("http");
```

Existen distintos módulos que están disponibles de manera predeterminada en cualquier proyecto NodeJS y que por tanto no necesitamos traernos previamente a local mediante el

gestor de paquetes npm. Esos toman el nombre de "**Módulos nativos**" y ejemplos de ellos tenemos el propio "http", "fs" para el acceso al sistema de archivos, "net" que es un módulo para conexiones de red todavía de más bajo nivel que "http" (de hecho "http" está mayormente escrito sobre el módulo "net" github.com/joyent/node/blob/master/lib/http.js), "URL" que permite realizar operaciones sobre "url", el módulo "util" que es un conjunto de utilidades, "child_process" que te da herramientas para ejecutar sobre el sistema, "domain" que te permite manejar errores, etc.

Podemos crear fácilmente nuestros módulos exportando las funciones que deseemos

Por supuesto, nosotros también **podemos escribir nuestros propios módulos** y para ello usamos module.exports. Escribimos el código de nuestro módulo, con todas las funciones locales que queramos, luego hacemos un module.exports = {} y entre las llaves colocamos todo aquello que queramos exportar.

```
function suma(a,b){
  return a + b;
}

function multiplicar(a,b){
  return a * b;
}

module.exports = {
  suma: suma,
  multiplicar: multiplicar
}
```

Asumiendo que el archivo anterior se llame "operaciones.js", nosotros podríamos requerirlo más tarde e otro archivo.js de la siguiente manera:

```
var operaciones = require('./operaciones');
operaciones.suma(2,3);
```

Obviamente, esto es solo una toma de contacto y volveremos sobre este asunto más adelante.

Comando npm para gestión de paquetes

Por lo que respecta al uso de npm, es un comando que funciona desde la línea de comandos de NodeJS. Por tanto lo tenemos que invocar con **npm seguido de la operación que queramos realizar**.

```
npm install async
```

Esto instalará el paquete async dentro de mi proyecto. Lo instalará dentro de la carpeta

"node_modules" y a partir de ese momento estará disponible en mi proyecto y podré incluirlo por medio de "require":

```
require("async");
```

Nota: Cabe fijarse en que no hace falta darle la ruta al paquete "async" porque npm me lo ha instalado dentro de node_modules y todo lo que está en esa carpeta se encuentra disponible para require() sin necesidad de decirle la ruta exacta para llegar.

Otras instrucciones posibles de npm son la de publicar paquetes (con "publish"), instalar globalmente (poniendo el flag -g al hacer el "install"), desinstalar, incluso premiar (puntuar paquetes hechos por otras personas), etc.

Podemos ver paquetes a instalar entrando en la página del gestor de paquetes npm: npmjs.org

Por último, sobre npm se mencionó que cada paquete tiene entre su código un archivo package.json que contiene en notación JSON los datos del paquete en sí. Es como una tarjeta de identificación del paquete, que puede servir para informarte a ti mismo y a cualquier sistema informático de sus características. Si tú fueras el creador de un paquete, o crearas alguna aplicación con NodeJS, también deberías incluir "package.json" con los datos del módulo o aplicación que se está creando, como son el autor, versión, dependencias, etc.

En el siguiente artículo vamos a ver algo más específico, que también nos sirva como primer ejemplo para realizar en NodeJS. En concreto veremos el módulo HTTP, con el que podremos hacer un rudimentario servidor web.

Instalar paquetes de manera global con npm

Como se ha dicho antes, npm instala los paquetes para un proyecto en concreto, sin embargo existen muchos paquetes de Node que te facilitan tareas relacionadas con el sistema operativo. Estos paquetes, una vez instalados, se convierten en comandos disponibles en terminal, con los que se pueden hacer multitud de cosas. Existen cada vez más módulos de Node que nos ofrecen muchas utilidades para los desarrolladores, accesibles por línea de comandos, como Bower, Grunt, etc.

Las instrucciones para la instalación de paquetes de manera global son prácticamente las mismas que ya hemos pasado para la instalación de paquetes en proyectos. En este caso simplemente colocamos la opción "-g" que permite que ese comando se instale de manera global en tu sistema operativo.

```
npm install -g grunt-cli
```

Ese comando instala el módulo grunt-cli de manera global en tu sistema.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 25/03/2013
Disponible online en <http://desarrolloweb.com/articulos/modulos-npm-nodejs.html>

Ejemplo Node JS con el módulo HTTP

Ejemplo un poco más práctico de un ejercicio realizado con Node.JS y el módulo HTTP, que sirve para implementar comunicaciones HTTP con NodeJS.

Este es el último bloque de la charla [#nodeIO Introducción a NodeJS](#) emitida en DesarrolloWeb.com por Alejandro Morales. En este caso nos introducimos en un terreno más práctico, que nos permitirá ver un primer ejemplo en Node.JS. Es también el quinto artículo del [Manual de NodeJS](#), en el que ya nos ponemos manos a la obra para lanzarnos a trabajar con código fuente en un primer programita.



Este es el módulo que nos sirve para trabajar con el protocolo HTTP, que es el que se utiliza en Internet para transferir datos en la Web. Nos servirá para crear un servidor HTTP que acepte solicitudes desde un cliente web y servirá como colofón a la introducción a Node.

Referencia: Podemos encontrar la documentación de este módulo en <http://nodejs.org/api/http.html>

Como queremos hacer uso de este módulo, en nuestro ejemplo empezaremos por requerirlo mediante la instrucción "require()".

```
var http = require("http");
```

Nota: Existen otros módulos que crean servidores como es el caso de "net", "tcp" y "tls".

A partir de este momento tenemos una variable `http` que en realidad es un objeto, sobre el que podemos invocar métodos que estaban en el módulo requerido. Por ejemplo, una de las tareas implementadas en el módulo HTTP es la de crear un servidor, que se hace con el módulo "`createServer()`". Este método recibirá un *callback* que se ejecutará cada vez que el servidor reciba una petición.

```
var server = http.createServer(function (peticion, respuesta){
    respuesta.end("Hola DesarrolloWeb.com");
});
```

La función *callback* que enviamos a `createServer()` recibe dos parámetros que son la petición y la respuesta. La petición por ahora no la usamos, pero contiene datos de la petición realizada. La respuesta la usaremos para enviarle datos al cliente que hizo la petición. De modo que "`respuesta.end()`" sirve para terminar la petición y enviar los datos al cliente.

Ahora voy a decirle al servidor que se ponga en marcha. Porque hasta el momento solo hemos creado el servidor y escrito el código a ejecutar cuando se produzca una petición, pero no lo hemos iniciado.

```
server.listen(3000, function(){
    console.log("tu servidor está listo en " + this.address().port);
});
```

Con esto le decimos al servidor que escuche en el puerto 3000, aunque podríamos haber puesto cualquier otro puerto que nos hubiera gustado. Además "`listen()`" recibe también una función *callback* que realmente no sería necesaria, pero que nos sirve para hacer cosas cuando el servidor se haya iniciado y esté listo. Simplemente, en esa función *callback* indico que estoy listo y escuchando en el puerto configurado.

Código completo de servidor HTTP en node.JS

Como puedes ver, en muy pocas líneas de código hemos generado un servidor web que está escuchando en un puerto dado. El código completo es el siguiente:

```
var http = require("http");
var server = http.createServer(function (peticion, respuesta){
    respuesta.end("Hola DesarrolloWeb.com");
});
server.listen(3000, function(){
    console.log("tu servidor está listo en " + this.address().port);
});
```

Ahora podemos guardar ese archivo en cualquier lugar de nuestro disco duro, con extensión `.js`, por ejemplo `servidor.js`.

Nota: Si estamos en Windows podríamos guardar el archivo en una carpeta que se llame node en c:. La ruta de nuestro archivo sería c:/node/servidor.js.

Poner en ejecución el archivo con Node.JS para iniciar el servidor

Ahora podemos ejecutar con Node el archivo que hemos creado. Nos vamos desde la línea de comandos a la carpeta donde hemos guardado el archivo servidor.js y ejecutamos el comando "node" seguido del nombre del archivo que pretendemos ejecutar:

