

INTRODUCCIÓN A NODE.JS

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



.js

GUSTAVO MORALES

Tabla de contenido

Portada

Introducción	1.1
¿Qué 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?

Utilizar 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

Depurando aplicaciones

Patrones de flujo asíncronico en Node JS

Servidor Web

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](#)

Acercas 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 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 <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 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 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](https://brew.sh/) 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](https://github.com/nvm-sh/nvm) 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](#)
- 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](#) está toda la información detallada sobre el proceso de instalación y los posibles problemas asociados a la misma.

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](#)
- 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, instalaré 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 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:

```
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 [Guía 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 un proyecto que incluyen *npm*

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 pos *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.

Utilizar 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 sobrescritas 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 nuestro 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 una 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](#)

Funciones como objetos de primera clase en JavaScript

En JavaScript las funciones son *first class citizens*; esto significa que las funciones pueden ser asignadas a variables, ser enviadas como parámetros a otra función y ser retornadas como funciones también.

A continuación se mostrará un *function statement* comúnmente utilizado en otros lenguajes de programación:

```
function sum(a,b){  
  return a+b;  
}  
  
console.log(sum(40,2)); //42
```

Pero en JavaScript se pueden crear *function expression* y asignárselas a una variable, lo mostraremos con el siguiente ejemplo:

```
const sum = function(a,b){  
  return a+b;  
};  
  
console.log(sum(40,2)); //42
```

Existen dos importantes diferencias, primero, la función en sí es una *anonymous function*; es decir, que no tiene ningún nombre, pero es asignada a una variable (desde donde podremos referenciarla luego) y, segundo, la declaración se ha convertido en una sentencia terminada en punto y coma como cualquiera otra línea de programación.

Inclusive podemos asignar la función (de hecho la referencia) a otra variable :

```
const sum = function(a,b){  
  return a+b;  
};  
  
const sumFunc = sum;  
  
console.log(sumFunc(40,2)); //42
```

En el código anterior utilizamos una *anonymous function*, pero también se puede utilizar una *named function* si se quisiera, por ejemplo, para recursividad, tómese en cuenta el siguiente ejemplo:

```
const factorial = function ff(number) {  
  if (number <= 0) {  
    return 1;  
  } else {  
    return (number * ff(number-1));  
  }  
};  
  
console.log(factorial(6)); // 720
```

También es posible asignar una función como una propiedad de un objeto en JavaScript:

```
const calculator = {  
  total: 0,  
  sum: function (a,b){  
    return a+b;  
  }  
};
```

Y porqué no, almacenar una función en una posición de un *Array* y llamarla desde allí:

```
const sum = function(a,b){  
  return a+b;  
};  
  
let stack = [];  
  
stack.push(sum);  
  
const next = stack[0];  
  
next(40, 2); // 42  
  
// También es posible llamar la función directamente desde el Array y pasarle los  
// argumentos  
  
stack[0](40, 2); // 42
```

Como *first class citizens* podemos pasar una función como parámetro dentro de otra función y utilizarla dentro de esta segunda función, tome como ejemplo el siguiente código::

```
var sum = function(a,b){  
  return a+b;  
};  
  
function calc(a,b,op){  
  var result = op(a,b);  
  return result;  
}  
  
calc(40,2,sum); // 42
```

Finalmente, también podemos retornar una función a partir de otra función, pero ¿Por qué esto es tan importante en Node.js? Por un patrón llamado *callback*, este estilo es llamado *Continuation-passing style (CPS)* el cual revisaremos en detalle en la siguiente sección.

Callbacks y the continuation-passing style (CPS)

Los *callbacks* son funciones invocadas para propagar el resultado de una operación una vez esta es completada,; este estilo se conoce como *The continuation-passing style (CPS)* y es lo que se utiliza en operaciones asincrónicas. JavaScript es un excelente lenguaje para utilizar *callbacks*, porque como se vió anteriormente, las funciones son objetos de primera clase y pueden ser fácilmente asignadas a variables, pasadas como argumentos e inclusive ser retornadas de otra función.

Analicemos el siguiente código:

```
function sumSync(a, b, callback){  
    callback(a+b);  
}
```

El código anterior es una función sincrónica que utiliza el estilo CPS, ya que una vez termine la función principal su continuación se realizará a través de la función llamada *callback*, lo cual se demuestra en el siguiente código:

```
console.log("before");  
  
sumSync(40, 2, function(result){  
    console.log("Result:", result);  
});  
  
console.log("after");
```

El resultado será:

```
before  
Result 42  
after
```

Asynchronous continuation-passing style

A continuación se creará una función asincronica:

```
function sumAsync(a, b, callback){
  setTimeout( function(){
    callback(a+b)
  },
  1000);
}
```

En el código anterior, utilizamos la función `setTimeout()` para simular un llamado asíncrono y, luego de 1 segundo, invocar la función. Ahora vuelva a ejecutar el código utilizando la función creada anteriormente:

```
console.log("before");

sumAsync(40, 2, function(result){
  console.log("Result:", result);
});

console.log("after");
```

El resultado será:

```
before
after
Result 42
```

Después de 1 segundo la función `setTimeout` ejecuta nuestra función y no espera hasta que la función de *callback* sea ejecutada; por el contrario, devuelve el control a la función `sumAsync` para que se siga ejecutando. Este comportamiento es muy importante en Node.js para no bloquear el programa y que pueda seguir realizando las demás operaciones.

Convenciones de *callback* en Node.js

Los métodos del API de Node.js que utilizan los *callbacks* y CPS, siguen unas convenciones muy específicas. Estas convenciones son aplicadas también en la mayoría de código creado por los usuarios, por lo cual es muy importante conocerlas y aplicarlas.

Los *callbacks* van al final

En todos los métodos de Node.js que aceptan un *callback* como parámetro, este es pasado siempre como último argumento. Veamos como ejemplo la versión asíncronica de la función

`readFile` :

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

fs.readFile('data.json', 'utf8', function(err, content){
  if(err) throw err;
  console.log(content);
});
```

Como pueden observar el *callback* una vez se ha leído el archivo es el último argumento de la función.

Los errores van de primero

En CPS los errores pueden ser propagados como un tipo de resultado de la función invocada, en Node.js; es pasado como primer argumento y de allí en adelante se pasan los demás parámetros a la función. En una operación sin errores, este toma el valor de *null* o *undefined*. Analicemos nuevamente el mismo fragmento de código:

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

fs.readFile('data.json', 'utf8', function(err, content){
  if(err) throw err;
  console.log(content);
});
```

Como puede observar, el parámetro de error (`err`) es enviado de primero en la función de *callback* y, de segundo, el contenido del archivo leído.

Módulos

Los módulos son una de los aspectos más poderosos de Node.js. Permiten organizar el código estableciendo así las mejores prácticas, tales como: separar la responsabilidad, reutilizar el código y ejecutar pruebas independientes.

Los módulos de Node.js utilizan una nomenclatura llamada CommonJS que es diferente a la nomenclatura usada en el navegador con la última definición de ECMAScript 2015 (o comúnmente conocido como ES6), pero ambos cumplen la misma función. Aunque en el mundo de Node.js es más común el uso de CommonJS también se pueden utilizar los ES6 modules con ayuda de un *transpiler* de código como *Babel* o de manera nativa en las versiones recientes de Node.js. En esta sección se discutirán los módulos en notación CommonJS.

En las secciones anteriores ya habíamos "importado" módulos, como:

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

El uso de la función `require` es el que permite importar el módulo y "almacenarlo" en la variable establecida para utilizarlo en el programa. Finalmente, un módulo termina siendo un archivo que puede tener variables y funciones y este a su vez puede tener más módulos. Veamos el siguiente ejemplo:

```
const sum = require("./sum");
```

A diferencia del primer ejemplo, el anterior tiene el prefijo `./` el cual indica que es un módulo local del programa o creado por el usuario; es decir, que Node.js no lo buscará en el directorio de `node_modules` o, en otras palabras, un módulo dentro de la misma ruta del archivo que lo requiere. Las rutas especificadas en el `require` son relativas; por lo tanto, si el módulo requerido está en otro nivel, por ejemplo el módulo requerido se encuentra en un nivel superior y dentro de otro directorio llamado `utils`, se escribiría de la siguiente manera:

```
const sum = require("../utils/sum");
```

Nótese que los archivos no tienen la extensión `.js`, la cual no es necesaria; el sistema asume que tienen esta extensión.

Creemos un módulo simple: cree un archivo llamado `sum.js` y coloque el siguiente código:

```
function sum(x, y) {  
  return x + y;  
}  
  
module.exports = sum;
```

Este módulo contiene una función que recibe dos argumentos y devuelve la suma de estos, pero la palabra especial `module.exports` permite "publicar" esa función para que pueda ser utilizada fuera del módulo.

Node.js encapsula cada módulo en la siguiente función llamada *The module wrapper*:

```
(function(exports, require, module, __filename, __dirname) {  
  // Module code actually lives in here  
});
```

Gracias a esto se tiene acceso en cada módulo a las variables enviadas como argumentos automáticamente por Node.js con sus respectivos valores del contexto.

A continuación se crea otro archivo llamado `app.js` en el mismo directorio de `sum.js`, para utilizar el módulo creado:

```
const sum = require("./sum");  
  
console.log( sum(40,2) );
```

Ejecutamos la aplicación:

```
node app.js
```

El módulo creado puede ser reutilizado, puede ser probado independiente del contexto donde se requiere y solo tiene una responsabilidad. Ahora bien, un módulo no solamente puede tener una sola función; para demostrarlo renombramos el archivo de `sum.js` por `math.js` y agregar el siguiente código:

```
const pi = 3.141593;

function sum(x, y) {
  return x + y;
}

function subtraction(x, y) {
  return x - y;
}

function circleArea(r){
  return pi*r*r;
}

module.exports = {
  sum: sum,
  sub: subtraction,
  pi: pi
};
```

Se pueden observar diferentes cambios en este nuevo módulo:

- Se pueden exportar funciones y variables.
- Es posible establecer un alias con el que se exporta la función o variable.
- No es necesario exportar todas las funciones o variables del módulo.
- Las funciones o variables que no se exporten no estarán disponibles cuando se importe el módulo.

Una sintaxis equivalente a `module.exports` para el módulo de `math.js`, es la siguiente:

```
module.exports.sum = function(x, y) {
  return x + y;
}
```

Finalmente, cuando se "compila" el módulo, el objeto `exports` tiene toda la información pública del mismo.

A continuación se modifica `app.js` para importar el nuevo módulo:

```
const mathFuncs = require("./math");

console.log( mathFuncs.sum(40, 2) );

console.log( mathFuncs.sub(100, 58) );

console.log( mathFuncs.pi );
```

Nuevamente podemos observar varios cambios:

- La variable que requiere el módulo no necesariamente se debe llamar igual que el módulo .
- La variable `mathFuncs` es un objeto con varias propiedades y funciones al que se puede acceder mediante la sintaxis de punto (`.`) .

Ejecutamos la aplicación:

```
node app.js
```

Se recomienda establecer una buena estructura de directorios para organizar los módulos de la aplicación y se puede aprovechar la convención del archivo `index.js` . Es decir, si dentro del directorio de la aplicación tengo un directorio llamado `utils` y dentro un módulo llamado `index.js` se podría importar de la siguiente manera:

```
const utils = require("./utils");
```

Si no existe el archivo `utils.js` en el directorio actual del archivo que requiere el módulo, el sistema buscará un directorio llamado `utils` y dentro un módulo llamado `index.js`

Más información:

- [Node JS - Modules](#)
- [Node JS - The module wrapper](#)
- [CommonJS](#)
- [Babel](#)
- [Babel / Node](#)

Leer y escribir archivos

Recordemos que Node.js se ejecuta del "lado del servidor"; por lo tanto, tiene acceso a todos los recursos físicos de la máquina a los cuales tenga permiso, esto incluye leer y escribir archivos del sistema. Node.js tiene un módulo nativo para manipular archivos llamado `fs` (*FileSystem*).

Cree un nuevo proyecto llamado `bucket-list` y proceda a iniciarlo.

Vamos a utilizar la librería de `command-line-args`, así que se procede a instalarla como dependencia:

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

Editamos el contenido del archivo `index.js` con el siguiente contenido:

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

const params = [
  { name: 'item', alias: 'i', type: String },
  { name: 'times', alias: 't', type: Number },
  { name: 'frequency', alias: 'f', type: String }
];

const options = commandLineArgs(params);
console.log(`You want to ${options.item} ${options.times} time(s) every ${options.frequency} `);
```

Y ejecutamos nuestra aplicación:

```
node index -i "Jogging" -t 3 -f "Week"
```

```
You want to Jogging 3 time(s) every Week
```

Como podemos observar, nuestros datos se perderían una vez que termina de ejecutarse la aplicación; por consiguiente, debemos utilizar la persistencia de datos, en este caso guardar cada línea en un archivo en el directorio del proyecto, para lo cual vamos a incluir:

- La librería *core* de `fs` no solamente lee y escribe archivos, tiene una gran variedad de funciones para manipular buffers y streams.
- La librería *core* de `os` tiene mucha información del sistema operativo entre ella EOL,

que es el salto de línea dependiendo del sistema operativo sea *nix o Windows.

- La librería `core path` nos ayudará a resolver todas las rutas del sistema y establecer una ruta para el archivo donde vamos a guardar la información.

Procedemos a modificar el archivo de `index.js` de la siguiente manera :

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

const filename = 'data.txt';
const filepath = `${path.resolve('.')}/${filename}`;
const params = [
  { name: 'item', alias: 'i', type: String },
  { name: 'times', alias: 't', type: Number },
  { name: 'frequency', alias: 'f', type: String }
];
const options = commandLineArgs(params);

const line = `You want to ${options.item} ${options.times} time(s) every ${options.frequency}`;
fs.appendFileSync(filepath, line + os.EOL);
```

En el código anterior creamos las constantes:

- `filename` : Que contiene el nombre del archivo .
- `filepath` : Contiene la ruta absoluta donde se guardará el archivo, la función `path.resolve` recibe como argumento el directorio de la ruta que desea resolver, en este caso, es `.` que significa el directorio actual del proyecto .
- `params` : Los parámetros que se recibirán por línea de comandos .
- `options` : Queda almacenado el objeto ya estructurado con todas los parámetros .

Ahora, antes de escribir la línea en el archivo, estamos almacenando en una variable y luego utilizamos la función `appendFileSync` que toma como primer argumento la ruta y nombre del archivo, y, segundo, el contenido que se va a escribir. Nótese dos cosas importantes:

1. ¿Por qué la función termina en Sync? ¿Qué significa esto? Esta versión de la función es sincrónica; es decir, que la aplicación no continúa hasta que no termine la operación. También existe la versión asincrónica, la cual veremos con más detalle en un capítulo más adelante, pero esto no quiere decir que sea buena o mala, solo que es sincrónica.
2. ¿Por qué estamos añadiendo `os.EOL` al final del contenido? Este es el carácter especial para indicar salto de línea dependiendo del sistema operativo.

Si ejecutamos nuestra aplicación una vez más:

```
node index -i "Jogging" -t 3 -f "Week"
```

Se puede observar que la aplicación no imprime nada en la consola, pero si revisamos el archivo `data.txt`, allí veremos nuestros datos guardados. Simplemente podemos volver a ejecutar la aplicación para seguir añadiendo datos con argumentos diferentes.

Para terminar la aplicación sería interesante imprimir todo el listado de contenido del archivo para lo cual añadimos el siguiente código al final de nuestro archivo `index.js`:

```
...  
  
