

# INTRODUCCIÓN A NODE.JS

*Conoce las capacidades de una de las  
plataformas más populares.*



.js

GUSTAVO MORALES

# Introducción a Node.js

Gustavo Morales

Este libro está a la venta en <http://leanpub.com/introduccion-a-nodejs>

Esta versión se publicó en 2020-08-02



Éste es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

© 2019 - 2020 Gustavo Morales

# Índice general

<b>Introducción</b>	<b>1</b>
A quién está dirigido este libro	1
Acerca de este libro	1
Acerca del autor	1
Acerca de la editora	2
Acerca del editor técnico	2
Agradecimientos	2
<b>¿Qué es Node.js?</b>	<b>3</b>
Librería de V8 JavaScript	3
Librería libuv	3
¿Node.js es <i>single thread</i> ?	4
<b>Instalación</b>	<b>5</b>
<b>Administrador de versiones</b>	<b>6</b>
Instalación de administrador de versiones	6
Instalación en Mac OS	6
Instalación en Windows	7
Administración de versiones	8
Versiones de Node.js y Long Term Support (LTS)	9
Seleccionar automáticamente la versión correspondiente para cada proyecto	11
<b>Primer programa en Node.js</b>	<b>12</b>
<b>¿Cómo funciona Node.js?</b>	<b>13</b>
REPL	13
Objeto Global	14
Objeto Process	15
Argumentos de línea de comando	16
<b>Instalación y administración de paquetes con npm</b>	<b>18</b>
Instalar paquetes locales	18
Administración de versiones de paquetes	19
Utilizar paquetes locales	20

## ÍNDICE GENERAL

Instalar y utilizar paquetes globales . . . . .	21
Resolución de paquetes . . . . .	22
Otros administradores de paquetes . . . . .	22
<b>Inicialización de un proyecto de Node . . . . .</b>	<b>24</b>
Administración de paquetes con el archivo package.json . . . . .	25
Configuración de git y Github . . . . .	26
<b>Configuración de npm scripts . . . . .</b>	<b>28</b>
Scripts incorporados . . . . .	28
Scripts personalizados . . . . .	29
Convenciones de un proyecto que incluyen <i>npm</i> . . . . .	29
Utilizando las dependencias y <i>scripts</i> . . . . .	29
Pre y pos <i>scripts</i> . . . . .	30
<b>Estandarizar argumentos de línea de comandos . . . . .</b>	<b>31</b>
<b>Convenciones de <i>callback</i> en Node.js . . . . .</b>	<b>33</b>
Funciones como objetos de primera clase en JavaScript . . . . .	33
<i>Callbacks</i> y <i>the continuation-passing style (CPS)</i> . . . . .	35
<i>Asynchronous continuation-passing style</i> . . . . .	36
Los <i>callbacks</i> van al final . . . . .	37
Los errores van de primero . . . . .	37
<b>Trabajar con archivos . . . . .</b>	<b>38</b>
Escribir archivos JSON . . . . .	39
Leer archivos JSON . . . . .	42
Imprimir contenido de archivos JSON . . . . .	42
<b>Módulos . . . . .</b>	<b>44</b>
Modulizar la aplicación . . . . .	48
<b>Interactuar con la consola . . . . .</b>	<b>51</b>
<b>Depurar aplicaciones . . . . .</b>	<b>54</b>
Depurar con Node.js . . . . .	54
Depurar con <i>Chrome Dev Tools</i> . . . . .	57
Depurar con VS Code . . . . .	62
<b><i>Blocking code</i> y <i>Non-Blocking code</i> . . . . .</b>	<b>69</b>
El <i>Event loop</i> . . . . .	70
El patrón <i>observer</i> y <i>Event Emitter</i> . . . . .	73
<b>Servidor Web . . . . .</b>	<b>77</b>
Renderizar páginas Web . . . . .	77
Renderizar vistas dinámicas . . . . .	82

## ÍNDICE GENERAL

Utilizar Express JS . . . . .	86
<b>Conclusión . . . . .</b>	<b>91</b>

# Introducción

## A quién está dirigido este libro

Existen muchos lenguajes de programación, algunos de ellos con un propósito específico y otros con propósitos más generales e inclusive se habla de lenguajes de programación para la Web; por lo tanto, este libro está dirigido a personas con conocimientos generales de programación y conocimiento básico de la sintaxis de JavaScript u otro lenguaje ANSI C como: Java, PHP, etc., que quieran introducirse en el mundo del desarrollo Web, específicamente del servidor con Node.js, con un lenguaje de propósito “general” como JavaScript. Pero también personas que han trabajado JavaScript del lado del cliente, ya sea simple *vanilla* JavaScript, librerías y/o frameworks, y quieran aprovechar ese conocimiento que tienen del lenguaje JavaScript para utilizar en el servidor.

## Acerca de este libro

Este libro tiene un nivel básico introductorio. Busca aclarar los conceptos principales de Node.js desde su instalación y configuración hasta el uso de características más utilizadas. El texto guía a través de ejemplos prácticos, pero también propone ejercicios para afianzar dichos conceptos. Muchos de los ejemplos son patrones y fragmentos de código utilizados en la industria, pero no pretende ser la guía definitiva que aborde cada una de las características de Node.js a profundidad.

El código fuente de los ejemplos y ejercicios utilizados en este libro se pueden encontrar en el siguiente [repositorio](#)<sup>1</sup>.

## Acerca del autor

Soy un entusiasta de la Web desde finales de los 90, cuando se utilizaba el bloc de notas y nombre de etiquetas en mayúsculas para hacer las páginas Web. Soy Ingeniero de Sistemas y tengo más de doce años de experiencia como profesor de pregrado en la asignatura de programación bajo la Web y en posgrado con otras asignaturas relacionadas con la industria del desarrollo de software. Trabajo actualmente para una agencia digital en la que específicamente elaboro proyectos de programación Web. Puedes encontrar más información acerca de mi hoja de vida en el siguiente enlace: <http://gmoralesc.me><sup>2</sup>.

**Gustavo Morales**

---

<sup>1</sup><https://github.com/gmoralesc/introduccion-a-node-js>

<sup>2</sup><http://gmoralesc.me>

## Acerca de la editora

Creo fervientemente que la tecnología debería estar al alcance de todos y no ser el privilegio de pocos; por ello, me he dedicado a la edición de libros educativos digitales.