```
node servidor.js
```

Entonces en consola de comandos nos debe aparecer el mensaje que informa que nuestro servidor está escuchando en el puerto 3000.

El modo de comprobar si realmente el servidor está escuchando a solicitudes de clientes en dicho puerto es acceder con un navegador. Dejamos activa esa ventana de línea de comandos y abrimos el navegador. Accedemos a:

<http://localhost:3000>

Entonces nos tendría que aparecer el mensaje "Hola DesarrolloWeb.com".

Conclusión sobre la introducción sobre NodeJS

Con este ejemplo en funcionamiento hemos completado material divulgado en el webcast #nodeIO de introducción a NodeJS. La verdad es que, puesto en palabras, impresiona la cantidad de información que nos pasó Alejandro Morales @_alejandromg, al que mando mi más sincero agradecimiento y enhorabuena por esa magnífica exposición.

En siguientes webcast continuaremos hablando de NodeJS y realizando otros ejemplos desde cero que nos puedan ir soltando en el mundo "Node".

Acabo con un par de recomendaciones del propio Alejandro:

- Aprende Javascript para poder pasar luego a NodeJS con garantías. Es más importante el hecho de dominar el propio Javascript que tener idea de otros lenguajes de programación del lado del servidor.
- Si tienes una base suficiente de Javascript, destina al menos una semana, unas 40 horas, para aprender NodeJS y realizar algún ejemplo interesante. muéstralo a tus amigos y promociona lo que has aprendido y estarás ayudando a la comunidad de NodeJS, divulgando esta tecnología.

Conclusión

Continuaremos a partir de aquí comentando algunos otros actores de la programación sobre NodeJS, como los eventos y los *stream* y realizando ejemplos un poco más complejos. Si quieres ir aprendiendo las cosas que veremos en los próximos artículos, te recomiendo ver el segundo *webcast* [#nodeIO dedicado a los eventos, streams y más](#).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 05/04/2013
Disponible online en <http://desarrolloweb.com/articulos/ejemplo-nodejs-modulo-http.html>

Eventos en NodeJS

Los eventos en NodeJS, cómo se implementan y qué características tienen los eventos Javascript del lado del servidor.

En el [Manual de NodeJS](#) hemos presentado ya informaciones bastante amplias para empezar a dar los primeros pasos con esta plataforma de desarrollo. Aún nos quedan muchas cosas importantes y básicas por conocer, como los eventos.



En este artículo explicaremos qué son los eventos del lado del servidor y cómo se implementan en Node.JS. Por si os interesa, es importante comentar que este texto está extractado del segundo programa #nodeIO emitido en directo en Desarrolloweb.com por Alejandro Morales y Miguel Angel Alvarez: [#nodeIO eventos, streams y más.](#)

Eventos del lado del servidor con Javascript

Lo primero que debemos entender es qué son eventos del lado del servidor, que no tienen nada que ver con los eventos Javascript que conocemos y utilizamos en las aplicaciones web del lado del cliente. Aquí los eventos se producen en el servidor y pueden ser de diversos tipos dependiendo de las librerías o clases que estemos trabajando.

Para hacernos una idea más exacta, pensemos por ejemplo en un servidor HTTP, donde tendríamos el evento de recibir una solicitud. Por poner otro ejemplo, en un *stream* de datos tendríamos un evento cuando se recibe un dato como una parte del flujo.

Módulo de eventos

Los eventos se encuentran en un módulo independiente que tenemos que requerir en nuestros programas creados con Node JS. Lo hacemos con la sentencia "require" que conocimos en artículos anteriores cuando hablábamos de [módulos](#).


```
var eventos = require('events');
```

Dentro de esta librería o módulo tienes una serie de utilidades para trabajar con eventos.

Veamos primero el emisor de eventos, que encuentras en la propiedad EventEmitter.

```
var EmisorEventos = eventos.EventEmitter;
```

Nota: Ese "EmisorEventos" es una clase de programación orientada a objetos (POO), por eso se le ha puesto en el nombre de la clase la primera letra en mayúscula. Por convención se hace así con los nombres de las clases en POO.

Cómo definir un evento en NodeJS

En "Node" existe un bucle de eventos, de modo que cuando tú declares un evento, el sistema se queda escuchando en el momento que se produce, para ejecutar entonces una función. Esa función se conoce como "callback" o como "manejador de eventos" y contiene el código que quieres que se ejecute en el momento que se produzca el evento al que la hemos asociado.

Primero tendremos que "instanciar" un objeto de la clase EventEmitter, que hemos guardado en la variable EmisorEventos en el punto anterior de este artículo.

```
var ee = new EmisorEventos();
```

Luego tendremos que usar el método on() para definir las funciones manejadoras de eventos, o su equivalente addEventListener(). Para emitir un evento mediante código Javascript usamos el método emit().

Nota: Como veremos, se encuentran muchas similitudes a la hora de escribir eventos en otras librerías Javascript como jQuery. El método on() es exactamente igual, al menos su sintaxis. El método emit() sería un equivalente a trigger() de jQuery.

Por ejemplo, voy a emitir un evento llamado "datos", con este código.

```
ee.emit('datos', Date.now());
```

Ahora voy a hacer una función manejadora de eventos que se asocie al evento definido en "datos".

```
ee.on('datos', function(fecha){  
  console.log(fecha);  
});
```

Si deseamos aprovechar algunas de las características más interesantes de aplicaciones NodeJS quizás nos venga bien usar `setInterval()` y así podremos estar emitiendo datos cada cierto tiempo:

```
setInterval(function(){  
  ee.emit('datos', Date.now());  
}, 500);
```

Código completo del ejemplo de eventos y ejecución

Con esto ya habremos construido un ejemplo NodeJS totalmente funcional. El código completo sería el siguiente:

```
var eventos = require('events');  
  
var EmisorEventos = eventos.EventEmitter;  
var ee = new EmisorEventos();  
ee.on('datos', function(fecha){  
  console.log(fecha);  
});  
setInterval(function(){  
  ee.emit('datos', Date.now());  
}, 500);
```

Esto lo podemos guardar como "eventos.js" o con cualquier otro nombre de archivos que deseemos. Lo guardamos en el lugar que queramos de nuestro disco duro.

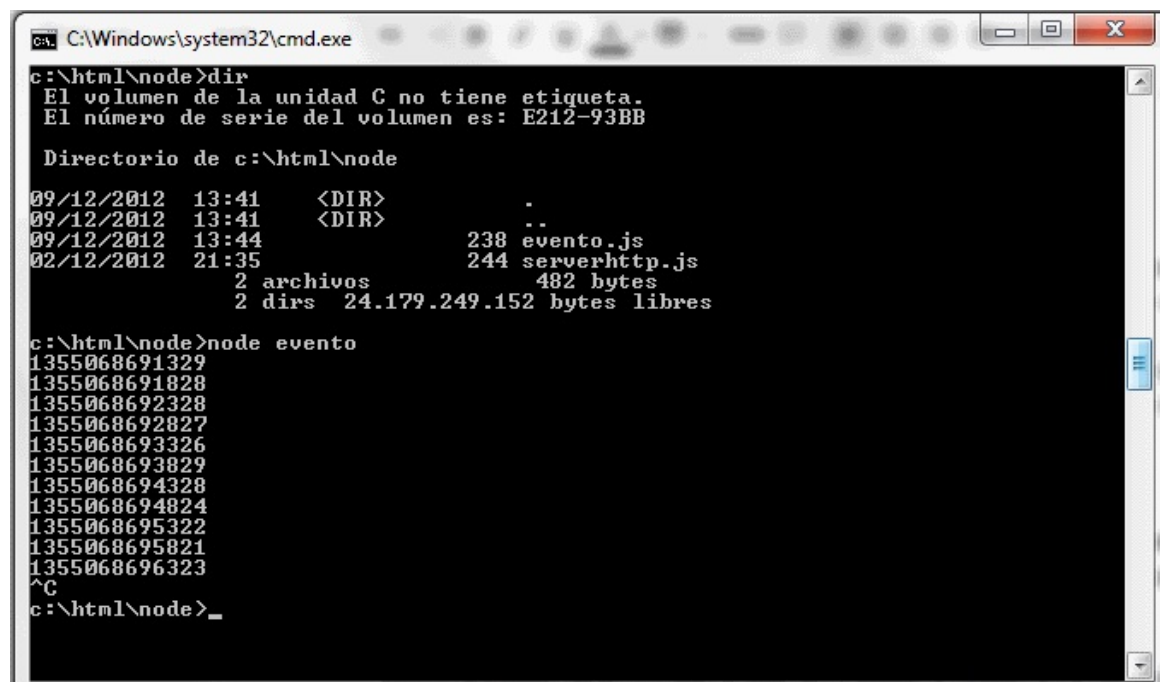
Para ponerlo en ejecución nos vamos en línea de comandos hasta la carpeta donde hayamos colocado el archivo "eventos.js" o como quiera que lo hayas llamado y escribes el comando:

```
node eventos.js
```

Nota: También podrías haber ejecutado el comando "node eventos" sin indicar la extensión ".js" porque NodeJS ya da por supuesto que le has colocado esa extensión al archivo.

Como resultado, veremos que empiezan a aparecer líneas en la ventana del terminal del sistema operativo, con un número, que es el "timestamp" de la fecha de cada instante inicial. Puedes salir del programa pulsando las teclas CTRL + c.

Por si te lía esto de ejecutar archivos por medio de la línea de comando, a continuación puedes ver una pantalla del terminal donde hemos puesto en marcha este pequeño ejercicio de eventos.