const content = fs.readFileSync(filepath, 'utf-8');  
console.log(content);
```

Los tres puntos (`...`) son una convención para indicar que hay más contenido antes en el archivo, no se deben añadir.

Así como utilizamos la versión sincrónica del `fs` `appendFileSync` utilizaremos la misma pero esta vez para leer `readFileSync` y guardamos el resultado en la variable `content`. El argumento para la codificación de caracteres `utf-8` no es opcional, ya que si no lo especificamos lo almacenado en `content`, será el buffer del archivo binario del sistema, lo cual no es lo que necesitamos. Finalmente imprimimos el resultado en la consola.

Código fuente de la aplicación

Más información:

- [fs](#)
- [os](#)
- [path](#)

Archivos JSON

En la sección anterior vimos cómo, mediante la librería `fs`, podemos tener acceso a leer y escribir archivos en el sistema, pero la información que guardamos fueron solo líneas. Es un archivo, no fueron datos estructurados que luego podamos recuperar y manipular fácilmente. Existe un formato de archivos estructurados que es muy popular en todos los lenguajes y especialmente en JavaScript, estamos hablando de JSON (*JavaScript Object Notation*). Un ejemplo de este, es el siguiente código:

```
{
  "item": "Jogging",
  "times": 3
  "frequency": "Week"
}
```

O si es un arreglo de datos:

```
[{
  "item": "Jogging",
  "times": 3
  "frequency": "Week"
},{
  "item": "Read",
  "times": 1
  "frequency": "Day"
},{
  "item": "Vacations",
  "times": 2
  "frequency": "Year"
}]
```

Si analizamos los ejemplos anteriores, la estructura de los archivos JSON son iguales a los objetos literales en JavaScript. Por lo tanto, es muy sencillo leer el contenido del archivo y almacenarlo en un objeto literal de JavaScript, y a la hora de guardarlo se haría con la misma estructura. Inclusive Node.js puede "requerir" directamente los archivos JSON como módulos sin necesidad de la librería `fs`, lo que los hacen también ideales para usarlos como archivos de configuración.

Más información:

- [JSON](#)
- [JSON JavaScript](#)

Creamos e iniciamos un nuevo proyecto llamado `bucket-list-2.0` que colocamos en el siguiente contenido en el archivo de `index.js` :

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

const filename = 'data.json';
const filepath = `${path.resolve('.')}/${filename}`;
const params = [
  { name: 'item', alias: 'i', type: String },
  { name: 'times', alias: 't', type: Number },
  { name: 'frequency', alias: 'f', type: String }
];

const options = commandLineArgs(params);
let items = [];

try{
  //if file exist
  const content = fs.readFileSync(filepath, 'utf-8');
  items = JSON.parse(content);
}catch(e){
  //if file does not exist
  fs.openSync(filepath, 'w');
}

items.push({
  item: options.item,
  times: options.times,
  frequency: options.frequency
});

fs.writeFileSync(filepath, JSON.stringify(items));
console.log(items);
```

Antes de ejecutar, analicemos lo que cambió en el código, fragmento por fragmento:

```
var items = [];

try{
  //if file exist
  const content = fs.readFileSync(filepath, 'utf-8');
  items = JSON.parse(content);
}catch(e){
  //if file does not exist
  fs.openSync(filepath, 'w');
}
```

Para leer el archivo colocamos nuestro código dentro de un bloque `try / catch`, pues la operación de `readFileSync` puede fallar si el archivo no existe. Nuevamente cabe recordar que más adelante, en el siguiente capítulo, veremos las versiones asíncronas de estas funciones que tienen una estructura y manejo de errores diferente. Realizamos las siguientes operaciones:

1. Leemos el contenido del archivo y lo almacenamos en la variable `content`.
2. Utilizamos la función `JSON.parse` para convertir ese contenido del archivo en un objeto literal de JavaScript y almacenarlo en la variable `items`.
3. Si la operación `readFileSync` falla, ya que el archivo no existe, la aplicación entra en el bloque de `catch`, por lo cual procedemos a crear un archivo vacío en modo escritura (`w`) con la función `openSync`.

```
items.push({
  item: options.item,
  times: options.times,
  frequency: options.frequency
});
```

Ya que la variable `items` es un *Array*, ya sea porque el archivo esté vacío o se leyó el contenido del archivo, podemos utilizar la función `push` para añadir un objeto literal a final de este, que obtuvimos de los parámetros enviados por línea de comandos en la ejecución de nuestra aplicación. Se indican los pares de llave y valor para cada uno de los campos capturados desde la línea de comandos en la variable `options`.

```
fs.writeFileSync(filepath, JSON.stringify(items));
console.log(items);
```

Utilizamos la función `JSON.stringify` para hacer el proceso inverso, esta vez convertir el contenido de la variable `items` en una cadena y guardar su contenido en el archivo con la función `writeFileSync`, que a diferencia de `appendFileSync` (que añade contenido al final del archivo) esta sobrescribe todo el contenido del archivo. Finalmente imprimimos por pantalla cuáles son los items que tenemos en nuestro *bucket list*.

Ejecute la aplicación un par de veces para ver el resultado:

```
node index -i "Jogging" -t 3 -f "Week"
```

```
[ { item: 'Jogging', times: 3, frequency: 'Week' } ]
```

```
node index -i "Vacation" -t 2 -f "Year"
```

```
[ { item: 'Jogging', times: 3, frequency: 'Week' },  
  { item: 'Vacation', times: 2, frequency: 'Year' } ]
```

Como se puede ver, el programa guarda e imprime por la consola la información estructurada que tenemos almacenada en el archivo `data.json`, pero la información se está almacenando toda en una sola línea, lo cual no es legible en el archivo. Por otra parte, si se compara esta versión de la aplicación con la anterior no agrega mucha funcionalidad, pues de nada nos sirve tener los datos estructurados si no tenemos la ventaja de manipularlos.

Realicemos esos dos cambios a continuación:

Modificamos la línea que convierte el JSON en cadena y le añadimos dos argumentos más:

```
JSON.stringify(items, null, 2);
```

El segundo argumento es una función para transformar el JSON si lo necesitamos; como esta no es la ocasión, le pasamos `null` a la función y el tercer argumento es la cantidad de espacios para la identificación del archivo JSON en este caso el número 2.

```
items.forEach(function (el){  
  console.log(`You want to ${el.item} ${el.times} time(s) every ${el.frequency}`);  
});
```

Ahora, en vez de escribir solamente el contenido de la variable `items` en la consola, recorreremos el Array y, por cada elemento lo imprimimos en la consola, pero esta vez podemos elegir cómo imprimirlo, ya que en cada iteración del ciclo tenemos disponible la variable `el` que contiene los datos de ese elemento del *bucket list*. Finalmente, si revisamos el contenido del archivo `data.json` vemos que ahora se almacena en un formato más legible para los usuarios.

[Código fuente de la aplicación](#)

Interactuar con la consola

Hasta el momento hemos capturado los argumentos desde la línea de comandos, pero existen otras formas de que la aplicación interactúe con el usuario en la consola. Para ello se utilizará una librería llamada InquirerJS que nos permite una gran variedad de opciones, tales como: preguntas desde la consola para que el usuario escriba la respuesta y presione la tecla `ENTER` para continuar, mostrar una pregunta con un listado como opciones de respuesta para que el usuario seleccione, con las teclas de desplazamiento, la opción deseada y, finalmente, la tecla `ENTER` para continuar y muchísimas más. Así que creemos un nuevo proyecto llamado `bucket-list-3.0` :

Instalamos la librería como dependencia en nuestro proyecto:

```
npm i -S inquirer
```

Agregamos el siguiente código a nuestro archivo `index.js` :

```
const inquirer = require('inquirer');

const options = [{
  type: 'input',
  name: 'item',
  message: "What do you want to add?"
},{
  type: 'input',
  name: 'times',
  message: "How many times do you want to do it?"
},{
  type: 'list',
  name: 'frequency',
  message: "Which will be the frequency?",
  choices: ['Day', 'Week', 'Month', 'Year']
}];

inquirer.prompt(options).then(answers => {
  console.log(answers);
});
```

