INTRODUCCIÓN A NODE.JS

Conoce las capacidades de una de las plataformas más populares.

GUSTAVO MORALES

Tabla de contenido

Portada

Introducción 1.1

¿Que es Node.js? 1.2

Configurando Node

Instalación y configuración 2.1

Instalación y administración de paquetes con npm 2.2

Inicialización de un proyecto de Node 2.3

Configuración de npm scripts 2.4

¿Como funciona Node.js?

Utilizando Node REPL 3.1

Objeto Global 3.2

Objeto Process 3.3

Argumentos desde la línea de comandos 3.4

Convenciones utilizadas en Node JS

Funciones como objetos de primera clase en JavaScript 4.1

Callbacks y CPS 4.2

Convenciones de callback en Node JS 4.3

Módulos 4.4

Persistencia de datos

Leer y escribir archivos 5.1

Archivos JSON 5.2

Interactuando con la consola 5.3

Depurando aplicaciones

Depurando con Node JS 6.1
Depurando con Chrome Dev Tools 6.2
Depurando con VS Code 6.3

Patrones de flujo asincrónico en Node JS

Código bloqueante y código no bloqueante 7.1
El Event loop 7.2
El patrón observer y Event Emitter 7.3

Servidor Web

Renderizar paginas Web 8.1
Renderizar vistas dinámicas 8.2
Utilizar Express JS 8.3
Conclusión 8.4

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.

Si es nuevo en el lenguaje JavaScript le recomiendo tomar el siguiente curso:

- Introduction to JavaScript - Code Academy

Adicionalmente en este libro se tratarán conocimientos de las funcionalidades de ES2015 al menos:

- *Arrow functions*
- *let + const*
- *Template Strings*
- *Destructuring assignment*
- *Promises*

Para mayor información, el lector puede consultar el siguiente enlace:

- Learn ES2015 - Babel

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.

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.

Gustavo Morales

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: https://www.uninorte.edu.co/web/decastro

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 Alquería, 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 https://medium.com/@zorrodg

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.

¿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.

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, 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

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

¿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 sobreescribir 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 y configuración

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. 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.

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 está la información detallada sobre el proceso y los posibles problemas asociados a la misma.

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 está toda la información detallada sobre el proceso de instalación y los posibles problemas asociados con la misma.

Instalación en Windows

Instalación de cmder

Versiones anteriores a Windows 10 ofrecen una terminal muy básica; pero el proyecto *cmder* 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 cmder
- 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 cmder.exe
- Se recomienda ejecutar el programa con permisos de Administrador

En el repositorio oficial de cmder está toda la información detallada sobre el proceso de instalación y los posibles problemas asociados a la misma.

Instalación de *nvm* for 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
- 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 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 https://nodejs.org/dist/v10.4.0/node-v10.4.0-darwin-x64.tar.gz...
######################################################################## 100.0%
Computing checksum with shasum -a 256
Checksums matched!
Now using node v10.4.0 (npm v6.1.0)
```

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:

```
v10.4.0
```

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; 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 8
```

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:

```
        v6.11.5
->      v8.11.2
        v10.4.0
default -> 6 (-> v6.11.5)
node -> stable (-> v10.4.0) (default)
stable -> 10.4 (-> v10.4.0) (default)
iojs -> N/A (default)
lts/* -> lts/carbon (-> v8.11.2)
lts/argon -> v4.9.1 (-> N/A)
lts/boron -> v6.14.2 (-> N/A)
lts/carbon -> v8.11.2
```

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, la versión `argon` solo recibe actualizaciones de mantenimiento y la versión `boron` tiene activo el soporte 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:

```
8
```

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.

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/) 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/).

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 "playground" e ingresamos a él con el siguiente comando:

```
mkdir playground && cd playground
```

Para el siguiente ejemplo, instalará el paquete `date-fns` de manera local con el siguiente comando:

```
npm install date-fns
```

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.

Administración de versiones

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 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 crear una mini aplicación que cada vez que la ejecutamos nos imprima la fecha actual en un formato especial; por lo tanto, ubicados en el directorio de trabajo creamos un archivo llamado `dateapp.js` con el siguiente texto:

```
const dateFns = require("date-fns");

console.log( dateFns.format(new Date(), "YYYY/MM/DD") );
```

Procedemos a ejecutar la mini aplicación en la terminal con el siguiente comando:

```
node dateapp.js
```

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

Se puede observar en la consola la fecha con el formato aplicado. Esto lo pudimos lograr ya que en nuestra mini aplicación requerimos el paquete de moment, que fue encontrado automáticamente en nuestro directorio local `node_modules`.

Para conocer más información de la librería *date-fns* puede consultar su página web oficial.

Código fuente de la aplicación

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 manifesto (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:

```
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:

```
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.