```
C:\Windows\system32\cmd.exe
c:\html\node>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: E212-93BB

Directorio de c:\html\node

09/12/2012  13:41    <DIR>          .
09/12/2012  13:41    <DIR>          ..
09/12/2012  13:44                238 evento.js
02/12/2012  21:35                244 serverhttp.js
                2 archivos                482 bytes
                2 dirs 24.179.249.152 bytes libres

c:\html\node>node evento
1355068691329
1355068691828
1355068692328
1355068692827
1355068693326
1355068693829
1355068694328
1355068694824
1355068695322
1355068695821
1355068696323
^C
c:\html\node>
```

Conclusión

Con esto hemos podido obtener una primera aproximación a lo que son los eventos en NodeJS y la manera de crear esas funciones "callback" para ejecutar cuando se producen, también llamadas manejadores de eventos.

En los siguientes capítulos pasaremos a hablaros de los streams, que son bastante más importantes en el mundo de NodeJS.

Este artículo es obra de *Alejandro Morales Gámez*
Fue publicado por primera vez en 30/07/2013
Disponible online en <http://desarrolloweb.com/articulos/eventos-nodejs.html>

Buffer en NodeJS

Qué son buffer en NodeJS, junto con algunos ejemplos simples del trabajo con buffers.

Los buffer son conjuntos de datos en crudo, datos binarios, que podemos tratar en NodeJS para realizar diversos tipos de acciones. Los implementa Node a través de una clase específica llamada Buffer, que era necesaria porque Javascript tradicionalmente no era capaz de trabajar con tipos de datos binarios.

Los buffer son similares a arrays de enteros, en los que tenemos bytes que corresponden con

datos en crudo de posiciones de memoria fuera de la pila de V8. Aunque desde Javascript ES6 con los TypedArray ya es posible trabajar con un buffer de datos binarios, NodeJS mantiene su clase Buffer que realiza un tratamiento de la información más optimizado para sus casos de uso.



Entre los casos de uso más populares de la plataforma NodeJS se encuentran muchos en los que los buffer son verdaderos protagonistas, como por ejemplo en la comunicación bidireccional que tenemos cuando manejamos sockets, pero también al manipular imágenes o streams de datos.

En este artículo veremos una breve introducción a los buffer, abordando su creación y uso.

Crear un Buffer

Para trabajar con buffers debemos apoyarnos en la clase Buffer, que es global en NodeJS, por lo que no necesitas hacer el require de ningún módulo para poder usarla. Ten en cuenta que el tamaño de un buffer se establece en el momento de su creación y luego ya no es posible cambiarlo.

Podríamos crear un buffer de diversas maneras, la más sencilla es la siguiente:

```
var b1 = Buffer.alloc(20);
```

Esto crea un buffer con tamaño de 20 bytes, en el que no hemos definido su contenido, de modo que cada uno de sus bytes estará inicializado a cero.

Pero ten en cuenta que esta manera de crear un buffer es propia de Node 6 en adelante. Antes el equivalente sería así:

```
var b1 = new Buffer(20); // obsoleto desde NodeJS 6.0
```

Si queremos crear un buffer incluyendo ya su contenido podemos usar otros métodos de la clase Buffer, como from().

```
var b2 = Buffer.from('Mañana más');
```

El método `from()` permite crear un nuevo buffer en el que contendremos una cadena de caracteres. Además podemos asignar una codificación, siendo "utf8" la codificación predeterminada.

Escribir un buffer por pantalla

El buffer es una cadena de bytes, en binario, por lo que si lo mostramos por pantalla obtendremos una secuencia de valores que inicialmente pueden no tener mucho sentido:

Por ejemplo con los buffer `b1` y `b2` creados en el punto anterior, podríamos imprimirlos en pantalla con este código.

```
console.log('Este es mi buffer inicializado a cero');
console.log(b1);
console.log('-----');
console.log('Este es mi buffer creado con un string');
console.log(b2);
```

Ejecutando esto, obtendremos una salida como esta.

```
Este es mi buffer inicializado a cero
<Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>
-----
Este es mi buffer creado con un string
<Buffer 4d 61 c3 b1 61 6e 61 20 6d c3 a1 73>
```

Nota: Esa representación en pantalla nos muestra cada una de las posiciones del buffer, que son bytes, con sus valores en hexadecimal. Recuerda que un byte es equivalente a 8 bits. En decimal, un byte te permite acomodar valores numéricos de 0 a 255 y en hexadecimal equivale a valores desde 00 a FF.

Ahora bien, si lo que queremos es ver la cadena tal cual y no los datos en binario, entonces podemos usar el método `toString()` del buffer. El buffer "b1" (que no se había inicializado) no pintaría nada por pantalla, así que para probar es mejor hacer el volcado de la cadena que tenemos en "b2".

```
console.log(b2.toString());
```

Así nos aparecerá la cadena que tenemos en el buffer, tal cual se había introducido "Mañana más".

Manipular el buffer como un array de enteros

Hemos dicho que un buffer es como un array de enteros, por lo que para manipularlo lo podremos usar como tal.

Cada una de las posiciones del buffer se acceden con un índice numérico, comenzando con el índice cero, como los arrays en general en Javascript.

Si haces esto:

```
console.log(b2[0]);
```

Obtienes el valor 77, que en hexadecimal corresponde con el valor 4d y que a su vez es equivalente a la "M".

Podríamos modificar de manera arbitraria varias posiciones de nuestro buffer asignando nuevos valores numéricos y luego imprimirlo de nuevo en pantalla para ver lo que ha pasado.

```
b2[0] = 46;  
b2[2] = 54;  
b2[3] = 55;  
b2[4] = 77;  
console.log(b2.toString())
```

El resultado que veremos es algo como esto:

.a67Mna más

Hemos dicho que el valor de la "M" es 77. Por tanto, si queremos machacar todos los caracteres del buffer, colocando en todas sus casillas la letra "M", podríamos hacer algo como esto:

```
for (var i=0; i<b2.length; i++) {  
    b2[i] = 77;  
}
```

Conclusión

Hemos adquirido un conocimiento muy superficial de los buffer en NodeJS. De hecho, si miras la [documentación de buffer](#) podrás observar que es muy extensa, ofreciendo decenas de métodos para hacer innumerables cosas.

Lo cierto es que, explicados los buffer de manera individual no cobran mucho sentido. En resumen, te permiten tratar información binaria a muy bajo nivel, pero hasta que no nos introduzcamos en otros conceptos de Node, como los streams, no seremos capaces de apreciar su verdadera utilidad.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 10/01/2017
Disponible online en <http://desarrolloweb.com/articulos/buffer-en-nodejs.html>

Streams en NodeJS

Qué son los Streams en NodeJS, cómo podemos usar streams para comunicar un flujo de información de un lugar a otro.

Otro de los conceptos básicos que se manejan muy habitualmente en NodeJS son los streams. En general son flujos de información, o chorros de información que usamos en la transmisión de datos binarios.

El flujo de información que forma un stream se transmite en pedazos, conocidos habitualmente con su término en inglés "chunk". Los chunk no son más que objetos de la clase Buffer, que conocimos en el artículo anterior [Buffer en NodeJS](#).

Es poco habitual crear strings desde cero en un programa, generalmente los usaremos cuando consultemos información que viene de diversas fuentes de datos. En este artículo veremos algunos conceptos generales sobre streams y trabajaremos con streams que nos vienen dados mediante la lectura de archivos de texto que tenemos en el sistema de archivos del ordenador. Sin embargo, los streams los vamos a recibir desde muchas fuentes de información como la manipulación de imágenes, las request del protocolo HTTP o los sockets.



Existen tres tipos de streams, según el uso que queramos realizar mediante estos flujos de datos. Los tenemos de lectura, escritura y duplex (que permiten ambas operaciones a la vez).

Crear un stream a partir de un archivo de texto

Aunque no hemos visto todavía cómo realizar la [lectura de archivos de texto](#) en NodeJS, vamos a apoyarnos en el sistema de archivos del sistema para acceder a un stream de información, con el que realizar algunos ejemplos básicos.

Nota: Aunque FileSystem trabaja con streams, también hay métodos que nos sirven para acceder al contenido de los archivos sin tener que tocar streams directamente y nos devuelven el texto como una cadena.

Antes de nada, si vamos a usar el sistema de archivos del ordenador, tenemos que hacer un require del módulo 'fs' (file system).


```
var fs = require('fs');
```

El método `createReadStream()` del objeto `fs` nos devuelve un stream a cambio de la ruta del archivo que pretendemos leer.

```
var streamLectura = fs.createReadStream('./archivo-texto.txt');
```

Al usar `createReadStream()` recibiremos un stream de lectura. Con él podremos hacer todas las cosas que se encuentran disponibles en el API de NodeJS para los streams. Ahora veremos un par de ejemplos sencillos, pero recuerda que tienes la [documentación de streams](#) para obtener más información.

Eventos en streams

Existen varios eventos que podemos usar con los streams, para realizar acciones cuando ocurran ciertos sucesos. Los eventos disponibles dependen del tipo de stream que tenemos con nosotros. Por ejemplo, para streams de lectura tenemos los eventos "close", "data", "end", "error", "readable".

El evento "data" ocurre cada vez que se reciben datos desde un stream, lo que ocurre al invocar diversos métodos del objeto stream. Además, al crearse un manejador de evento para el evento "data", comienza también la lectura del stream. Cuando el dato se haya leído y se encuentre disponible se ejecutará el propio manejador de evento asociado, recibiendo como parámetro un buffer de datos.

```
streamLectura.on('data', (chunk) => {  
  //chunk es un buffer de datos  
  console.log(chunk instanceof Buffer); //escribe "true" por pantalla  
});
```

Con ese flujo de datos podemos hacer cosas de las que hemos aprendido en el [artículo anterior de los buffer](#). Por ejemplo podríamos ver su contenido de esta manera:

```
streamLectura.on('data', (chunk) => {  
  console.log('He recibido ' + chunk.length + ' bytes de datos.');
```

```
  console.log(chunk.toString());  
});
```

Stream de escritura `process.stdout`

Para hacer alguna cosa con streams de tipo de escritura vamos a basarnos en una propiedad del objeto global `process` de Node, llamada `stdout`. No es más que un stream de escritura con el que podemos generar salida en nuestro programa. La salida de `stdout` es la salida estándar de NodeJS, la propia consola.

Nota: Anteriormente ya conocimos el objeto process y aprendimos alguno de sus métodos en el artículo [proceso de ejecución de NodeJS](#).

Mediante el método write() se escribe en un stream de escritura. Para ello tenemos que enviarle un buffer y otra serie de parámetros opcionales como el tipo de codificación y una función callback a ejecutar cuando termine la operación de escritura.

```
process.stdout.write(objetoBuffer);
```

Como imaginarás, nuestro manejador de evento anterior podría apoyarse en este método write() para conseguir la escritura, en lugar de usar el console.log().

```
streamLectura.on('data', (chunk) => {  
  process.stdout.write(chunk)  
});
```

Usar la entrada estándar process.stdin

Igual que existe un process.stdout para la salida estándar, process.stdin es un stream de lectura para la entrada estándar. Por medio de éste podremos obtener entrada de datos por consola. El proceso no es tan simple como para realizarse en una única acción, como un prompt() de Javascript del lado del cliente, pero se puede entender bien.

Comenzamos con la configuración de la entrada de datos por consola.

```
process.stdin.setEncoding('utf8');
```

Luego podemos mostrar un mensaje en la consola para que se sepa qué dato se está solicitando al usuario.

```
process.stdout.write('¿Qué sugerencias puedes dar a DesarrolloWeb.com? ');
```

Posteriormente podemos asociar un manejador de eventos a stdin, para el evento "data". Este evento en sí lo acabamos de conocer hace un poco. Él produce que la lectura comience y una vez que se tenga algún dato, ejecutará la función callback. En la función callback recibiremos el chunk (buffer) de datos escritos en la consola hasta la pulsación de la tecla enter.

```
process.stdin.once('data', function(res) {  
  process.stdout.write('Has respondido: ');  
  process.stdout.write(res);  
  process.stdin.pause();  
});
```

```
});
```

Dentro de la función callback producimos un poco de salida, mostrando entre otras cosas aquello que se escribió en la consola.

Por último ejecutamos el método `pause()` que producirá que el stream pare de emitir datos, por lo que se dejará de leer en `stdin` y por tanto el programa acabará.

Nota: Observa que ahora estamos asociando un manejador de eventos con el método `once()`, esto quiere decir que el manejador de eventos solo se ejecutará una vez. No obstante, estamos obligados a usar el `process.stdin.pause()` para que el stream de lectura pare de emitir datos. Si no lo hacemos, el proceso quedará suspendido y tendremos que salir de la consola con CTRL + C.

Generar tuberías entre streams

Podemos conectar un stream de lectura a un stream de escritura, produciendo una tubería que enviará los datos del origen para el destino. Para ello usamos el método `pipe()`.

Para crear esa tubería tenemos que invocar el método `pipe` sobre un stream de lectura. Ahora, además del stream de lectura del archivo de texto de antes, vamos a crear un stream de escritura sobre otro archivo del sistema.

```
var streamLectura = fs.createReadStream('./archivo-texto.txt');
var streamEscritura = fs.createWriteStream('./otro-archivo.txt');
```

Ahora vamos a usar el método `pipe()` para realizar ese flujo de datos de un stream a otro.

```
streamLectura.pipe(streamEscritura);
```

Si ejecutamos ese método produciremos la copia del contenido del archivo "archivo-texto.txt" hacia el fichero "otro-archivo.txt".

Podemos saber cuándo terminó esa copia del fichero si añadimos un evento "end" a nuestro stream de lectura.

```
streamLectura.on('end', function() {
  console.log('La lectura del fichero se ha completado');
});
```

Conclusión

Hemos conocido los streams de NodeJS, tanto de lectura como escritura, y hemos

experimentado con algunas funcionalidades de su API. También hemos abordado los streams de lectura y escritura por la entrada/salida estándar en Node, stdin y stdout. Pero además hemos podido darle un poco más de sentido a los buffer de Node, abordados en el artículo anterior, realizando diversas operativas con ellos.

Claro que hay mucho más que aprender pero estamos seguros que este conocimiento general serás capaz de entender muchos de los procedimientos habituales en el trabajo con NodeJS.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 12/01/2017
Disponible online en <http://desarrolloweb.com/articulos/streams-nodejs.html>

Lectura de archivos con NodeJS

Te explicamos cómo realizar la operación de lectura de un archivo en NodeJS, con el módulo fs (File System).

En NodeJS todas las operaciones de acceso al sistema de archivos están englobadas dentro del módulo "fs" (File System). Si queremos leer un archivo de texto que tenemos en local simplemente usaremos ese módulo para extraer el contenido del fichero, indicando su ruta y otra serie de parámetros que ahora describiremos.

En lo que respecta a los métodos del módulo fs hay que señalar que se entregan en dos alternativas, tanto síncrona como asíncrona, que pueden cubrir diferentes situaciones y necesidades que tengamos. Quizás esta sea la parte más destacable del API de NodeJS, del que seguro que ya nos venimos familiarizando a lo largo del [Manual de Node](#), la necesidad de lidiar con funciones asíncronas y sus callback. Lo explicaremos con detalle.



Importar el módulo fs

Igual que hacemos con otros módulos de Node, hay que procesar el correspondiente require para tener disponibles las funciones de acceso al sistema de ficheros. Se encuentran en el module llamado 'fs' y lo vamos a importar con el siguiente código:

```
let fs = require('fs');
```

Nota: El nombre del objeto para operar con el sistema de archivos lo hemos guardado en una variable llamada 'fs'. Podrías usar el nombre de variable que tú desees. Observa además que en vez de "var" estamos usando "let", que es la forma más habitual de declarar variables en ES6. Método `readFile()` asíncrono.

Este método accede a un fichero para su lectura y nos entrega el contenido en forma de buffer o en forma de cadena.