`inquirer` funciona parecido a la librería `command-line-args` ya que definimos un Array de opciones que esta librería recibe mediante el método `prompt` . Pero a diferencia de la otra librería, `inquirer` utiliza un llamado asíncronico para cuando se terminen de contestar todas las preguntas,. En el próximo capítulo se tratará más en detalle sobre los llamados asíncronicos, pero por el momento lo importante es que el código que se ejecute, luego de

que el usuario ha contestado las preguntas, esté dentro de las llaves (`{ }`). Las respuestas las almacena como un objeto que contiene llave / valor en la variable `answers` donde la llave es la misma llave `name` especificada en el Array de `options` y, el valor, su respectiva respuesta.

```
npm start
```

Recuerde configurar el *script* de *start* en el `package.json` en la sección de *scripts*

```
? What do you want to add? Travel
? How many times do you want to do it? 1
? Which will be the frequency? Month
```

Las respuestas dadas en el fragmento anterior son una sugerencia y, al final, observamos el contenido de la variable `answers` impreso en la consola.

```
{ item: 'Travel', times: '1', frequency: 'Month' }
```

Ahora mezcle el código que teníamos de la versión anterior de bucket list con este nuevo:

```
const fs = require('fs');
const path = require('path');
const inquirer = require('inquirer');

const filename = 'data.json';
const filepath = `${path.resolve('.')}/${filename}`;
let items = [];

try{
  const content = fs.readFileSync(filepath, 'utf-8');
  items = JSON.parse(content);
}catch(e){
  fs.openSync(filepath, 'w');
}

const options = [{
  type: 'input',
  name: 'item',
  message: "What do you want to add?"
},{
  type: 'input',
  name: 'times',
  message: "How many times do you want to it?"
},{
  type: 'list',
  name: 'frequency',
  message: "Which will be the frequency?",
  choices: ['Day', 'Week', 'Month', 'Year']
}];

inquirer.prompt(options).then(answers => {
  items.push({
    item: answers.item,
    times: answers.times,
    frequency: answers.frequency
  });

  fs.writeFileSync(filepath, JSON.stringify(items, null, 2));

  items.forEach(function (el){
    console.log(`You want to ${el.item} ${el.times} time(s) in a ${el.frequency}`);
  });
});
```

Como se puede apreciar, el código que está dentro de las llaves (`{ }`) de `inquirer` se ejecuta sólo cuando el usuario termina de contestar todas las preguntas. Una vez se ejecute nuevamente el proyecto, se puede ver cómo interactuar con la consola de una forma más dinámica.

Esta aplicación es monolítica, lo cual nos indica que todo el código está en un solo archivo, como un bloque, y a medida que se van agregando más funcionalidades va creciendo. Esto se puede organizar aún más utilizando los módulos que vimos en el capítulo anterior y, así, poder extrapolar ciertas funcionalidades como la lectura y escritura del archivo o el formato en que se imprime la lista. Crear un archivo llamado `datastore.js` con el siguiente contenido

:

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

module.exports.load = function (filepath) {
  try{
    const content = fs.readFileSync(filepath, 'utf-8');
    return JSON.parse(content);
  }catch(e){
    fs.openSync(filepath, 'w');
    return [];
  }
}

module.exports.save = function (filepath, content) {
  fs.writeFileSync(filepath, JSON.stringify(content, null, 2));
}
```

Este módulo tendrá la única responsabilidad de la persistencia de los datos, en este caso será el *File System*; por lo tanto, este módulo es el que necesita la librería *core* `fs` y no el módulo principal. Esto es una gran ganancia pues si en el futuro queremos cambiar el método de persistencia como una base datos o una conexión remota, lo realizaremos en este módulo sin afectar nuestro módulo principal. Una pequeña modificación sustancial se ha realizado en la función `load` : se devuelve lo que se lee del archivo como un Array.

Creamos otro archivo llamado `utils.js` para guardar funciones variadas, como imprimir en la consola una lista (Array):

```
module.exports.printList = function(items){
  items.forEach(function (el){
    console.log(`You want to ${el.item} ${el.times} time(s) in a ${el.frequency}`);
  });
}
```

Finalmente, hacemos las modificaciones necesarias en nuestro archivo principal de la aplicación:

```
const inquirer = require('inquirer');
const path = require('path');

const datastore = require('./datastore');
const utils = require('./utils');

const filename = 'data.json';
const filepath = `${path.resolve('.')}/${filename}`;
const items = datastore.load(filepath);
const options = [{
  type: 'input',
  name: 'item',
  message: "What do you want to add?"
}, {
  type: 'input',
  name: 'times',
  message: "How many times do you want to it?"
}, {
  type: 'list',
  name: 'frequency',
  message: "Which will be the frequency?",
  choices: ['Day', 'Week', 'Month', 'Year']
}];

inquirer.prompt(options).then(answers => {
  items.push({
    item: answers.item,
    times: answers.times,
    frequency: answers.frequency
  });

  datastore.save(filepath, items);
  utils.printList(items);
});
```

Código fuente de la aplicación

Más información:

- [inquirer](#)

Depurar aplicaciones

Una parte fundamental en el desarrollo de aplicaciones es la depuración, ya sea para encontrar errores o ver el estado de la aplicación en un momento de la ejecución de este mismo. Node.js ofrece diferentes formas para llevar a cabo este proceso; así como el complemento con otras herramientas como Google Chrome y editores de texto como Visual Studio Code, que facilitan mucho esta tarea.

En las secciones anteriores hemos utilizado la función `console.log` para escribir en la consola el texto que nosotros deseamos, pero también podemos utilizarlo para imprimir variables, objetos y muchas otras estructuras, con el fin de hacer una depuración del programa. Pero se convertiría en un proceso tedioso al tener que reiniciar cada vez la aplicación para agregar, quitar o modificar los llamados de estas funciones y, peor aún, perdiendo el estado de la aplicación. El punto es: puedes utilizar `console.log` para depurar tu aplicación, y de pronto en algunos casos sea justificable, pero existen otras formas y herramientas para hacerlo, lo cual veremos a continuación.

Depurar con Node.js

Una de las diferentes formas de depurar un aplicación en Node.js es con el comando `inspect`, que ejecuta la aplicación línea a línea; permite interactuar con ella en la consola y ofrece diferentes comandos para decidir qué hacer en la siguiente iteración. Tomemos como ejemplo la aplicación de la sección "Interactuar con la consola" (bucket-list-3.0) y ejecutemos el siguiente comando:

```
node inspect index
```

También es posible utilizar la opción equivalente `--inspect t` de la siguiente forma: `node --inspect index`, pero está es para utilizar solo herramientas externas

Lo primero que podemos notar es el parámetro adicional para ejecutar el comando de inicio de la aplicación; este comando ejecutará y detendrá su ejecución en la primera línea de la aplicación para que el usuario decida qué hacer con ella:

```
< Debugger listening on ws://127.0.0.1:9229/4e2b0d48-5643-455d-af74-1d5d7007740c
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in index.js:1
> 1 (function (exports, require, module, __filename, __dirname) { var inquirer = requi
re('inquirer');
  2
  3 var datastore = require('./datastore');
debug>
```

En la salida de la pantalla podemos observar diferentes cosas:

1. El *debugger listening* que nos permitirá depurar la aplicación con herramientas de terceros, lo cual veremos en la siguiente sección.
2. `Break on start in index.js:1`, nos indica en cuál archivo y línea de la aplicación estamos.
3. El cursor se encuentra en el número de línea 1 de la aplicación; vale recordar que los módulos (como vimos en su respectiva sección) son realmente una IIFE (*immediate invoke function expression*) con los parámetros que necesita para ejecutarse. Así cada uno de ellos se ejecuta en el *scope* de la función sin compartir variables con el resto de la aplicación.
4. Por último, Node.js espera la instrucción para continuar con el proceso de depuración (`debug>`).

Las diferentes opciones son:

- c: (continue) Continuar con la depuración hasta el próximo breakpoint. Veremos como introducir manualmente estos más adelante.
- n: (next) Ejecutar y seguir a la siguiente línea. Este es el más utilizado.
- s: (step) Ingresar en la siguiente función.
- o: (out) Salir de la función actual.
- repl: Permite evaluar código. Este comando lo habíamos visto en una sección anterior y se puede utilizar, por ejemplo, para ver el valor de una variable.

Introducimos el comando next, que es la letra n, seguido de la tecla ENTER hasta obtener el siguiente resultado en la consola:

```
break in index.js:7
  5
  6 var filePath = 'data.json';
> 7 var items = datastore.load(filePath);
  8
  9 var options = [{
```

En este momento la línea número 7 de nuestra aplicación no se ha ejecutado, pero queremos ver el contenido de la variable `filePath`. Para ello introducimos el comando `repl` seguido de la tecla ENTER, entramos en el modo de REPL. El sistema nos indica que para salir de ese modo es necesario presionar las teclas CTRL+C. Procedemos a escribir el nombre de la variable (comando o función que deseemos ejecutar), como por ejemplo `filePath`, y presionamos la tecla ENTER. Inmediatamente podemos ver el resultado.