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.

Código fuente de la aplicación

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 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:

```
brew install yarn
```

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

```
npm install -g yarn
```

Para conocer más sobre las opciones de uso de *Yarn* puede visitar la Guia de uso de 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; por lo tanto, nos trasladamos nuevamente al directorio del usuario (en realidad puede ser cualquier ubicación) y allí creamos el directorio del proyecto:

```
cd ~
```

```
mkdir greeting && cd 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.

Más información:

- npm documentation - npm 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 date-fns
```

Instalar la dependencia en el proyecto con el siguiente comando:

```
npm install -S date-fns
```

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

```
"dependencies": {
    "date-fns": "^1.30.1"
}
```

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 `date-fns` equivalente a `1.30.1` . 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 `1.30.1 <= x < 2.0.0` . Esta nomenclatura es llamada semver 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 date-fns
```

Procedemos a instalar otra dependencia con el siguiente comando:

```
npm install -S colors
```

Más información

- colors

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`

Recuerde que para que *ESLint* funcione dentro del proyecto se deben especificar las reglas de comprobación de sintaxis o al menos el archivo vacío `.eslintrc` en la raíz del proyecto con el siguiente contenido:

```
{}
```

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": "^4.17.0"
}
```

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:

```
node_modules/
.DS_Store
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 o Bitbucket es muy recomendable crear un archivo llamado `README.md` con la descripción del proyecto; pues una vez publicado, el repositorio mostrará información preliminar de este mismo. La sintaxis con la que se escribe este archivo es Markdown.

Más información:

- Markdown

Configuración de npm scripts

Cuando estemos trabajando con el proyecto vamos a repetir una y otra vez los mismos comandos en la terminal. Estos comandos se pueden automatizar para agilizar el trabajo, y *npm* lo hace a través de los *scripts*. Si observamos nuevamente el contenido del archivo `package.json` podrá observar la siguiente sección:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

En esta sección se pueden especificar los comandos que se desean utilizar en la terminal. En el texto anterior está configurado para que el *script* `test` imprima una cadena en la pantalla, indicando que aún no se han configurado las pruebas. Este *script* se puede ejecutar de la siguiente manera:

```
npm test
```

Scripts incorporados

npm tiene varios alias de *scripts* ya incorporados para trabajar, como por ejemplo `test`, que vimos en la sección anterior. Así como su nombre lo indica, se utiliza para ejecutar las pruebas del proyecto. Uno casi mandatorio es el *script* `start` que se ha convertido es una convención para iniciar los proyectos en Node.js, el cual configuraremos más adelante.

Antes de agregar el *script* tengamos en cuenta que en la llave `main` del archivo `package.json` está declarando cual será el archivo de entrada por defecto del proyecto para ejecutarlo, este se llama `index.js`. Cree ese archivo en blanco primero con el siguiente comando:

```
touch index.js
```

> Por convención, se llama `index.js` pero puede ser cualquier nombre e inclusive ubicación, no necesariamente en la raíz del proyecto

Ahora modifiquemos la sección de `scripts` del archivo `package.json`

```
"scripts": {
  "start": "node index.js",
  "test": "echo \"Error: no test specified\"&& exit 1"
},
```

De ahora en adelante, para iniciar nuestro proyecto de Node.js ejecutemos el siguiente comando:

```
npm start
```

Scripts personalizados

Es posible crear *scripts* personalizados para ejecutar diferentes tipos de tareas. En la sección anterior, instalamos y utilizamos el paquete global *ESLint*, agreguemos un *script* personalizado que compruebe los archivos del proyecto::

```
"scripts": {
  "start": "node index.js",
  "test": "echo \"Error: no test specified\"&& exit 1",
  "lint": "eslint **.js"
},
```

Para ejecutar esta tarea personalizada esta vez añadimos el prefijo `run`, pues no es un *script* incorporado:

```
npm run lint
```

Para listar todas las tareas que se pueden ejecutar con *npm* podemos ejecutar el siguiente comando:

```
npm run
```

Convenciones de *npm* en un proyecto

Normalmente todos los proyectos que utilicen *npm* como administrador de paquetes y dependencias, utilizan la siguiente convención cada vez que se descarga o clona un proyecto:

1. `npm install`

2. `npm start`

Y si desea ejecutar las pruebas `npm test` inclusive lo más recomendado es incluir estas instrucciones en el archivo `README.md` .

Utilizando las dependencias y scripts

Editamos el archivo index.js con el siguiente contenido:

```
// Require libraries
const format = require('date-fns/format');
const colors = require('colors/safe');

// Get hour number in 24 format
const hour = format(new Date(), "H");