```
fs.readFile(archivo [, options], callback)
```

Recibe tres parámetros, siendo el segundo de ellos opcional. En el primer parámetro le indicamos el nombre del archivo que deseamos leer. Generalmente será una cadena de texto que contenga el nombre del archivo y la ruta que estamos accediendo, pero también podrá ser un buffer.

El segundo parámetro, el opcional, puede ser tanto un objeto como una cadena. Generalmente le indicarás una cadena que contendrá el juego de caracteres en el que el archivo está codificado, por ejemplo 'utf-8'.

El tercer parámetro es la función callback, que se ejecutará en el momento que el archivo está leído y se encuentra disponible para hacer alguna cosa. Esta función recibirá dos parámetros, el archivo ya leído o bien, si ocurre algún error, entonces recibiremos el correspondiente objeto de error de Node. Se puede ver en el siguiente ejemplo:

```
let fs = require('fs');

fs.readFile('archivo.txt', 'utf-8', (err, data) => {
  if(err) {
    console.log('error: ', err);
  } else {
    console.log(data);
  }
});
```

Nota: Estoy usando la notación 'arrow function' de ES6 para resumir las funciones de Javascript en menos código. Esto es perfectamente compatible con las versiones modernas de Node y una recomendación por varios motivos.

Si ese archivo existe, simplemente veremos su contenido como salida del programa. Ojo en este punto, pues si no indicas la codificación, la función te devolverá un buffer en lugar del string.

Si el archivo no existía, nos mostrará el error y podremos ver los diferentes campos del objeto

error de node, desde los que podremos saber qué es lo que ha ocurrido. La salida sería parecida a esta:

```
error: { Error: ENOENT: no such file or directory, open 'archivo-inexistente.txt'
  at Error (native)
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'archivo-inexistente.txt' }
```

Método readFileSync síncrono

Como alternativa tenemos el método `readFileSync()` que hace básicamente lo mismo, pero bloquea la ejecución de las siguientes líneas de código hasta que la lectura haya concluido y se tenga el contenido del archivo completo.

Nota: Digamos que este método equivaldría a la lectura de archivos en la mayoría de los lenguajes de programación. Al ejecutarse bloquea el flujo de ejecución actual, de modo que el código del programa se procesará siempre de manera secuencial.

El método se usa igual, solo que dispensamos la función callback.

```
fs.readFileSync(file[, options])
```

Otra diferencia es que el archivo leído será devuelto por la función. Podemos ver un ejemplo a continuación.

```
let archivo = fs.readFileSync('archivo2.txt', 'utf-8');
```

En esta función, en caso de ocurrir algún tipo de error se lanzará una excepción. En el anterior código no lo hemos realizado, pero se debería tratar si no queremos romper la ejecución del programa.

Lectura síncrona / asíncrona

Primero hay que admitir que la programación se simplifica bastante con la función síncrona, pero lo cierto es que para que el código de ambos ejemplos fuera equivalente, tendríamos que haber tratado en error, en cuyo caso el resultado se comenzará a parecer en complejidad.

La ventaja de la función síncrona es que quizás, para muchas personas, sea más sencillo de codificar, ya que te ahorras las funciones callback. La desventaja es que el procesamiento estará menos optimizado, ya que el hilo de ejecución se quedará unos instantes simplemente esperando, incapaz de adelantar trabajo con el resto del script. en cualquier caso, las

diferencias de optimización de la función asíncrona se notarán más con la lectura de varios archivos grandes, o cuando hay diversos accesos concurrentes al servidor donde se realizan diversos tipos de operaciones.

Para acabar vamos a dejar un código que pone en demostración el flujo de ejecución del programa, usando las dos variantes, síncrona y asíncrona. Es interesante que lo analices y trates de averiguar qué mensajes se mostrarán antes o después.

```
let fs = require('fs');

fs.readFile('archivo-inexistente.txt', 'utf-8', (err, data) => {
  if(err) {
    console.log('error: ', err);
  } else {
    console.log(data);
  }
});

console.log('esto se ejecuta antes que esté el archivo');

let archivo = fs.readFileSync('archivo2.txt', 'utf-8');
console.log(archivo);

console.log('Esto se muestra después de haber leído el archivo2.txt (por el readFileSync)');
```

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 01/12/2016
Disponible online en <http://desarrolloweb.com/articulos/lectura-archivos-nodejs.html>

Variables de entorno en NodeJS, acceso y definición

Qué son las variables de entorno, cómo generarlas al ejecutar una aplicación NodeJS y cómo acceder a ellas desde el código de un script.

Continuamos explicando cosas básicas de la plataforma Node, necesarias para seguir avanzando nuestro [Manual de NodeJS](#). En este artículo veremos qué son las variables de entorno y cómo usarlas para solucionar una de las necesidades básicas de programas que deben ser capaces de funcionar en diferentes ambientes.

Básicamente aprenderemos a definir los valores de las variables de entorno al ejecutar un script Node y cómo recuperarlos dentro del código de los programas, por medio de la biblioteca básica de NodeJS.

Qué son las variables de entorno

Las variables de entorno sirven para definir parámetros sencillos de configuración de los programas, de modo que éstos puedan ejecutarse en diferentes ambientes sin necesidad de modificar el código fuente de un script.

Son útiles en diversos casos del desarrollo en general, pero muy habituales en el desarrollo web porque en la mayoría de las ocasiones los programas se deben ejecutar en diferentes ordenadores. Por ejemplo, durante la etapa de desarrollo puede que tengamos unas configuraciones de entorno y cuando se ponga el programa en producción éstas cambien.

Ejemplos habituales de las variables entorno serían el puerto donde escucha un servidor web, el servidor de base de datos, junto con el usuario y clave para conexión, llaves de APIs, etc. Todos esos valores generalmente pueden cambiar cuando se escribe un programa (al ejecutarlo en local) y cuando se publica el script en un servidor en producción.

Los valores no queremos escribirlos "a fuego" en el programa (de manera literal en el código) porque esto nos obligaría a cambiar el código del programa cada vez que se ejecuta en diferentes lugares, dificultando el proceso de despliegue de los programas. Generalmente desearemos definir esos valores al ejecutar el programa, de modo que cuando lancemos los scripts se pueblen esos valores necesarios para funcionar en cualquier ambiente.

Seguro que si estás familiarizado con el desarrollo web estarás al tanto de la necesidad de las variables de entorno y si no es así este artículo te puede abrir las puertas a soluciones que más tarde o temprano vas a tener que implementar.



Cómo leer variables de entorno en un script NodeJS

Gracias a el objeto "process", disponible en todo programa NodeJS (que nos da información sobre el proceso que está ejecutando este script), tenemos acceso a las variables de entorno de una manera sencilla. Simplemente usamos la propiedad "env" y luego el nombre de la variable que necesitamos acceder.

Por ejemplo, una variable de entorno llamada "PORT" se accedería a través de esta referencia:

```
process.env.PORT
```

Nota: Debido a la naturaleza de los objetos en Javascript, podemos acceder a las propiedades de objetos con notación similar a la que se accede a los arrays. Esto es algo especialmente útil cuando el nombre de la variable de entorno lo tenemos en una cadena. Esta misma variable de entorno la podríamos acceder de este modo:

```
process.env['PORT']
```

Lo que puede ocurrir es que las variables de entorno no siempre se hayan definido, por lo que es útil que en nuestro programa les asignemos unos valores predeterminados. Esto lo podemos conseguir con un código como este:

```
var puerto;  
if(process.env.PORT) {  
  puerto = process.env.PORT;  
} else {  
  puerto = 3000;  
}
```

Sin embargo, una estructura condicional como la anterior se puede expresar de una manera mucho más resumida en Javascript, que debes de conocer:

```
var apiKey = process.env.APIKEY || 'HF33o9o-gl444o1k-992ks';
```

Otra cosa que podrías realizar para definir los valores de variables de entorno por defecto es escribirlos directamente sobre el propio objeto "process.env". Esto podría ser útil para compartir esos valores en diferentes módulos de tu aplicación, ya que process.env es un objeto disponible de manera global.