Tengo 34 años de experiencia como docente universitaria. También he sido la editora de una revista académica especializada en investigación educativa. He coordinado la Unidad de Innovación e Investigación del Centro para la Excelencia Docente de la universidad donde laboro.

Pueden consultar mi página Web si están interesados en tener mayor información: [Blog<sup>3</sup>](#)

Adela De Castro

## Acerca del editor técnico

Diseñador Gráfico profesional y Desarrollador Front End por gusto. Con más de 7 años de experiencia en Desarrollo Web, Diseño Gráfico y Motion Graphics. Líder Técnico del área de Front End en Zemoga desde 2015, ha trabajado con clientes en Colombia y en Estados Unidos como Alquilería, Federación de Ciclismo de Colombia, Sears, Fox Entertainment y Morningstar entre otros. Cuenta con experiencia en tecnologías de Front End (HTML/CSS/JavaScript) así como tecnologías de Back End como PHP y recientemente Node.js. Ha participado en proyectos con distintos requerimientos técnicos en donde ha propuesto soluciones de arquitectura escalable con microservicios, *serverless* y otros. Su blog es [Blog<sup>4</sup>](#)

Andres Zorro

## Agradecimientos

Quiero agradecer a mi esposa Milagro y a mi hijo Miguel Ángel por apoyarme y entender la gran cantidad de horas que tuve que invertir frente al computador investigando, estudiando, probando y escribiendo.

---

<sup>3</sup><https://www.uninorte.edu.co/web/decastro>

<sup>4</sup><https://medium.com/@zorrodg>

# ¿Qué es Node.js?

Node.js es un programa de escritorio que, principalmente, está escrito en C++ (y JavaScript) pero permite escribir aplicaciones utilizando como lenguaje JavaScript. Por esta razón muchos le llaman el JavaScript de escritorio, pero Node.js es un poco más que eso. Con el soporte de múltiples librerías proporciona una gran variedad de apoyos para la creación de diferentes tipos de aplicaciones, desde aplicaciones de línea de comando, REST API hasta servidores Web, pero uno de sus usos más efectivos es todo lo relacionado con RTC (*Real Time Communication*).

De hecho Node.js soporta la incorporación de librerías escritas en C++ y estas se pueden vincular a los proyectos ampliando aún más el sustento de nuevas funcionalidades.

Node.js es *Open Source*, ya que muchas personas contribuyen y mantienen su [código fuente](#)<sup>5</sup>.

## Librería de V8 JavaScript

Había mencionado anteriormente que Node.js permite escribir aplicaciones utilizando JavaScript como lenguaje, pero está escrito en C++ ¿Como es esto posible?, Es debido a una de las librerías más importantes para Node.js, V8 también escrita en C++. Esta permite convertir el código escrito por el usuario en JavaScript a código de máquina para ser interpretado por el computador más eficientemente.

Esta librería es ahora [código abierto](#)<sup>6</sup>, liberado por Google. Es utilizada por otras aplicaciones como el navegador Web *Google Chrome*, el cual la usa para interpretar el código escrito en JavaScript en el navegador Web.

- [Código fuente V8](#)<sup>7</sup>

## Librería libuv

La librería *libuv* permite a Node.js implementar muchas operaciones asincrónicas como: lectura de archivos, peticiones HTTP y muchas otras operaciones mayormente de entrada/salida, permitiendo así procesar varias cosas al “mismo tiempo”.

- [Lib UV](#)<sup>8</sup>

---

<sup>5</sup><https://github.com/nodejs/node>

<sup>6</sup><https://v8.dev/>

<sup>7</sup><https://github.com/v8/v8>

<sup>8</sup><https://github.com/libuv/libuv>



## ¿Node.js es *single thread*?

Cuando se habla de “programas” ejecutándose, realmente la mayoría lo hace como procesos en el sistema. Cada uno de estos procesos puede utilizar uno o varios hilos (*threads*) para hacer las diferentes operaciones con el procesador u operaciones de entrada/salida como lectura de archivos; esto permite realizar varias tareas de manera “simultánea”, aunque este último concepto también depende de la arquitectura que tenga el procesador para soportar este tipo operaciones.

Se menciona mucho que Node.js es *single thread* debido a la implementación de JavaScript de la librería V8 que es *single thread*. Esto significaría que solo puede procesar una cosa al tiempo, pero esto no es del todo cierto, así que realizaremos la primera aclaración:

- El código escrito por el usuario se ejecuta en *single thread* en el *event loop* (el cual veremos con más detalle más adelante)

Pero como se dijo antes, Node.js utiliza otras librerías para añadir funcionalidades como *libuv*; esta le permite leer archivos del sistema. Entonces, cuando esta instrucción escrita por el usuario es ejecutada en el *event loop*, se crea un nuevo hilo en el proceso para realizar esta operación y notificará al hilo principal cuando este termine. Es entonces cuando realizaremos nuestra segunda aclaración:

- Cuando una operación requiere hacer una operación de entrada/salida o un procesamiento independiente, un nuevo hilo se crea en el proceso y, una vez finalizada la operación, notifica al hilo principal.

Por defecto *libuv* determina cual es la cantidad de hilos que tiene para procesar las operaciones y, si estos están “ocupados”, deja la tarea en espera hasta que algún otro hilo termine su ejecución. Este parámetro se puede sobrescribir en el código del usuario si es necesario, pero también depende mucho de la arquitectura del procesador para realizar un buen balance entre la capacidad del procesador y el número de hilos que se establezcan.

La anterior es la razón por la cual se menciona que *libuv* le agrega a Node.js la capacidad de procesar tareas de manera simultánea, y muchas de estas funciones tiene su versión sincrónica y asincrónica, lo que también explicaremos en detalle más adelante.

En conclusión Node.js no es solo *single thread*, es *multi thread* cuando es necesario.