// Ask for hours range
if (hour >= 6 && hour < 12) {
  console.log(colors.blue('Good morning'));
} else if (hour >= 12 && hour < 18) {
  console.log(colors.yellow('Good afternoon'));
} else if (hour >= 18 && hour < 23) {
  console.log(colors.gray('Good evening'));
}
```

Este sencillo programa consta de tres partes:

1. Requerir las dependencias que fueron previamente instaladas
2. Obtener la hora actual del sistema y almacenarlo en una variable
3. Dependiendo del rango de la hora, mostrar en la consola un mensaje de saludo con su color respectivo

Ejecutar el programa se hace con el siguiente comando:

```
npm start
```

Código fuente de la aplicación

Pre y post scripts

Es posible tomar partida de la convención de los *scripts* de *npm* para ejecutar otros *scripts* antes y/o después de otro *script* por ejemplo:

```
"scripts": {
  "postinstall": "npm test",
  "prestart": "npm run lint",
  "start": "node index.js",
  "test": "echo \"Error: no test specified\"&& exit 1",
  "lint": "eslint **.js"
},
```

En el ejemplo anterior se especifica que luego de ejecutar el *script* `npm install` se ejecutarán automáticamente los comandos (no está limitado solo a ejecutar otros *scripts*) especificados en el *script* `postinstall`. Y antes de ejecutar el *script* `npm start` se ejecutarán automáticamente los comandos especificados en el *script* `prestart`. No importa el orden en que se coloquen en el archivo `package.json` dentro de la sección de *scripts*; lo importante es la nomenclatura.

Utilizando Node.js REPL

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

```
node dateapp.js
```

Utilizamos el comando `node` y le enviamos el argumento de `dateapp.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

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

```
node (ENTER)
var a = 0; (ENTER)
if (a === 0) { (ENTER)
 console.log("a == 0"); (ENTER)
} (ENTER)
```

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:

```
node
> .editor
// 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

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` , `process` , `module` , `exports` , `require` , `__filename` , `__dirname` . El primero de estos `console` ya lo hemos utilizado para escribir mensajes de texto como salida de la consola.

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 sobreescritas en cualquier parte de la aplicación llevando a estados no deseados.

Más información:

- Global Objects

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 es un objeto de JavaScript. Un ejemplo sería la información acerca de las librerías *core*:

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

Pero no solo se puede consultar información, también se puede almacenarla, ya que la interfaz para leer y escribir información en este objeto es igual a la de un objeto de JavaScript, todo este facilitado por Node.js.

Uno de las principales usos para guardar información en el objeto *process* es almacenar variables de entorno, para lo cual se utiliza el objeto `process.ENV`. Como por ejemplo el puerto por defecto en que se ejecutará la aplicación.

```
const port = process.ENV.PORT || 3000;
console.log(port);
```

En el código anterior, nuestra aplicación espera encontrar en el proceso una variable `process.ENV.PORT` para asignar cuál es el valor del puerto donde se ejecutará nuestra aplicación; pero en el caso de no existir, se le asignaría el valor de `3000` por defecto.

Lo interesante es que estas variables se pueden establecer en el sistema y no en el código fuente de la aplicación, lo cual brinda un nivel de configuración dependiendo del entorno de ejecución. Por ejemplo: podemos establecer un puerto de manera local para desarrollo, pero una vez la aplicación se publique en el servidor podemos establecer un puerto

totalmente diferente. Lo cual nos lleva a otro objeto importante y bastante utilizado `process.NODE_ENV`, en el que establecemos cual es el entorno en el que se está ejecutando la aplicación, por ejemplo: *development*, *test* o *production*.

En sistemas **nix* podríamos asignarla directamente en el *npm script* de la siguiente manera:

```
"scripts": {
    "dev":   "NODE_ENV=development node index.js",
    "start": "NODE_ENV=production  node index.js",
    "test":  "NODE_ENV=test echo \"Error: no test specified\"&& exit 1",
    "lint":  "eslint **.js"
},
```

En el ejemplo anterior, se establece el valor de la variable de entorno dependiendo del entorno que estemos ejecutando. Esta variable puede ser leída dentro de nuestra aplicación y, por ejemplo, determinar qué configuración se carga o cómo se imprimen los logs.

Lamentablemente, el código anterior no funciona en entornos Windows. Para asegurar que funcione en diferentes sistemas operativos se usará una librería de *npm* llamada `cross-env`. La instalamos como una dependencia de nuestra proyecto, para asegurar que siempre esté disponible de la siguiente manera:

```
npm i -S cross-env
```

Entonces, finalmente, para utilizarla modificaremos nuestros *npm scripts* de la siguiente manera:

```
"scripts": {
    "dev":   "cross-env NODE_ENV=development node index.js",
    "start": "cross-env NODE_ENV=production  node index.js",
    "test":  "cross-env NODE_ENV=test echo \"Error: no test specified\"&& exit 1",
    "lint":  "eslint **.js"
},
```

Más información:

- process
- cross-env

Argumentos desde la línea de comandos

Cada vez que ejecutamos una aplicación de Node.js estamos enviando al menos un argumento al ejecutable de Node.js (`node`), este argumento es el archivo de entrada de la aplicación (`index.js`).