```
process.env.DB_HOST = process.env.DB_HOST || 'host.db.app.example.com';
```

Esto produce que, si existe esa variable de entorno, no se modifique, pero si no existe se crea con un valor predeterminado. Esta construcción es perfectamente posible porque en Javascript somos capaces de escribir cualquier propiedad en cualquier objeto, aunque sean objetos "de sistema".

Cómo definir las variables de entorno en el momento de ejecución de un programa Node

Al invocar un script Node desde el terminal podemos definir si lo deseamos los valores de las variables de entorno que necesitemos. Esto se hace antes de ejecutar el programa en si, asignando los valores a las variables necesarias.

Como sabes, una aplicación node se ejecuta así: (si el archivo de arranque se llama "simple-env.js")

```
node simple-env
```

Ahora, para especificar las variables de entorno, en lo que era nuestro comando "node" seguido del script a ejecutar, anteponeamos la asignación de aquellas variables que deseemos definir.

```
PORT=3001 APIKEY=880088 node simple-env
```

Conclusión

Eso es todo lo que necesitas conocer de las variables de entorno para usarlas en tu aplicación NodeJS. Como has visto, su uso es realmente sencillo y sus aplicaciones son muy amplias y útiles.

Otra cosa es cómo las quieras gestionar de una manera ágil en tu programa, manteniendo quizás sus valores en un archivo de texto independiente, fácil de editar y de mantener en tu repositorio de código. Eso que es un poco más avanzado lo veremos en un artículo más adelante: [Gestión ágil de variables de entorno NodeJS](#).

Este artículo es obra de *Miguel Angel Alvarez*
 Fue publicado por primera vez en 27/10/2016
 Disponible online en <http://desarrolloweb.com/articulos/variables-entorno-nodejs.html>

Gestión de variables de entorno NodeJS

Cómo realizar un mantenimiento ágil de variables de entorno en aplicaciones NodeJS, para producción y desarrollo, manteniendo sus valores en un archivo independiente.

Es normal que en una aplicación NodeJS mantengamos variables de entorno con valores diferentes cuando estamos en desarrollo y cuando estamos en producción o en cualquier otro ambiente que tengamos. En este artículo te explicaremos una posible manera de estructurar nuestras variables de entorno, en grupos, de modo que las podamos cambiar todas de una única vez dependiendo de si estamos en Producción, Desarrollo, Test, etc.

En un artículo anterior ya explicamos [cómo realizar la gestión básica de variables de entorno en NodeJS](#), tanto su acceso desde un script como su definición en el sistema que va a ejecutarlo. Ahora vamos a profundizar un poco sobre esta necesidad básica de los programas, explicando una posible implementación sencilla de usar y bastante versátil.



La idea o el objetivo que perseguimos en esta práctica es doble:

1. Que no necesitemos especificar toda la lista de variables de entorno posibles al ejecutar un programa, sino simplemente informar si estamos en desarrollo, producción, test, etc.
2. Que podamos mantener todas las variables de entorno en un archivo de texto, de modo que podamos editarlo cómodamente como cualquier otro archivo de código de la aplicación.

Nota: Como hemos dicho, vamos a mantener las cosas sencillas, pero aumentar por nuestra cuenta la versatilidad de un script, para que se pueda ejecutar sin cambiar el código fuente en diversos ambientes. Si ya queremos mejorar las prestaciones y disponer de otras funcionalidades es interesante comentar que existen librerías completas para la gestión de variables de entorno como es el caso de <https://github.com/indexzero/nconf>

En la implementación que vamos a realizar podremos cambiar todo el conjunto de variables de entorno de una única vez, diciendo al ejecutar la aplicación si estamos en desarrollo o producción.

Por ejemplo, para usar el conjunto de variables de entorno predeterminado, que podrían ser los valores necesarios en la etapa de desarrollo, podríamos ejecutar nuestra aplicación sin definir ninguna variable de entorno:

```
node group-env.js
```

Si luego queremos ejecutar esta aplicación con los valores de todas las variables de entorno necesarias para producción, entonces tendríamos que ejecutar la aplicación indicando que estamos en el entorno de producción:

```
NODE_ENV=production node group-env.js
```

Nota: NODE_ENV es un nombre de variable de entorno habitual. Lo usaremos en nuestro programa también, aunque tú podrás usar el nombre de variable de entorno que desees, siempre que en tu código cuando vayas a traerte esa variable lo hagas con el nombre que estés usando.

Definición de las variables de entorno en un JSON

Podemos usar un literal de objeto Javascript (JSON) escrito en un archivo independiente para mantener todas nuestras variables de entorno. La clave aquí es que vamos a definir diversos conjuntos de variables de entorno.

```
{  
  "development": {  
    "SERVERURL": "http://localhost:3001/",
```

```
"PORT": 3001
},
"production": {
  "SERVERURL": "https://app.desarrolloweb.com/",
  "PORT": 5000
}
}
```

Este código lo colocaremos en un archivo a parte, que requeriremos cuando sea necesario. El nombre del archivo es indiferente, en nuestro caso lo hemos guardado dentro de "env.variables.json".

Tenemos dos conjuntos de valores de entorno, "development" y "production", pero tú podrás tener tantos como necesites y por supuesto, también tendrás tantas variables de entorno como requieras y no solamente PORT y SERVERURL como en el JSON anterior.

Require de las variables de entorno

En todo lugar donde necesitemos acceder a las variables de entorno haremos el correspondiente require del JSON anterior.

```
var envJSON = require('./env.variables.json');
```

El el caso anterior, todos los posibles juegos de variables de entorno se volcarán en un objeto al que hemos nombrado envJSON.

Ya solo nos queda definir qué conjunto de variables vamos a usar, para lo que vamos a recibir un única variable de entorno "NODE_ENV". Además, si no se encuentra esa variable vamos a especificar que su valor sea "development".

```
var node_env = process.env.NODE_ENV || 'development';
```

Con esa variable de entorno podemos tomar los valores necesarios del JSON, por ejemplo:

```
var TodasMisVariablesDeEntorno = envJSON[node_env];
```

Y si queremos acceder a una única variable de entorno, podríamos hacer algo como esto:

```
var puerto = envJSON[node_env].PORT;
```

Definir un module para la gestión de variables de entorno

Si queremos ir un poco más allá, podríamos definir el acceso a las variables de entorno en un módulo aparte, de modo que siempre que las necesitamos sea solo realizar el require de ese

módulo de configuración.

El código de nuestro módulo podría ser algo como esto: (lo hemos colocado en un archivo config-module.js, pero podrías usar cualquier nombre que quieras)

```
exports.config = function() {  
  var envJSON = require('./env.variables.json');  
  var node_env = process.env.NODE_ENV || 'development';  
  return envJSON[node_env];  
}
```

Ahora, para acceder a las variables de entorno hacemos el require del módulo anterior (y ejecutamos la función que nos exporta, llamada config) que realiza el acceso al grupo de valores de entorno que necesitamos en cada caso.