# Instalación

Normalmente, para instalar cualquier tecnología, el primer paso es visitar su página oficial, ir a la sección de descargas, seleccionar el sistema operativo y seguir las instrucciones de instalación. Esto funcionará siempre y cuando sólo vayamos a utilizar una sola versión de la tecnología o lenguaje de programación en el sistema para todos los proyectos. Pero lo más probable es que ese no sea el caso, ya que a medida que pase el tiempo van saliendo nuevas versiones y de seguro tendremos que apoyar algún proyecto que funciona en una versión específica o anterior a la última de Node.js. Por lo tanto en esta primera sección lo instalaremos primero de manera tradicional.

Para instalar Node.js seguir los siguientes pasos:

1. Visitar la página Web oficial de [Node.js](https://nodejs.org/es/)<sup>9</sup>.
2. La página detectará automáticamente el sistema operativo.
3. Hacer clic en el enlace que tiene como leyenda: “Recomendada para la mayoría”.
4. Seguir las instrucciones del instalador

Para comprobar la versión instalada:

1. Abrir la Terminal
2. Ejecutar el comando:

```
node -v
```

Es muy tedioso tener que instalar y desinstalar las diferentes versiones, por lo cual la estrategia más recomendable es instalar un administrador de versiones, para este caso específico de Node.js. El cual veremos en la próxima sección, cabe notar que este paso es opcional.

---

<sup>9</sup><https://nodejs.org/es/>

# Administrador de versiones

Un administrador de versiones, como su nombre lo indica, permite administrar diferentes versiones de la tecnología o lenguaje de programación en el sistema. Entre las operaciones que podemos realizar están: instalar nuevas versiones, desinstalar versiones existentes, seleccionar qué versión se va a utilizar y seleccionar la versión por defecto seleccionada en el sistema.

Si ya tiene una versión instalada de Node.js en el sistema, no es necesario desinstalarla para utilizar el administrador de versiones, puesto que inclusive el administrador de versiones la incluye dentro de las opciones que se pueden seleccionar.

## Instalación de administrador de versiones

Existen diferentes administradores de versiones para Node.js. Uno de los más populares es *nvm*, el cual es una utilidad que nos permite administrar las diferentes versiones de Node.js en el sistema, brindando así la facilidad de seleccionar la versión necesaria para cada proyecto.

## Instalación en Mac OS

### Instalación de Homebrew

Mac OS cuenta con un programa muy popular para administrar diferentes paquetes llamada *brew*. Para instalarla, abrimos la terminal de Mac OS y ejecutamos el siguiente comando:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```



Durante el proceso de instalación, si es necesario, el instalador preguntará por su contraseña para establecer los permisos necesarios para completar satisfactoriamente la instalación.

Terminada la instalación podemos comprobar en la terminal la versión instalada con el siguiente comando:

```
brew -v
```

En la página oficial de [Homebrew](https://brew.sh)<sup>10</sup> está la información detallada sobre el proceso y los posibles problemas asociados a la misma.

---

<sup>10</sup><https://brew.sh>

## Instalación de nvm

Una vez instalado *brew* procedemos a instalar *nvm*:

```
brew install nvm
```

Terminada la instalación, podemos comprobar en la terminal la versión instalada con el siguiente comando:

```
nvm -v
```



Lo más probable es que *nvm* requiera realizar una configuración adicional, así que debes seguir las recomendaciones indicadas una vez finalice el proceso de instalación

Adicionalmente se recomienda reiniciar la sesión de la terminal.

En el repositorio oficial de [nvm](https://github.com/nvm-sh/nvm)<sup>11</sup> está toda la información detallada sobre el proceso de instalación y los posibles problemas asociados con la misma.

## Instalación en Windows

### Instalación de cmdr

Versiones anteriores a Windows 10 ofrecen una terminal muy básica; pero el proyecto *cmdr* ofrece una terminal con muchas opciones dentro de ella, la mayoría de comandos de los sistemas Linux, *git* y muchas otras herramientas.

Para instalarlo, seguimos los siguientes pasos:

- Visitar la página oficial de [cmdr](https://github.com/cmderdev/cmder)<sup>12</sup>
- Hacer clic en el enlace “Download Full”
- Descomprimir el archivo descargado en una ubicación conocida como: Archivos de programa
- Hacer un acceso directo en el Escritorio del archivo *cmdr.exe*
- Se recomienda ejecutar el programa con permisos de Administrador

En el repositorio oficial de [cmdr](https://github.com/cmderdev/cmder)<sup>13</sup> está toda la información detallada sobre el proceso de instalación y los posibles problemas asociados a la misma.

---

<sup>11</sup><https://github.com/creationix/nvm#installation>

<sup>12</sup><https://legacy.gitbook.com/book/gmoralesc/introduccion-a-node-js/edit#>

<sup>13</sup><https://github.com/cmderdev/cmder>

## Instalación de *nvm* para Windows

La utilidad de *nvm* no se encuentra disponible para Windows, pero existe un proyecto muy similar que cumple el mismo objetivo.

Para instalarlo seguimos los siguientes pasos:

- Visitar la página oficial del proyecto [nvm for Windows](https://github.com/coreybutler/nvm-windows)<sup>14</sup>
- Hacer clic en el enlace “Download Now”
- Luego en el listado hacer clic en el enlace “nvm-setup.zip” de la versión más reciente
- Descomprimir el archivo descargado
- Ejecutar el instalador “nvm-setup.exe”

En el repositorio oficial de [nvm for Windows](https://github.com/coreybutler/nvm-windows)<sup>15</sup> está toda la información detallada sobre el proceso de instalación y los posibles problemas asociados a la misma.

## Administración de versiones

Una vez instalado *nvm* procedemos a instalar la versión estable de Node.js; para lo cual ,ejecutamos en la terminal el siguiente comando:

```
nvm install stable
```

Esta es la salida en la pantalla del comando anterior:

```
Downloading and installing node v12.13.1...
Downloading https://nodejs.org/dist/v12.13.1/node-v12.13.1-darwin-x64.tar.xz...
##### 100.0%
Computing checksum with shasum -a 256
Checksums matched!
Now using node v12.13.1 (npm v6.12.1)
```

Terminado el proceso de instalación verificamos en la terminal la versión instalada y activa en el sistema, con el siguiente comando:

```
node -v
```

Esta es la salida en la pantalla del comando anterior:

---

<sup>14</sup><https://github.com/coreybutler/nvm-windows#node-version-manager-nvm-for-windows>

<sup>15</sup><https://github.com/coreybutler/nvm-windows#node-version-manager-nvm-for-windows>

v12.13.1

## Versiones de Node.js y Long Term Support (LTS)

Las diferentes versiones de Node JS se pueden verificar en la siguiente página [Node JS Release<sup>16</sup>](#); allí podremos ver el ciclo de vida de cada una de ellas y cuál es el estado del *Long Term Support* (LTS), que indica desde y hasta cuando se realizan las actualizaciones de mantenimiento y seguridad de cada una de las versiones.