Vamos a crear un nuevo proyecto llamado `bmi-calculator` y procedemos a iniciarlo; a continuación, cada uno de los comandos utilizados para inicializar un proyecto de Node.js, como se hizo en el capítulo anterior:

1. Ir al directorio de trabajo
2. Crear e ir al directorio del proyecto: `bmi-calculator`
3. Iniciar el proyecto de Node.js con *npm*: `npm init -y`
4. Crear el archivo principal de entrada de la aplicación: `touch index.js`
5. Añadir en el listado de *scripts*: `start: "node index"`
6. Es recomendado iniciar el archivo que contiene la lista de archivos y/o directorios a ignorar si se utiliza *git* .

Node.js 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 editamos el contenido del archivo `index.js` con la siguiente línea:

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

Ejecutamos nuestra aplicación con el siguiente comando:

```
node index.js
```

Podemos observar el siguiente resultado en la consola:

```
[ '/Users/.../.nvm/versions/node/v8.1.2/bin/node',
  '/Users/.../bmi-calculator/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. A continuación ejecutemos las siguientes líneas en la terminal:

```
node index Gustavo 1.7 76
node index Gustavo Morales 1.7 76
node index "Gustavo Morales" 1.7 76
node index name=Gustavo height=1.7 weight=76
node index -n "Gustavo Morales" -h 1.7 -w 76
```

Como se puede observar, Node.js separa los argumentos por el espacio y el orden de almacenamiento es el mismo orden de entrada; es decir, que para el segundo ejemplo sería incorrecto enviar el nombre separado por espacio pues sería interpretado como 2 argumentos. La manera correcta sería como está el tercer ejemplo. Una forma de crear argumentos nombrados puede ser el cuarto ejemplo, luego tocaría separar la llave (*key*) del valor (*value*) y, por último, está la manera tradicional como se utilizan los argumentos nombrados, pero como podemos observar sería bastante trabajo tener que interpretar lo que queda almacenado en `process.argv`, por lo cual utilizaremos una librería llamada `command-line-args`, la instalamos como una dependencia del proyecto con el siguiente comando:

```
npm i -S command-line-args
```

Sustituimos el contenido de nuestro archivo `index.js` con el siguiente fragmento de código:

```
const commandLineArgs = require('command-line-args');

const params = [
  { name: 'name', alias: 'n', type: String },
  { name: 'height', alias: 'h', type: Number },
  { name: 'weight', alias: 'w', type: Number }
];

const options = commandLineArgs(params);

console.log(options);
```

Y ejecutamos nuevamente nuestra aplicación con los siguientes argumentos:

```
node index -n "Gustavo Morales" -h 1.7 -w 76
```

Como se puede observar, en la salida de la consola la librería se ha encargado de interpretar todos los argumentos, según las opciones que le establecimos, y ha creado un objeto de JavaScript donde la llave corresponde al nombre del argumento y se le asigna su valor respectivo:

```
{ name: 'Gustavo Morales', height: 1.7, weight: 76 }
```

Finalmente, reemplazamos la parte final del código para calcular el BMI basado en los argumentos y dar un respuesta apropiada:

```
...

var options = commandLineArgs(params)
var bmi = options.weight / Math.pow(options.height, 2);
var result = '';

if (bmi < 18) {
  result = 'Peso bajo';
} else if (bmi >= 18 && bmi < 25) {
  result = 'Normal';
} else if (bmi >= 25) {
  result = 'Sobrepeso';
}

console.log(`${options.name} el resultado de tu BMI es ${bmi} lo que indica: ${result}`
);
```

Y ejecutamos nuevamente la aplicación:

```
node index -n "Gustavo Morales" -h 1.7 -w 76
```

También es posible seguir enviando los argumentos a través del *npm script*; en este caso agregamos el script de start en el `package.json` :

```
"scripts": {
    "start": "node index",
    "test": "echo \"Error: no test specified\" && exit 1"
},
```

Y para enviar los argumentos se debe añadir doble guión (`--`):

```
npm start -- -n "Gustavo Morales" -h 1.7 -w 76
```

Finalmente tendrá el mismo resultado.

Código fuente de la aplicación

Es muy importante validar los argumentos que sean válidos pues en este ejercicio, y en los siguientes de este tipo, se asume que el usuario ingresa los argumentos en forma correcta. Para practicar y profundizar puede hacer sus propias comprobaciones.

Más información:

- command-line-args