```
var entorno = require('./config-module.js').config()
```

Esperamos que esta segunda aproximación al tratamiento de variables de entorno sea de tu agrado y la puedas implementar fácilmente en tus aplicaciones NodeJS.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 22/11/2016
Disponible online en <http://desarrolloweb.com/articulos/gestion-variables-entorno-nodejs.html>

Módulos externos a NodeJS de uso muy habitual

Hay módulos externos a la propia plataforma NodeJS que resultan muy populares y que cualquier desarrollador debería conocer, ya que se usan en gran cantidad de aplicaciones. Revisaremos algunos de ellos, realizando ejemplos de uso.

Primeros pasos con Express

Cómo dar los primeros pasos con Express, el popular framework de NodeJS con el que podemos crear aplicaciones web y APIs REST.

En este artículo vamos a explicar cómo comenzar con Express, el framework más usado en NodeJS, que nos facilitará la creación de servidores web en muy pocos minutos, personalizados según nuestras necesidades.

Instalaremos Express y crearemos nuestro primer servidor, configurado para atender ciertas rutas. Aprenderemos a poner en marcha el servidor, de modo que quede escuchando solicitudes de posibles clientes a los que atender.

Por qué Express

Lo cierto es que NodeJS ofrece ya una librería "http" para realizar servidores web. En pocas líneas de código y sin usar ninguna librería adicional puedes crear tu propio servidor de archivos estáticos, como ya pudimos aprender en el [artículo práctico sobre el módulo HTTP](#). Entonces ¿Para qué necesito Express? Básicamente porque con el mismo esfuerzo tendrás mucho más.

Prácticamente usarás la misma cantidad de líneas de código para iniciar Express que para hacer un servidor desde cero, pero te permitirá no solo servir archivos estáticos, sino atender todo tipo de solicitudes complejas que impliquen realizar cualquier tipo de acción, configurar rutas de manera potente, trabajar de detalladamente con las cabeceras del HTTP y las respuestas, etc. Incluso desarrollar tu propio API REST de una manera más o menos sencilla.

En definitiva, Express es un compañero ideal de NodeJS cuando lo que se quiere hacer es una aplicación web. Dentro de Node es la alternativa más usada, por lo que encontrarás cantidad de ayudas y comunidad para resolver tus dudas o necesidades.



Instalar Express

Express se instala vía "npm". Ya debes conocer la operativa para instalar paquetes de Node, así que no habrá muchas sorpresas. Iniciamos nuestro proyecto con:

```
npm init
```

Y después de contestar la serie de preguntas instalamos Express mediante el comando:

```
npm install --save express
```

Usar Express para crear nuestro primer servidor

Ahora vamos a crear nuestro primer script usando Express, con el código necesario para crear un servidor web. Comenzamos con el "require" del propio Express.

```
var express = require('express');
```

Nota: Este código lo puedes poner en tu index.js (o cualquier otro nombre de archivo que quieras usar, con extensión .js) que puedes crear en la raíz de tu proyecto.

A continuación vamos a inicializar una aplicación Express. Para ello vamos a ejecutar la función que obtuvimos al hacer el require del módulo 'express'.

```
var app = express();
```

Obtenemos como respuesta una variable, que nosotros hemos llamado "app" (convención usada en la mayoría de las veces), mediante la cual podemos configurar la aplicación haciendo uso del API de Express.

Uno de los muchos métodos que tenemos disponibles es listen(), que nos sirve para poner a nuestro servidor web a la escucha.

```
app.listen(3001, function() {  
  console.log('Servidor funcionando en http://localhost:3001');  
});
```

El método `listen` recibe el puerto donde el servidor debe comunicarse con el exterior. El resto de los parámetros son opcionales. En este caso estamos enviando también una función `callback`, que se ejecutará cuando el servidor esté listo, escuchando en el puerto indicado.

En principio con esto ya tenemos nuestro servidor listo. De hecho, si ahora ejecutamos nuestro script, desde el terminal con el comando:

```
node index
```

Comprobaremos que el servidor se pone a la escucha y nos avisa con el mensaje que contiene la ruta donde enviar las solicitudes `http`. Sin embargo, todavía no le hemos dicho qué debe responder ante qué solicitudes.

Crear rutas con Express

Ahora vamos a aprender a configurar el comportamiento de Express atendiendo solicitudes en determinadas rutas del servidor. Definimos una ruta usando el método `get()`

```
app.get('/', function(req, res) {  
  res.send('Hola mundo!! Express!!');  
});
```

Nota: Como puedes imaginar, además de `get()` existen métodos para definir comportamientos cuando se reciben solicitudes mediante otros verbos del HTTP, como `post()`, `put()`, etc.

Como primer parámetro del método `get()` debemos indicar el patrón de la ruta que queremos recibir. En este caso hemos colocado `/`, que equivale a la ruta raíz del servidor. Como segundo parámetro colocamos la función que se ejecutará cuando se reciba una solicitud con tal patrón.

La función encargada de resolver la solicitud recibe dos parámetros que nosotros hemos nombrado `"req"` y `"res"` (también por convención). No son más que la `"request"` de la solicitud HTTP y la `"response"` que enviaremos al cliente.

Apoyándonos en el método `send()` del objeto `"res"` (`response`) podemos enviar cosas como respuesta. En nuestro primer ejemplo hemos enviado una simple cadena de texto como respuesta, pero podría ser un HTML, un fichero, un JSON, etc. Si ahora ejecutas tu servidor de nuevo, accediendo a la raíz, podrás ver el mensaje `"Hola mundo!! Express!!"`.

Nota: Ten en cuenta que, para que funcione esta nueva ruta, debes detener y reiniciar el servidor. Desde el terminal de comandos debes salir de la ejecución del servidor con CTRL+C y luego reiniciarlo invocando de nuevo al programa con el comando "node index" (o cualquier nombre de archivo que hayas usado para este script).

Existen cientos de formas de aprovechar el sistema de rutas para conseguir cosas diferentes. Por ejemplo, usando parámetros en las rutas tal como sigue:

```
app.get('/bienvenido/:nombre', function(req, res) {  
  res.send('Bienvenido!! ' + req.params.nombre + '!!');  
});
```

En la ruta definida ahora ":nombre" indica que es un parámetro que puede tomar cualquier valor. Estamos configurando la ruta "bienvenido/" seguido de cualquier cadena. Para recibir el valor de la cadena en la ruta tendremos disponible en el objeto request.params. El resultado es que una ruta como como <http://localhost:3001/bienvenido/miguel>, nos contestará con el texto "Bienvenido!! miguel!!"

Simple, ¿no?. Quizás lo veas así, pero es muy potente, pues solo hemos visto lo más básico. Existen cientos de configuraciones en Express para satisfacer cualquier necesidad que se te ocurra.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 15/09/2016
Disponible online en <http://desarrolloweb.com/articulos/primeros-pasos-express.html>

Ejecutar una aplicación NodeJS en producción con PM2

Cómo ejecutar una aplicación NodeJS como proceso de manera perpetua en producción, usando el gestor de procesos PM2.

Este artículo está dedicado a resolver una necesidad que la mayoría de las personas tendrán cuando deban poner en producción una aplicación con realizada con NodeJS. Por el estilo de programas del lado del servidor que desarrollamos en NodeJS seguramente tu aplicación costará de un proceso, o varios, que se tienen que ejecutar de manera continua. Casos habituales podrían ser un servidor de ficheros estáticos o servicio web ([API REST](#) o similares).

Durante la etapa de desarrollo venías ejecutando "node index", o algo así, para arrancar tu aplicación, pero esa mecánica tiene varios problemas. Uno de ellos es que ante un error en tu aplicación simplemente se parará la ejecución quedando el servidor caído. También, en casos de reinicio del servidor se necesitaría volver a arrancar los procesos manualmente.

La solución pasa por instalar un gestor de procesos, o correr tu aplicación como servicio. En

este artículo vamos a explicar cómo usar un popular gestor de procesos (Process Manager) de NodeJS que está realmente completo. Se trata de PM2, una librería gratuita que vengo usando en un par de proyectos con mucho éxito, capaz de aguantar cantidades enormes de tráfico con un consumo de recursos realmente reducido y con herramientas que permiten realizar la monitorización de las aplicaciones de manera remota.



Puedes comenzar echando un vistazo a la página de PM2: <http://pm2.keymetrics.io/>

Instalar PM2

Si quieres disponer de tu gestor de procesos en el servidor, tendrás que instalarlo como cualquier otra aplicación Node.

```
npm install -g pm2
```

Una vez instalada disfrutarás de todas las funcionalidades de este process manager, como la posibilidad de parar y arrancar procesos, monitorizarlos en tiempo real, gestionar los log de aplicación, etc.

Arrancar y parar procesos

Lo primero que debes aprender es a arrancar y detener procesos, o volverlos a arrancar cuando sea necesario. Obviamente, en vez de solicitar a Node que ejecute tal o cual fichero, se lo pediremos directamente a PM2.