A modo de ejemplo vamos a instalar una versión anterior:

```
nvm install 10
```

Terminado el proceso de montaje verificamos en la terminal la versión instalada con el siguiente comando:

```
node -v
```

Como podemos observar, la versión reciente instalada ha quedado como la versión activa en esta sesión de la terminal. Si se abre una nueva sesión de la terminal quedará seleccionada la versión que se tiene indicada por defecto en el sistema. Es decir, que se pueden tener diferentes versiones de Node.js seleccionadas cada una de ellas en una sesión diferente de la terminal. Lo anterior es una enorme ventaja a la hora de ejecutar varios proyectos al mismo tiempo, que necesiten diferentes versiones.

Para listar todas las versiones instaladas localmente por nvm, ejecutamos el siguiente comando:

```
nvm ls
```

Esta es la salida en la pantalla del comando anterior:

```
v8.16.1
v10.17.0
-> v12.13.1
v13.2.0
default -> 10 (-> v10.17.0)
node -> stable (-> v12.13.1) (default)
stable -> 12.13 (-> v12.13.1) (default)
iojs -> N/A (default)
lts/* -> lts/erbium (-> v12.13.1)
lts/argon -> v4.9.1 (-> N/A)
```

---

<sup>16</sup><https://github.com/nodejs/Release>

```
lts/boron -> v6.17.1 (-> N/A)
lts/carbon -> v8.16.2 (-> N/A)
lts/dubnium -> v10.17.0
lts/erbium -> v12.13.1
```

A continuación, daré diferentes anotaciones sobre este listado anterior:

- El símbolo de `->` indica qué versión está seleccionada actualmente en la sesión de la terminal
- La versión seleccionada por defecto con una nueva sesión de la terminal es la que está marcada con `default`
- La versión estable es `stable`
- Sobre el soporte LTS, están marcadas con el prefijo `lts`.

*iojs* fue en algún momento una copia del proyecto de Node.js, pero afortunadamente se unieron nuevamente y publicaron la versión 4 de Node.js; por ello la razón del salto de la versión 0.12 a la versión 4

Una vez instaladas diferentes versiones mediante `nvm`, se puede seleccionar la versión con la cual se desea trabajar, en este caso seleccionamos la versión estable con el siguiente comando:

```
nvm use stable
```

El parámetro después del comando `use` es la versión que se desea seleccionar

Es posible seleccionar la versión instalada en el sistema con `nvm use system`

Vamos a seleccionar una versión como predeterminada para cada nueva sesión de la terminal, por ejemplo, la versión estable. Para ello ejecutamos el siguiente comando:

```
nvm alias default stable
```

Si usted lo desea, existe la posibilidad de crear más alias con `nvm` para las diferentes instalaciones

Para listar todas las versiones que `nvm` puede instalar en el sistema:

```
nvm ls-remote
```

## Seleccionar automáticamente la versión correspondiente para cada proyecto

Sería muy dispendioso, cada vez que vayamos a trabajar en un proyecto, seleccionar la versión necesaria (si no es la predeterminada), e inclusive, es posible confundir u olvidar cuál era la versión necesitada.

Para esto *nvm* tiene una solución, y es crear en la raíz del directorio del proyecto un archivo con el nombre `.nvmrc` y en su contenido colocar solamente el número de la versión de Node.js que corresponda a dicho proyecto. Así, cuando se ingrese al directorio del proyecto mediante la terminal *nvm*, automáticamente seleccionará la versión especificada en el archivo `.nvmrc`, por ejemplo:

10

Como podemos ver el contenido del archivo solo tiene el número de la versión de Node.js a utilizar en ese proyecto, en este caso es la versión anterior a la instalada por defecto.



# Primer programa en Node.js

Antes de empezar a trabajar seleccionemos un directorio de trabajo, en los sistemas Linux es muy sencillo llegar al directorio del usuario, actualmente autenticado en el sistema con el siguiente comando:

```
cd ~
```



Para conocer la ruta del directorio donde está localizado podemos utilizar el comando `pwd`.

A continuación crearemos el directorio de trabajo llamado “greeting” e ingresamos a él con el siguiente comando:

```
mkdir greeting && cd greeting
```

Vamos a crear una aplicación que cada vez que la ejecutamos salude al usuario y nos indique el año actual; por lo tanto, ubicados en el directorio de trabajo creamos un archivo llamado `index.js` con el siguiente contenido:

```
1  const name = "Gustavo";
2  const year = new Date().getFullYear();
3
4  console.log(`Hey ${name}, we are in ${year}`);
```

En el fragmento de código anterior estamos utilizando interpolación de variables con los *Template Strings* de JavaScript.

Ejecutamos nuestro programa con el siguiente comando:

```
node index.js
```

Felicitaciones has creado tu primer programa en Node.js.

[Código fuente de la aplicación<sup>17</sup>](#)

---

<sup>17</sup><https://github.com/gmoralesc/introduccion-a-node-js/tree/master/01-greeting>

# ¿Cómo funciona Node.js?

## REPL

En una sección anterior pudimos observar cómo ejecutar un archivo de JavaScript con Node.js, con el siguiente comando:

```
node index.js
```

Utilizamos el comando `node` y le enviamos el argumento de `index.js` para ejecutarlo. Pero Node.js no solamente se puede utilizar para ejecutar programas, también se puede utilizar como un programa de línea de comandos para evaluar expresiones; esto se llama REPL (*Read Eval Print Loop*). Para ingresar, sencillamente utilizamos el siguiente comando:

```
node
```

Se pueden escribir expresiones como: `1+1` y presionar la tecla `ENTER` o expresiones como `"A" === "A"` y presionar la tecla `ENTER`. Como su nombre lo dice, por cada uno de los comandos introducidos, realiza las siguientes operaciones:

1. Lee la expresión
2. Evalúa la expresión
3. Imprime el resultado
4. Repite el proceso

Para salir de este ciclo se puede utilizar la combinación de teclas `Ctrl+C` dos veces, también se puede utilizar la combinación de teclas `CTRL+D` o la expresión `process.exit(0)`.

Inclusive se puede escribir un programa completo.

```
1 node (ENTER)
2 var a = 0; (ENTER)
3 if (a === 0) { (ENTER)
4 console.log("a == 0"); (ENTER)
5 } (ENTER)
```



La palabra ENTER se coloca para indicar cuándo se presiona dicha tecla

Se puede notar que al escribir la expresión `if (a === 0) {` y se presiona la tecla ENTER; no se evalúa inmediatamente, ya que en la presencia del `{` indica el comienzo de un bloque y al ingresar `}` se termina evaluando todo el bloque.

Para mayor comodidad, se puede introducir el comando `.editor` para habilitar un editor en la línea de comando para crear el programa con mayor facilidad:

```
1 node
2 > .editor
3 // Entering editor mode (^D to finish, ^C to cancel)
```

Todos los comandos listados a continuación se pueden utilizar dentro de la línea de comandos de REPL y NO en la línea de comandos del sistema:

- La tecla TAB muestra un listado de todas las funciones disponibles.
- El comando `.break` o CTRL+C permite añadir una nueva línea.
- Una vez finalizamos la sesión todas las variables se pierden, pero se puede guardar en un archivo con el siguiente comando `.save [nombre-de-archivo]`.
- De la misma manera, se pueden cargar nuevamente con el comando `.load [nombre-del-archivo]`.
- El listado de otros comandos está disponible con el comando `.help`.

En conclusión, Node REPL es muy útil para probar funcionalidades de manera muy rápida o inclusive depurar un programa.

Más información:

- [Node REPL](#)<sup>18</sup>

## Objeto Global

En una aplicación de Node.js existen varios objetos y funciones globales disponibles en toda la aplicación sin necesidad de incluir ningún módulo. Un ejemplo de estos son:

- `console`: Permite imprimir texto en la consola.
- `process`: Permite acceder a toda la información del proceso de Node.js que se está ejecutando.
- `__filename__` y `__dirname`: Se utilizan para obtener información del directorio y archivo que se está ejecutando.

---

<sup>18</sup><https://nodejs.org/api/repl.html>

- `module`, `exports` y `require`: Se utilizan para exportar e importar módulos.

Existe un objeto global llamado de igual forma `global`. Este objeto en particular es el que se encuentra en el nivel superior del “*scope*” de cada aplicación de Node.js. Una analogía con el entorno del navegador Web sería el objeto `window`. Al tener esta característica muchas veces es utilizado para guardar variables y consultarlas en otras partes de la aplicación, lamentablemente esto no es una buena práctica. No es recomendable el uso de variables globales, pues no se puede tener un control total sobre estas, ya que pueden ser sobrescritas en cualquier parte de la aplicación llevando a estados no deseados.

Más información:

- [Global Objects](#)<sup>19</sup>

## Objeto Process

El objeto `process` provee información acerca del proceso actual en el cual se está ejecutando la aplicación de Node.js. Por lo tanto, es muy importante conocerlo ya que podemos obtener informaciones tales como: versión de Node.js, librerías `core`, directorio de usuario, directorio temporal, el objeto `PATH` y entre otros.

Para ver la información que contiene el objeto `process`:

```
node -p "process"
```



La bandera `-p` evalúa e imprime por pantalla la expresión que se introduce como parámetro

Como se puede observar tiene bastante información, pero se puede acceder directamente a la información específica, ya que en realidad dentro de la aplicación de Node.js es un objeto de JavaScript. Un ejemplo sería la información acerca de las librerías `core`:

```
node -p "process.versions"
```

Más información:

- [process](#)<sup>20</sup>

---

<sup>19</sup><https://nodejs.org/api/globals.html>

<sup>20</sup><https://nodejs.org/api/process.html>

## Argumentos de línea de comando

Cada vez que se ejecuta un programa con Node.js, este se encarga de procesar los argumentos y guardarlos en el objeto `process.argv`. Veamos qué contiene este objeto si se ejecuta la aplicación;

Para ello en nuestra aplicación de `greeting` editamos el contenido del archivo `index.js` y añadimos la siguiente línea al final:

```
console.log(process.argv);
```

Ejecutamos la aplicación con el siguiente comando:

```
node index.js
```

Podemos observar el siguiente resultado en la consola:

```
[ '/Users/.../.npm/versions/node/v12.13.1/bin/node',  
'/Users/.../02-greeting/index.js' ]
```



Se ha modificado la ruta de los directorios para omitir información no relevante y reemplazado con tres puntos seguidos (...)

El objeto `process.argv` es un *Array* que en cada posición almacena cada argumento, en este caso el primero, es la ubicación del ejecutable de Node.js y, el segundo, es el archivo de entrada de la aplicación. Los argumentos se interpretan ya que van separados por espacios.

Modificar el archivo `index.js` de la siguiente manera:

```
1  const args = process.argv.slice(2);  
2  const [name = "Friend"] = args;  
3  const hour = new Date().getHours();  
4  
5  // Ask for hours range  
6  if (hour >= 6 && hour < 12) {  
7    console.log(`Good morning ${name}`);  
8  } else if (hour >= 12 && hour < 18) {  
9    console.log(`Good afternoon ${name}`);  
10 } else if (hour >= 18 && hour < 23) {  
11   console.log(`Good evening ${name}`);  
12 } else {  
13   console.log(`Good night ${name}`);  
14 }
```

A continuación se describen los cambios:

1. Almacenamos en la variable `args` una porción del *Array* de argumentos originalmente de la propiedad `argv`, ignoramos los dos (2) primeros con el método `slice`.
2. Utilizamos *Array Destructuring* para obtener la primera posición del *Array* y si esta no existe le asignamos el valor por defecto de "Friend".
3. Obtenemos la hora local.
4. Finalmente dependiendo del valor de la hora local se imprime un mensaje saludando con el nombre de la persona que se envió como argumento en la ejecución del programa.