```
pm2 start index.js
```

Con esto podrás arrancar un proceso, asegurando que tu servidor permanezca encendido. Es habitual que quieras asignar un nombre al proceso, de manera que luego te puedas referir a él en otro tipo de comandos.

```
pm2 start app.js --name "mi-api"
```

Nota: Si no asignaste explícitamente al proceso un nombre, al hacer el "pm2 start" con la

opción "--name", se le asignará un nombre igualmente, generado a partir del nombre del archivo que se pone en ejecución, sin la extensión ".js".

Para detener el proceso se usará el comando stop, indicando el nombre del proceso que quieres parar.

```
pm2 stop mi-api
```

O para reiniciarlo, el comando restart.

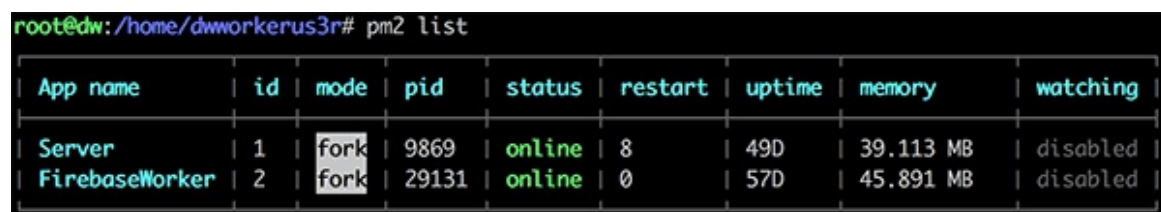
```
pm2 restart mi-api
```

Mantenimiento de los procesos

Tal como lo tienes ahora, gracias a que PM2 controla estos procesos listados, se arrancarán nuevamente en caso de error, manteniéndose encendidos mientras la máquina permanezca encendida. Es decir, en el hipotético caso que uno de ellos se termine por cualquier motivo, como un error en el programa que haga que el proceso se acabe, PM2 lo iniciará de nuevo automáticamente.

Para encontrar información sobre los procesos controlados por PM2 puedes listarlos con el comando list:

```
pm2 list
```



App name	id	mode	pid	status	restart	uptime	memory	watching
Server	1	fork	9869	online	8	49D	39.113 MB	disabled
FirebaseWorker	2	fork	29131	online	0	57D	45.891 MB	disabled

En la columna "restart" podremos encontrar un contador con el número de veces que se ha reiniciado el proceso. Si ese número crece continuamente querrá decir que PM2 está teniendo que re-arrancar el proceso y será un indicador de que algo no está funcionando bien.

El listado de procesos te da una información interesante, pero para saber qué es lo que está ocurriendo internamente te será generalmente más útil echar un vistazo a los log.

Nota: Nos log no solo te avisan de lo que está ocurriendo, como errores o reinicios de los procesos, también es donde aparecen todos los "console.log()" realizados desde el código de los programas, por lo que también son útiles para debuggear un código.

La administración de los log en PM2 es bastante configurable. Sin entrar en el tema de la configuración del sistema de logs, rotación de ficheros y esas cosas, es interesante comentar que hay un comando que te permite ver en tiempo real todos los log que se están produciendo en tus procesos.

```
pm2 log
```

Este comando te mostrará los últimos log y se quedará arrancado, mostrando nuevos mensajes que los procesos envíen como salida por consola. En caso de reinicio de un proceso arrancado con PM2 podrás observar en tiempo real cómo el gestor de procesos se encarga de reiniciarlo. En la siguiente imagen tienes un pedazo de salida del seguimiento de logs en tiempo real de PM2 cuando se produce el reinicio de un proceso:

```
[STREAMING] Now streaming realtime logs for [all] processes
PM2       | App [index] with id [0] and pid [3175], exited with code [0] via signal [SIGINT]
PM2       | Starting execution sequence in -fork mode- for app name:index id:0
PM2       | App name:index id:0 online
0lindex   | Servidor funcionando en http://localhost:3001
```

Como has podido comprobar, tus procesos reinician automáticamente, pero ¿Qué pasaría si la máquina se apaga? Todavía tenemos que configurar un último paso para permitir que los procesos arranquen también con el reinicio de la máquina.

Generación del script de startup

Para acabar nuestra configuración básica de PM2 necesitamos configurar el script de startup del servidor.

Con tus procesos en marcha, arrancados mediante PM2, ahora puedes generar de manera automatizada el correspondiente script, sin tener que preocuparte por la programación, ya que PM2 lo generará para ti. Para ello tenemos el comando siguiente:

```
pm2 startup
```

Ese comando detectará la plataforma donde estás ejecutando, si es Linux, Mac, Windows y generará el script de arranque. Sin embargo, la detección automática de la plataforma no siempre funciona, o al menos no siempre reconoce la distribución de Linux de tu servidor, por lo que es a veces necesario el informar nuestra plataforma de manera explícita:

```
pm2 startup centos
```

Existen varios [valores de plataformas que puedes consultar en la documentación](#).

En algunos casos me ha tocado hacer después un comando adicional para guardar la lista de procesos, me figuro es cuando necesitas cambiar la lista de procesos después de haber generado el script de startup:

```
pm2 save
```

Configuración de los procesos de tu aplicación

El siguiente paso para sacarle el partido a PM2 es configurar un archivo con la información de los procesos que tiene que administrar el sistema. En este archivo podemos especificar mucha información útil para el control de los procesos, que se tomará como valores cuando inicien, o ante reinicios.

El formato del archivo lo puedes realizar en JSON, YAML o en código Javascript y puedes encontrar bastante información en la documentación del ["process file de PM2"](#).

Lo bueno de este archivo es que permite establecer muchos tipos de configuraciones que no serían posibles de realizar por medio de línea de comandos. Además permite especificar un conjunto de procesos y arrancarlos todos con un único comando.

El archivo puede tener cualquier nombre, por ejemplo "process.json". Por ejemplo este podría ser un posible contenido para la configuración en formato JSON:

```
{
  apps : [{
    name      : "Server",
    script    : "./server.js",
    instances: 2,
    env: {
      "PORT": "3001"
    }
  },
  {
    name: "Worker",
    script: "./worker.js"
  }
]
```

Estamos especificando dos procesos y una pequeña muestra de posibles configuraciones. Si quieres arrancar estos procesos de una única vez lanzas el comando:

```
pm2 start process.json
```

De manera similar puedes reiniciar los procesos con "restart" o pararlos con "stop. Como estamos apoyándonos en el archivo de configuración se inician y se paran todos los que se haya indicado, con las opciones configuradas.

Ahora veremos algo básico, pero que necesitarás en la mayoría de tus proyectos, como es la definición de valores para las [variables de entorno](#).

Si por ejemplo quieres definir valores a variables de entorno podrías usar un archivo

process.json como este:

```
{
  apps : [{
    name      : "main-app",
    script    : "./server.js",
    env: {
      "NODE_ENV": "development",
    },
    env_production : {
      "NODE_ENV": "production"
    }
  }]
}
```

En este archivo estás indicando que la variable de entorno "NODE_ENV" tendrá el valor "development". Pero también estamos indicando que esa misma variable en entorno de producción tenga el valor "production".

Si inicias los procesos sin indicar nada:

```
pm2 start process.json
```

La variable de entorno "NODE_ENV" tendrá como valor "development". En el caso que desees especificar un entorno diferente lo indicarás de esta manera:

```
pm2 restart process.json --env production
```

En este caso la variable "NODE_ENV" tendrá el valor "production".

Conclusión

Con lo que hemos tratado tendrás suficiente para gestionar los procesos en aplicaciones realizadas con Node, pero detrás de PM2 hay mucho más.

Más adelante podríamos hablar de otras muchas funcionalidades que no hemos tocado y que seguro te vendrá bien conocer, como la gestión de logs, las herramientas de monitorización, el proceso de observación de archivos para re arranque automático cuando el código de la aplicación cambia, las utilidades de deploy, etc.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado por primera vez en 31/01/2017
Disponible online en <http://desarrolloweb.com/articulos/ejecutar-aplicacion-nodejs-pm2.html>