Ejecutamos la aplicación con el siguiente comando:

```
node index.js Gustavo
```

[Código fuente de la aplicación](#)<sup>21</sup>

---

<sup>21</sup><https://github.com/gmoralesc/introduccion-a-node-js/tree/master/02-greeting>

# Instalación y administración de paquetes con npm

Junto a la versión de Node.js queda instalado el administrador de paquetes o librerías llamado *npm*; el cual es una utilidad de línea de comandos que permite instalar, actualizar y publicar paquetes para Node.js. Por defecto, *npm* utiliza el registro público (<https://www.npmjs.com/><sup>22</sup>) para hacer todas las operaciones nombradas anteriormente; pero si una organización lo requiere, puede crear su propio registro privado para que sus paquetes no sean públicos y puedan ser consumidos de manera privada.

Por cada instalación de Node.js realizada mediante *nvm*, este instala una versión por defecto de *npm* para comprobar la versión instalada se ejecuta el siguiente comando:

```
npm -v
```

Por ejemplo, si seleccionamos con *nvm* la versión 8 de Node.js, la versión por defecto de *npm* será 5 y si seleccionamos la versión 10 de Node.js, la versión por defecto de *npm* será 6.

Pero lo anterior no significa que no se pueda actualizar la versión de *npm* que está asociada a la versión de Node.js seleccionada con *nvm*; si es necesario se puede actualizar la versión de *npm* con el siguiente comando:

```
npm install npm@latest --global
```

De allí en adelante, esa versión de *npm* estará actualizada para la versión de Node.js seleccionada con *nvm*.

## Instalar paquetes locales

*npm* permite instalar paquetes de manera global, como por ejemplo el mismo *npm* o en general binarios que servirán de utilidades de línea de comandos. Pero el uso más común es instalarlos de manera local para un proyecto. Es importante resaltar que los paquetes locales se instalan en el directorio actual en el que se encuentre la sesión de la terminal; por ende, es siempre recomendado comprobar el directorio en que se encuentra ubicado en la sesión de la terminal antes de ejecutar el comando de instalación.

Si desea conocer más información sobre un paquete instalado o no conoce el nombre específico de algún paquete, puede consultar el registro público de *npm* (<https://www.npmjs.com/><sup>23</sup>).

Continuaremos con nuestro proyecto *greeting*. Para el siguiente ejemplo, instalará el paquete *colors* de manera local con el siguiente comando:

---

<sup>22</sup><https://www.npmjs.com/>

<sup>23</sup><https://www.npmjs.com/>

```
npm install colors
```

Nótese que dentro del directorio actual se ha creado un directorio llamado `node_modules`; este es utilizado por `npm` para almacenar todos los paquetes que se instalen en el directorio del proyecto y este mismo es usado para resolver las dependencias entre los diferentes paquetes. Si observa el contenido del directorio allí se encuentra el paquete instalado

A continuación instale otro paquete llamado `chalk`, que a su vez depende de otros paquetes; esto con el fin de poder observar dónde `npm` instala las dependencias de los paquetes. En el mismo directorio de trabajo ejecutamos el siguiente comando::

```
npm install chalk
```

Se puede ver el contenido del directorio de `node_modules` desde la terminal, con el siguiente comando:

```
ls node_modules
```

Si se observa nuevamente el contenido de `node_modules`, se puede notar que a su vez se han instalado las dependencias del paquete `chalk`, y todas las dependencias están al mismo nivel, este tipo de organización que utiliza `npm` se llama *flat installation*.



En la versión 2 de `npm` tiene `node_modules` dentro del paquete para almacenar todos los paquetes que están relacionadas como dependencias y así consecutivamente.

Para desinstalar un paquete, se utiliza el siguiente comando:

```
npm uninstall chalk
```

Con la instalación (o desinstalación), `npm` analiza los paquetes y sus dependencias entre ellos, creando así un árbol de dependencias, esto es muy útil para mantener la integridad del proyecto. Por lo tanto, si al desinstalar un paquete este tenía asociado una dependencia común con otro paquete, entonces el comando anterior no borra dicha dependencia.

Para más información de `npm` se puede consultar la [guía oficial de uso](https://docs.npmjs.com/getting-started/)<sup>24</sup>.

## Administración de versiones de paquetes

Puede observar que se ha creado un archivo llamado `package-lock.json`, es cual almacena la versión exacta de cada librería que se instala y todas sus dependencias. `npm` garantiza que cada librería que tiene dependencias en común cumpla todos los requisitos, al solucionar cualquier

---

<sup>24</sup><https://docs.npmjs.com/getting-started/>



conflicto que pueda existir entre las diferentes versiones. Una vez realizado este trabajo queda plasmado en el archivo. Esto ofrece una gran ventaja a la hora de que otro usuario vuelva a instalar todas las dependencias del proyecto, ya que la resolución de las dependencias las realizará de manera determinística, garantizando así la instalación de la misma versión de las librerías y sus dependencias. Lo que no podría ser garantizado sin este archivo, ya que a la hora de instalar nuevamente las librerías y dependencias del proyecto pueden existir nuevas o inclusive versiones depreciadas de estas.



El archivo `package-lock.json` está disponible desde la versión 5 de *npm*.

## Utilizar paquetes locales

Vamos a utilizar la librería recientemente instalada, para ello, ubicados en el directorio de trabajo creamos un archivo llamado `index.js` con el siguiente texto:

```
1  const colors = require("colors/safe");
2
3  const args = process.argv.slice(2);
4  const [name = "Friend"] = args;
5  const hour = new Date().getHours();
6
7  // Ask for hours range
8  if (hour >= 6 && hour < 12) {
9    console.log(colors.yellow(`Good morning ${name}`));
10 } else if (hour >= 12 && hour < 18) {
11   console.log(colors.green(`Good afternoon ${name}`));
12 } else if (hour >= 18 && hour < 23) {
13   console.log(colors.cyan(`Good evening ${name}`));
14 } else {
15   console.log(colors.blue(`Good night ${name}`));
16 }
```

Ejecutar la aplicación en la terminal con el siguiente comando:

```
node index.js Gustavo
```



La extensión es opcional, la siguiente línea de comando es equivalente `node index`

Se puede observar en la consola el mensaje tiene el color indicado. Esto lo pudimos lograr ya que en nuestra aplicación requerimos el paquete de `colors`, que fue encontrado automáticamente en nuestro directorio local `node_modules`.

Para conocer más información de la librería `colors` puede consultar su [página web oficial](#)<sup>25</sup>.

[Código fuente de la aplicación](#)<sup>26</sup>

## Instalar y utilizar paquetes globales

Como se dijo antes, los paquetes globales, como su nombre lo sugiere, no se instalan en ningún directorio local, se instalan a nivel global de la versión seleccionada de Node.js con `nvm`. Es decir, que al cambiar de versión de Node.js con `nvm` solo estarán disponibles los paquetes globales instalados de la versión seleccionada.

Al ser globales, pueden ser utilizados en cualquier proyecto o ejecutados desde cualquier directorio seleccionado desde la terminal; lo anterior puede sonar a una enorme ventaja, pero es una práctica NO recomendada, ya que cada proyecto debe tener declarado en su manifiesto (el cual veremos con detalle en la próxima sección) todas los paquetes que necesita para ejecutar, tanto en desarrollo como en producción. En esta sección lo realizaremos a modo de información.

Como ejemplo proceda a instalar el paquete `ESLint` que nos ayuda a comprobar la sintaxis de los archivos en JavaScript. Para instalarlo de manera global, se adiciona la bandera `-g` o `--global`

```
npm install --global eslint
```



Nótese que npm crea un acceso directo de archivo ejecutable del paquete para ser accedido desde la terminal

Creamos en el directorio de trabajo un archivo temporal llamado `tempapp.js`, esto con el fin de añadir el error intencional para mostrar la utilidad del paquete `ESLint`, con el siguiente contenido:

```
1 var ups = ;
```

Ejecutamos ESLint para comprobar la sintaxis de nuestra aplicación temporal:

```
eslint tempapp.js
```



Muchos paquetes globales son inclusive utilizados para crear muchos scripts de flujos de trabajo local en el sistema.

Debemos obtener el siguiente resultado:

---

<sup>25</sup><https://www.npmjs.com/package/colors>

<sup>26</sup><https://github.com/gmoralesc/introduccion-a-node-js/tree/master/03-greeting>

```
1:11 error Parsing error: Unexpected token ;
```

```
▯ 1 problem (1 error, 0 warnings)
```

Para *ESLint* se pueden definir todas las reglas de sintaxis e inclusive extender de una guía de estilos ya existentes. Para mayor información puede visitar la [Guía de inicio de ESLint](#)<sup>27</sup>.

## Usar npx

Desde la versión 8 de Node.js se encuentra disponible la utilidad de *npx* (execute npm package binaries) que permite ejecutar directamente los binarios instalados por los paquetes, de hecho si el binario a ejecutar no existe, este lo instala automáticamente

En el caso anterior de ESLint el comando correspondiente sería:

```
npx eslint tempapp.js
```

Por lo tanto, *npx* va a asegurar que si el paquete *ESLint* no existe de igual manera lo instalará y posteriormente ejecutará el comando indicado.

## Resolución de paquetes

Una interrogante importante es qué pasaría si tenemos un paquete instalado, tanto localmente como globalmente, ¿Cuál versión utilizaría Node.js?. En este caso Node.js busca el paquete en el directorio local del proyecto (*node\_modules*); si no lo encuentra, procede a buscar el paquete en la instalación global y, finalmente, si no lo encuentra muestra un error

## Otros administradores de paquetes

El hecho de que *npm* se instale automáticamente con la instalación de Node.js no quiere decir que sea el único administrador de paquetes. *Yarn*<sup>28</sup> fue lanzado al público en el 2016 como uno de los proyectos *Open Source* de Facebook. Con una gran acogida, su característica principal es la velocidad de instalación de los paquetes debido a un enfoque determinístico.

Yarn utiliza el mismo archivo de manifiesto que *npm* (*package.json*), al igual que el directorio donde se instalan los paquetes (*node\_modules*), por lo que se puede utilizar sin mayor cambios como administrador de paquetes en un proyecto de Node.js. Si es un proyecto existente, se recomienda eliminar el directorio de *node\_modules* y hacer una instalación “limpia”. NO se recomienda utilizar los dos administradores de paquetes *npm* y *Yarn* simultáneamente en el mismo proyecto.

Si se tiene instalado Homebrew en Mac OS, *Yarn* se puede instalar con el siguiente comando:

---

<sup>27</sup><https://eslint.org/docs/user-guide/getting-started>

<sup>28</sup><https://yarnpkg.com/>

```
brew install yarn
```

E inclusive se puede instalar con npm con el siguiente comando:

```
1 npm install -g yarn
```

Para conocer más sobre las opciones de uso de *Yarn* puede visitar la [Guía de uso de Yarn](#)<sup>29</sup>.

---

<sup>29</sup>[/brewinstall yarn](#)

# Inicialización de un proyecto de Node

Normalmente, por cada proyecto de Node.js, se crea un directorio donde se almacenan todos los archivos y paquetes relacionados con el proyecto; No necesariamente un directorio debe estar en blanco para inicializarlo, por lo tanto, continuaremos trabajando con nuestro proyecto `greeting`.

`npm` puede almacenar toda la meta información y administrar los paquetes requeridos, con sus respectivas versiones del proyecto, a través de un archivo de manifiesto. Una de las maneras para crear este archivo manifiesto es con el comando `npm init`, que a su vez mostrará un asistente que nos guiará mediante preguntas con el fin de obtener la información necesaria para nuestro proyecto y así hasta finalizar el proceso: Pero una manera más rápida de hacer este proceso para crear el archivo manifiesto, con todas las opciones que trae por defecto, es con el siguiente comando:

```
npm init -y
```

El resultado es un archivo de texto llamado `package.json`, el cual contiene toda la información básica generada por defecto; estos valores se pueden cambiar solamente editando y guardando los cambios en el archivo. Lo más común y recomendado es que este archivo se encuentre en la raíz del directorio del proyecto.

Opcionalmente, para no tener que cambiar la información del archivo manifiesto cada vez que se crea un nuevo proyecto, es posible establecer valores por defecto para cuando se ejecute el comando `npm init`, como por ejemplo:

```
npm set init.author.name "Gustavo Morales"
npm set init.author.email "gustavo.morales@gmail.com"
npm set init.author.url "http://gmoralesc.me"
```

La otra opción es guardar estos valores en un archivo llamado `.npmrc` en la raíz del directorio del usuario o en la del directorio del proyecto, para ser más específico. Más información acerca del archivo `npmrc`<sup>30</sup>.

Más información:

- [npm documentation - npm init](#)<sup>31</sup>

---

<sup>30</sup><https://docs.npmjs.com/files/npmrc>

<sup>31</sup><https://docs.npmjs.com/cli/init>

# Administración de paquetes con el archivo `package.json`

## Instalación de dependencias

El archivo `package.json` es muy importante, pues será el manifiesto de la aplicación. De ahora en adelante, para instalar un paquete como dependencia en el proyecto añadimos la bandera `-S` o `--save`

Antes de instalar un paquete puede conocer toda la información acerca del mismo con el siguiente comando:

```
npm info colors
```

Instalar la dependencia en el proyecto con el siguiente comando:

```
npm install -S colors
```

Si observamos el archivo `package.json` se ha añadido la siguiente sección:

```
"dependencies": {  
  "colors": "^1.4.0"  
}
```

Indica el listado de dependencias del proyecto y la versión instalada de cada paquete. Si otra persona copiara este proyecto e instalará las dependencias (con el comando `npm install`) tendría la versión mínima del paquete `colors` equivalente a `1.4.0`. El carácter `^` al inicio de la versión del paquete indica que se puede instalar cualquier *upgrade* o *patch* de la versión 1, pero no instalará ninguna versión con el *mayor release* 2; en otras palabras, podrá instalar `1.4.0 <= x < 2.0.0`. Esta nomenclatura es llamada [semver](https://github.com/npm/node-semver)<sup>32</sup> y permite establecer las reglas para las versiones de los paquetes del proyecto.

Para listar todas las dependencias que tiene el proyecto con el siguiente comando:

```
npm list
```

O alguna en particular se agrega el nombre de la dependencia con el siguiente comando:

```
npm list colors
```

---

<sup>32</sup><https://github.com/npm/node-semver>

## Instalación de paquetes para desarrollo

Existen paquetes que no son dependencias, pero son necesarias en el entorno local de desarrollo; por ejemplo el paquete *ESLint* permite comprobar la sintaxis de los archivos JavaScript basado en un conjunto de reglas. Esto puede ser muy útil a la hora de estandarizar el código seleccionando o creando una guía de estilo. Para incluirla en nuestro manifiesto como paquete de desarrollo se incluye la bandera `-D` o `--save-dev`.

```
npm install -D eslint
```

Muchos subcomandos de *npm* tienen accesos directos, en el anterior se puede reemplazar `install` por la abreviación `i`, es decir: `npm i -D eslint`

Si observamos nuevamente el archivo `package.json`, está incluida una nueva sección para las paquetes que se van a utilizar en el entorno de desarrollo; de esta manera es como *npm* distingue las dependencias necesarias a instalar en ambientes de producción.

```
"devDependencies": {  
  "eslint": "^6.7.1"  
},
```

## Configuración de git y Github

Es muy importante tener en cuenta que si se está utilizando un sistema de control de versión se debe ignorar el directorio de `node_modules`, pues este no pertenece al código fuente del proyecto y además puede variar dependiendo de la versiones de los paquetes que se realice en cada instalación.

Si estamos utilizando *git* en la raíz del directorio del proyecto se crea un archivo llamado `.gitignore` y se coloca las siguientes líneas:

```
1 node_modules/  
2 .DS_Store  
3 Thumbs.db
```

Con lo anterior, se establece que los archivos o directorios que cumplan con este patrón no serán incluidos en el repositorio de *git*. Aquí también incluimos los archivos como `.DS_Store` en el caso de Mac OS o `Thumbs.db` en el caso del sistema Windows, los cuales son archivos temporales para la generación de la vista previa (*Thumbnails*) de los directorios del sistema.

Si el proyecto se va a alojar en un repositorio público como [Github](https://github.com)<sup>33</sup> o [Bitbucket](https://bitbucket.org/)<sup>34</sup> es muy recomendable crear un archivo llamado `README.md` con la descripción del proyecto; pues una vez

---

<sup>33</sup><https://github.com>

<sup>34</sup><https://bitbucket.org/>

publicado, el repositorio mostrará información preliminar de este mismo. La sintaxis con la que se escribe este archivo es Markdown.

Más información:

- [Markdown](#)<sup>35</sup>

---

<sup>35</sup><https://guides.github.com/features/mastering-markdown/>



# Conclusión

Con los conocimientos básicos de línea de comandos y del lenguaje JavaScript fue suficiente para explorar la versatilidad de Node JS mediante los diferentes temas y ejemplos. Es importante conocer cuáles son sus convenciones y patrones de diseño, que ha adaptado abiertamente la comunidad como CPS, métodos asincrónicos y código no bloqueante.

En la página de Node.js se provee una excelente [guía sobre principales los conceptos](https://nodejs.org/en/docs/guides/)<sup>75</sup> y en la [documentación del API](https://nodejs.org/api/)<sup>76</sup> suministra en detalle el funcionamiento de cada una de las funciones.

Node.js es un excelente programa de escritorio que puede ser la solución para diferentes tipos de proyectos, pero no es la panacea para todos los proyectos. Node.js no es tan fuerte para el cálculo intensivo de datos, para ello se recomienda utilizar otras soluciones especializadas.

En la arquitectura Web Node.js se utiliza mucho como middleware para servir de intermediario entre el Frontend y Backend, para diferentes tareas como: renderizado del lado del servidor, cache, manejo de sesión de usuario y demás. Otro de los usos más frecuentes de Node.js es utilizarlo como Backend para crear REST API con o sin persistencia de datos. Para ello, los invito a leer mi siguiente libro:

Creando APIs con Node JS, Express y Mongo DB

Muchas gracias

---

<sup>75</sup><https://nodejs.org/en/docs/guides/>

<sup>76</sup><https://nodejs.org/api/>