```
debug> repl
Press Ctrl + C to leave debug repl
> filePath
'data.json'
```

Finalmente, para salir del modo de REPL y luego del modo de depuración, presionamos dos veces la combinación de teclas: CTRL+C.

No es muy común depurar una aplicación línea por línea, pero podemos introducir un *breakpoint* manualmente en el lugar que queremos empezar con la depuración. Editamos temporalmente (no guardar este cambio, solo es con un propósito de demostración) el archivo `index.js` e introducimos la siguiente línea:

```
...

});

debugger;

inquirer.prompt(options).then(answers => {

...

```

El comando `debugger` solo tendrá efecto si la aplicación está en modo de depuración de lo contrario el sistema la ignorará. Ejecutamos nuevamente nuestra aplicación en modo de depuración:

```
node inspect index
```

Esta vez presionamos el comando `continue` que es la letra `c`, seguido de la tecla ENTER, y podremos observar cómo nuestra aplicación "salta" hasta nuestro *breakpoint*, permitiendo empezar a depurar la aplicación desde ese punto:

```
break in index.js:24
  22  }];
  23
>24  debugger;
  25
  26  inquirer.prompt(options).then(answers => {
```

De hecho se pueden colocar todos los *breakpoints* que se deseen.

Esta es la forma más básica de depurar una aplicación en Node.js, pero como mencionamos anteriormente se pueden utilizar herramientas de terceros para depurar de una manera más dinámica y visual

.

Mas información:

- [Node JS - Debugging Guide](#)

Depurar con *Chrome Dev Tools*

En la sección anterior vimos cómo depurar de manera básica una aplicación de Node.js, pero comentamos que este proceso permite que herramientas externas efectúen este cometido; esa es la función del *Debugger listener* que vimos en la sección anterior:

```
< Debugger listening on ws://127.0.0.1:9229/4e2b0d48-5643-455d-af74-1d5d7007740c
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
```

Esta vez se iniciará el proceso de depuración con la siguiente opción:

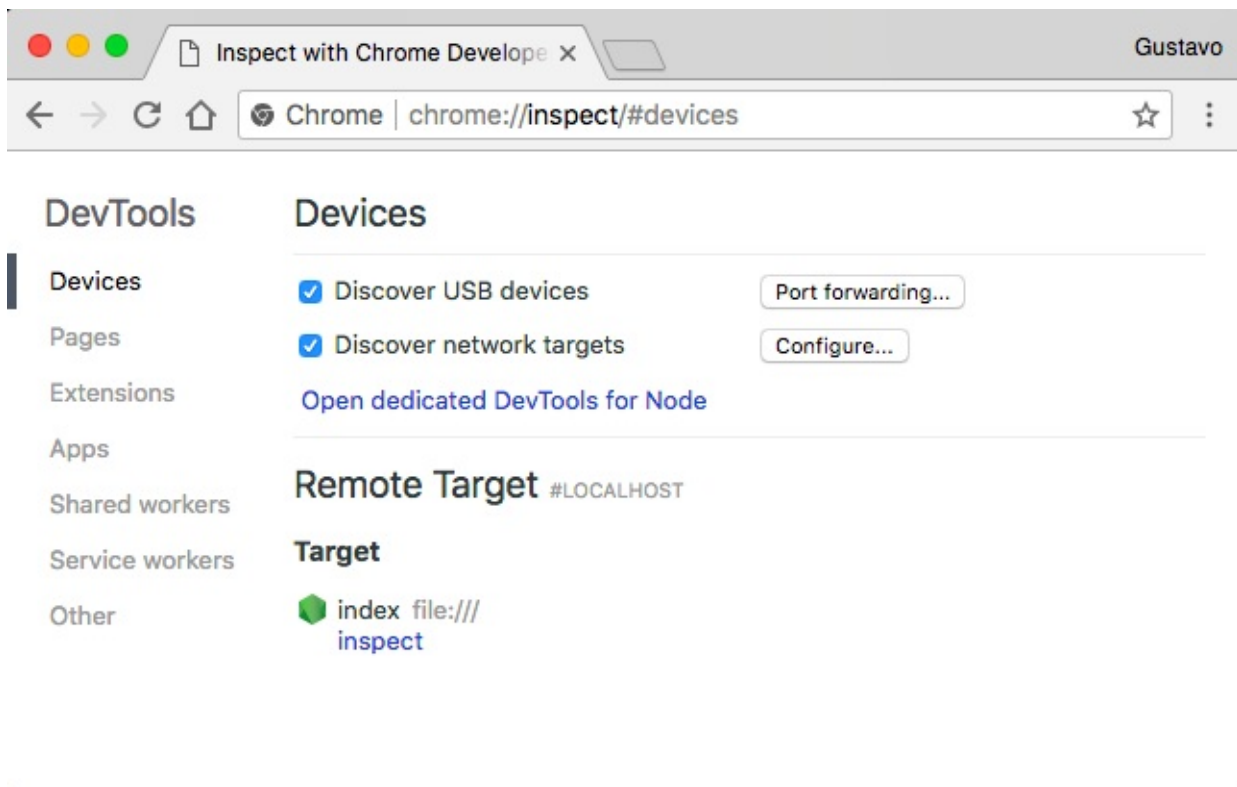
```
node --inspect index
```

Debemos tener en cuenta que no es la misma opción de la sección anterior (`inspect`); cumplen el mismo propósito pero esta vez la opción con los dos guiones (`--inspect`) ejecuta la aplicación y solo arranca en un proceso adicional el *Debugger Listener*, mas no habilita la opción para depurar por consola.

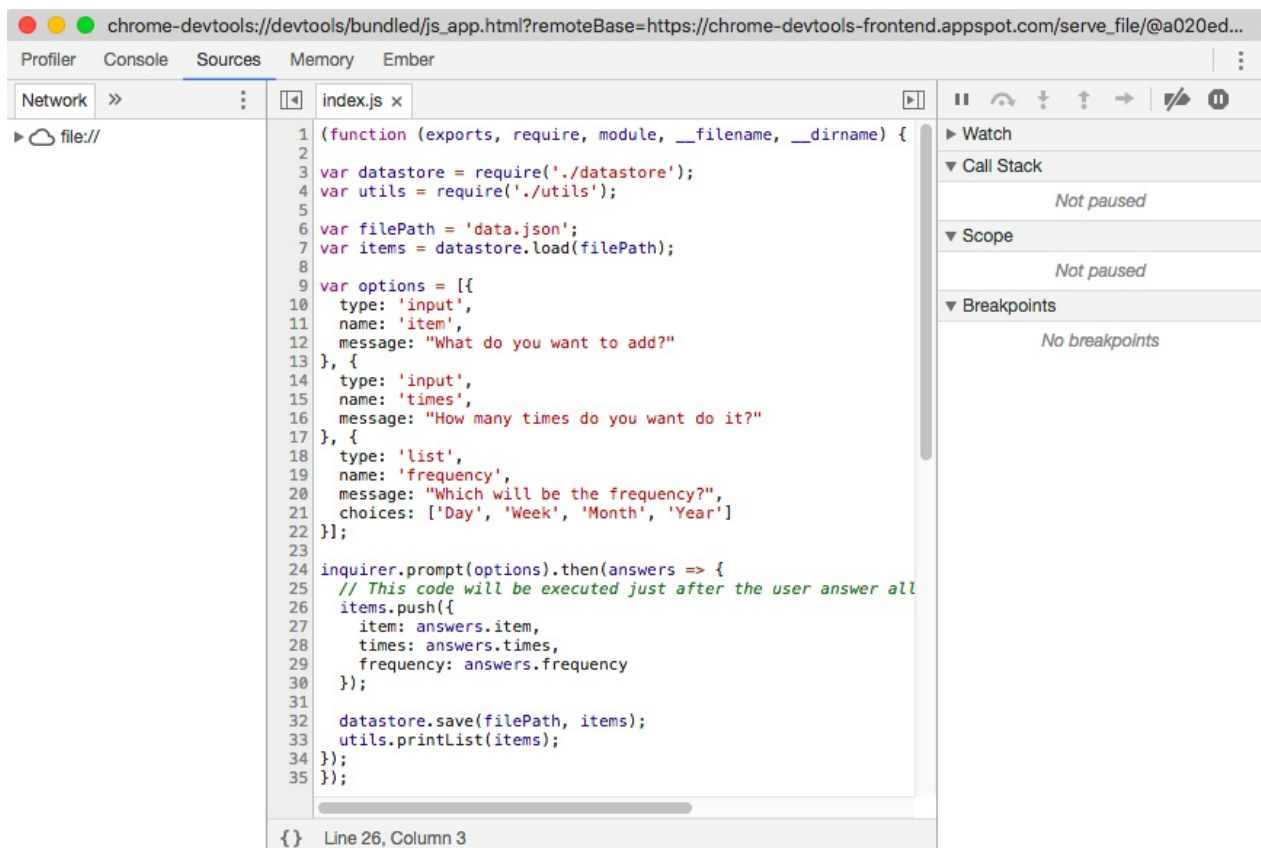
El Chrome Dev Tools utiliza el Inspector protocol para poder depurar la aplicación (o aplicaciones) de Node.js que se está ejecutando en este momento. Para ello, abrimos Google Chrome y escribimos lo siguiente en la barra de dirección:

```
chrome://inspect/#devices
```

 , como muestra la siguiente captura de pantalla:



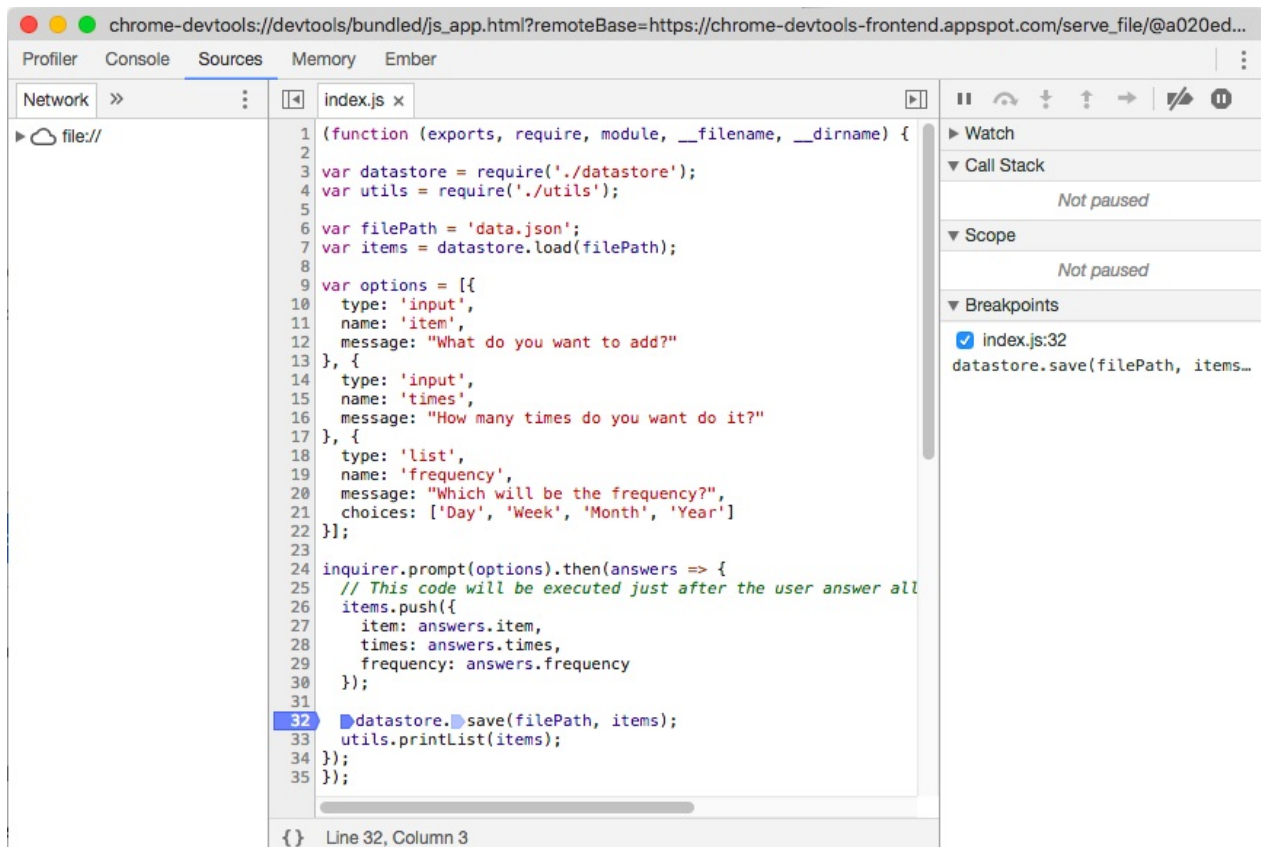
En la sección Remote Target hacemos clic en el enlace "inspect" para abrir la ventana de depuración:



Como podemos observar en la captura de pantalla anterior, el Chrome Developer Tools nos ofrece un ambiente un poco más amigable para hacer el proceso de depuración de la aplicación seleccionada. Aquí podremos hacer muchas operaciones, como ver los

diferentes archivos de nuestra aplicación, colocar todos los breakpoints en las líneas de los archivos, ver el valor de las variables en un momento determinado de la ejecución de la aplicación y muchas otras operaciones.

A continuación establezcamos un *breakpoint* en la línea 32 de nuestro archivo `index.js` haciendo clic en el número de línea; así podremos detener la ejecución de la aplicación en ese punto y ver el estado de las variables. Si lo hicimos de manera correcta debe verse como muestra la siguiente captura de pantalla:



Si regresamos a la consola, nuestra aplicación de Bucket List debe estar esperando que introduzcamos la información del próximo ítem:

```

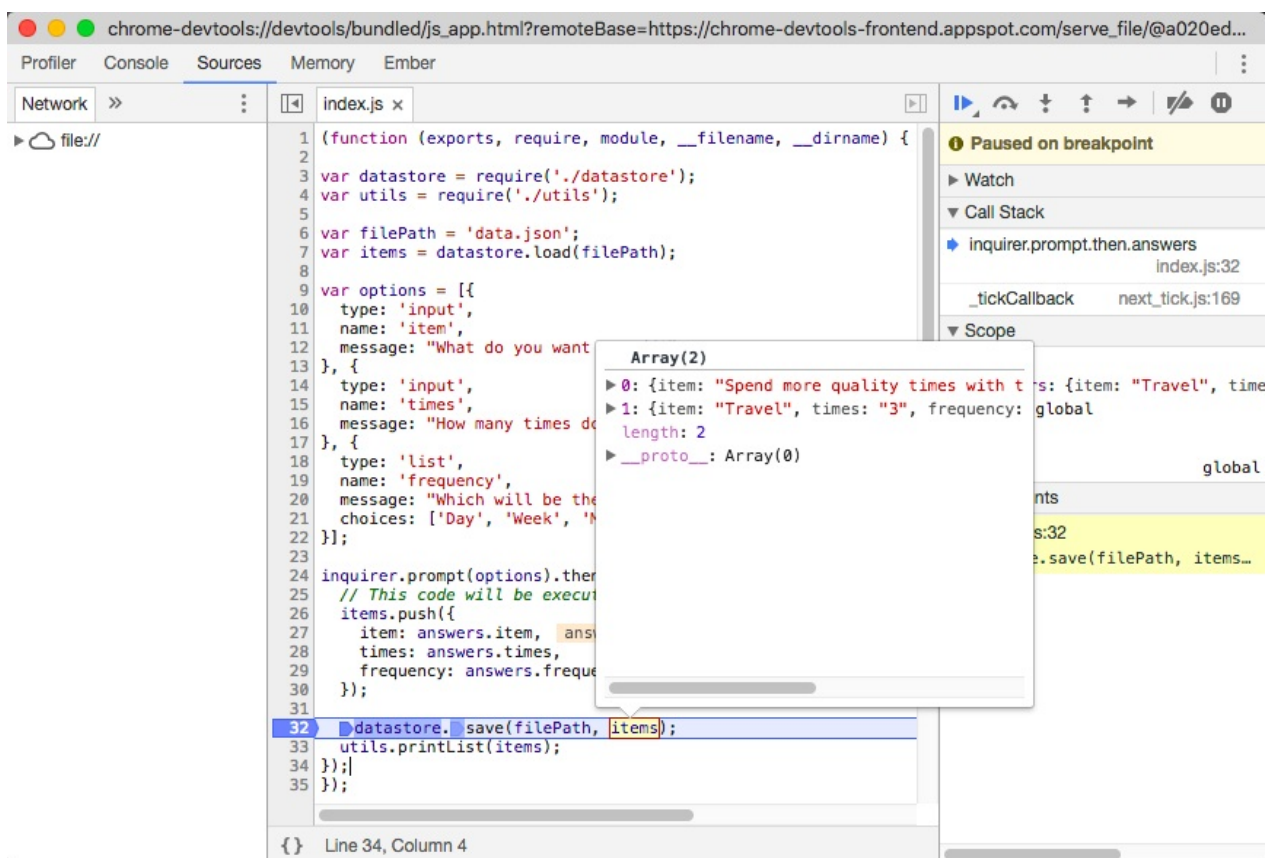
node --inspect index
Debugger listening on ws://127.0.0.1:9229/9d708bbc-5ed8-4c37-a36c-fa0b8f2500fd
For help see https://nodejs.org/en/docs/inspector
? What do you want to add? Debugger attached.

```

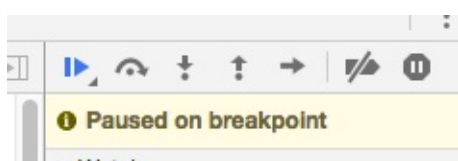
Ingresamos la información para un ítem:

```
node --inspect index
Debugger listening on ws://127.0.0.1:9229/9d708bbc-5ed8-4c37-a36c-fa0b8f2500fd
For help see https://nodejs.org/en/docs/inspector
? What do you want to add? Debugger attached.
? What do you want to add? Travel
? How many times do you want do it? 3
? Which will be the frequency? Year
```

Cuando terminamos de introducir la información la ventana de depuración ha detectado que llegamos al *breakpoint*. Allí cambia el foco y nos permite interactuar con la aplicación, por ejemplo podemos colocar el cursor sobre la variable `items` para ver los elementos que se van a guardar en el `datastore` de nuestra aplicación, así como muestra la siguiente captura de pantalla:



Si deseamos continuar con la ejecución de nuestra aplicación hacemos clic en el icono de Continuar (azul) de la barra de comandos de la ventana de depuración:



De hecho cada una de las primeras cuatro opciones tienen su equivalente en las que nos ofrecía la opción básica de depuración Node.js:

- Continue: c
- Next: n
- Step In: s
- Step Out: o

Una vez la aplicación vuelve a ejecución, podemos cancelar el proceso de ejecución presionando CTRL+C en la consola donde se inició el proceso de la aplicación.

Como vimos, este es un proceso un poco más versátil en el sentido que tenemos una interfaz gráfica que nos permite interactuar con la aplicación. Cabe anotar que inclusive todavía los *breakpoints* manuales (`debugger;`) son válidos.

Algo muy importante en la aplicación que estamos depurando, es que casualmente la misma aplicación detenía su ejecución esperando que el usuario ingresará los datos del elemento que desea agregar al *bucket list*, lo cual nos dio tiempo para colocar todos los breakpoints que queríamos. Pero esto no siempre es así. ¿Cómo detener la aplicación antes de que se ejecute la primera línea para que dé tiempo de colocar los breakpoints? Una solución podría ser colocar debugger en la primera línea de la aplicación del archivo de entrada de ejecución de la aplicación, pero sería un poco tedioso. Para esto Node.js tiene la siguiente opción:

```
node --inspect-brk index
```

Hace exactamente lo mismo que la opción `--inspect` , pero esta vez detiene la ejecución de la aplicación en la primera línea del archivo de entrada de la aplicación. Permite interactuar con la ventana de depuración y, por ejemplo, colocar todos los breakpoints antes de que comience la ejecución de la aplicación. Una vez terminado el proceso podemos hacer clic en el botón `Continue` de la barra de comandos.

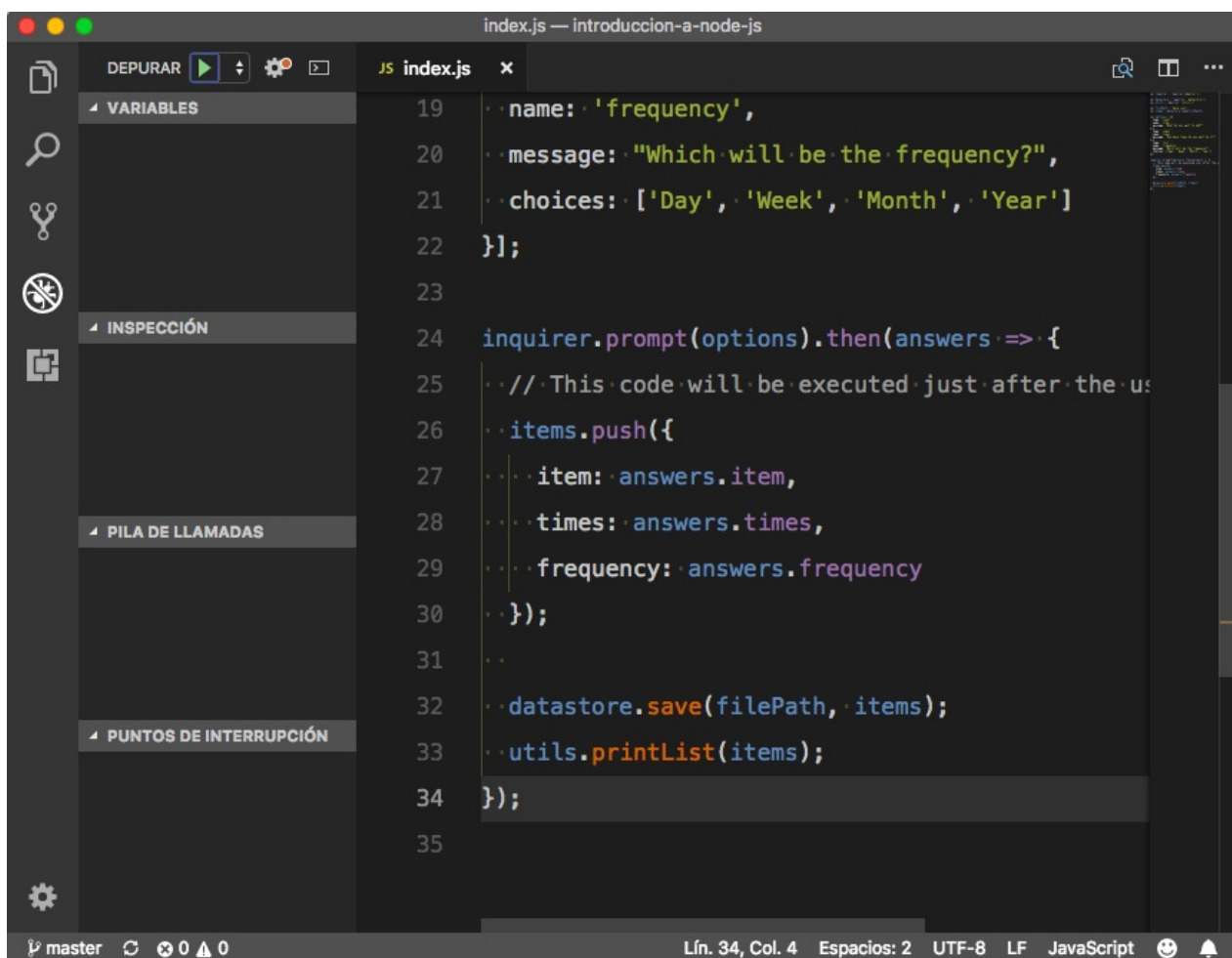
Mas información:

- [Inspector protocol](#)

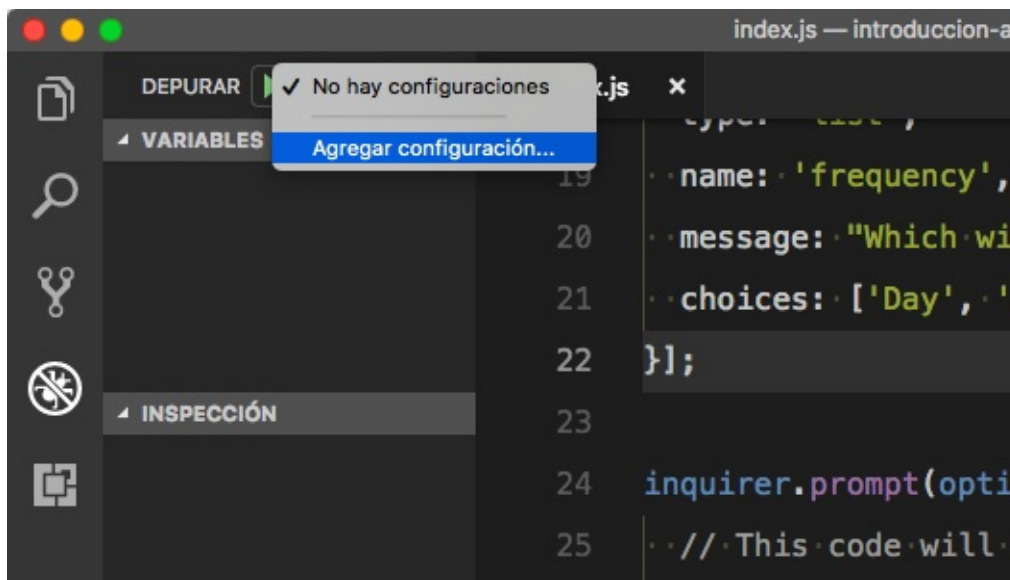
Depurar con VS Code

Una de las ventajas de los editores visuales es la interfaz gráfica, que nos permite escribir el código, y con la adición de plugins añaden mucho más funcionalidades. Visual Studio Code permite, entre otros, depurar aplicaciones de diferentes tipos sin necesidad de ningún plugin. Entre estas aplicaciones en Node.js, permitirán muchas de las opciones que nos brinda el Chrome Developer Tools, pero dentro del mismo editor en el cual estamos programando la misma aplicación.

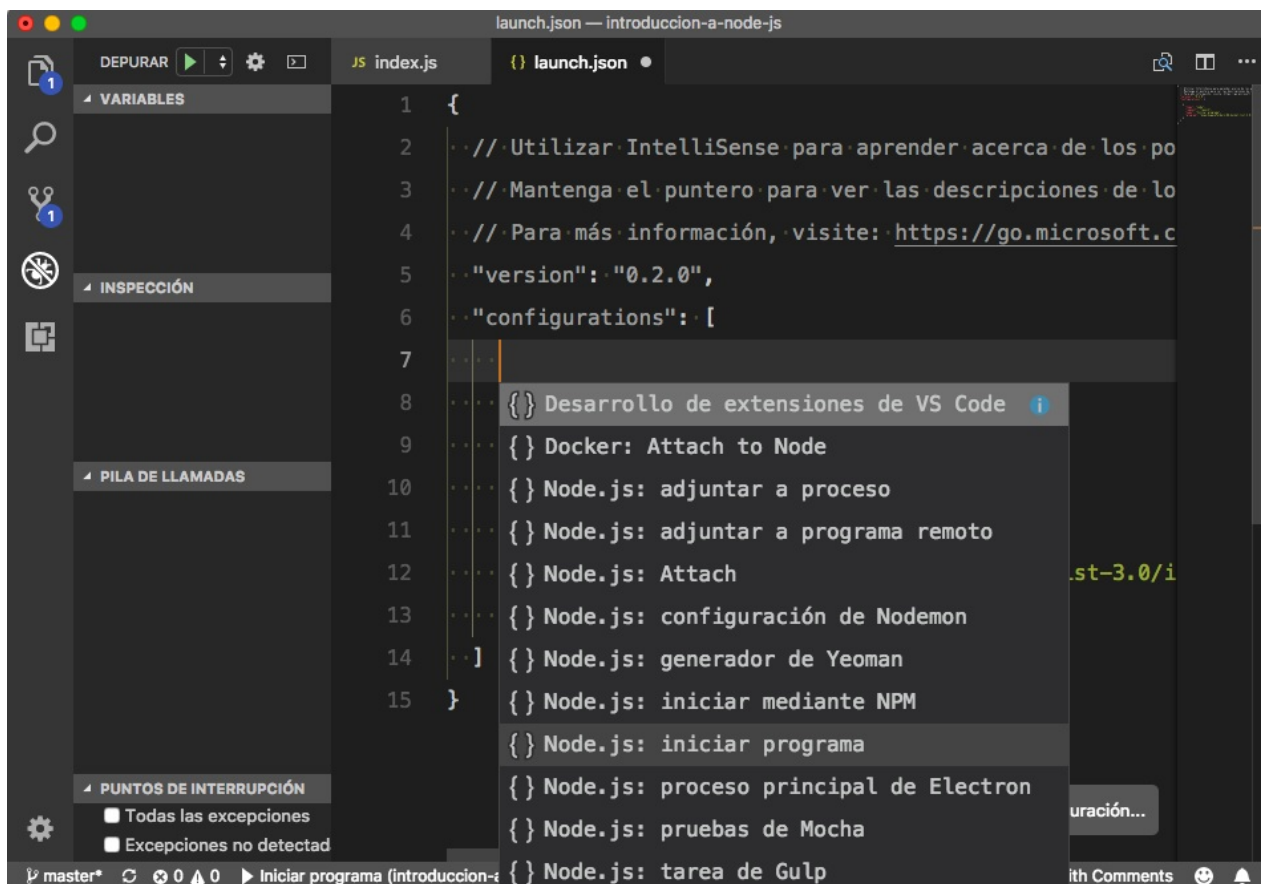
Una vez abrimos el proyecto de la aplicación hacemos clic en el icono de barra de accesos directos "Depurar" como muestra la siguiente captura de pantalla:



A continuación hacemos clic en la lista de selección que está al lado del botón de Inicio de depuración y seleccionamos la opción añadir configuración:



Seleccionamos la opción "Iniciar programa" de la lista de plantillas predeterminadas:



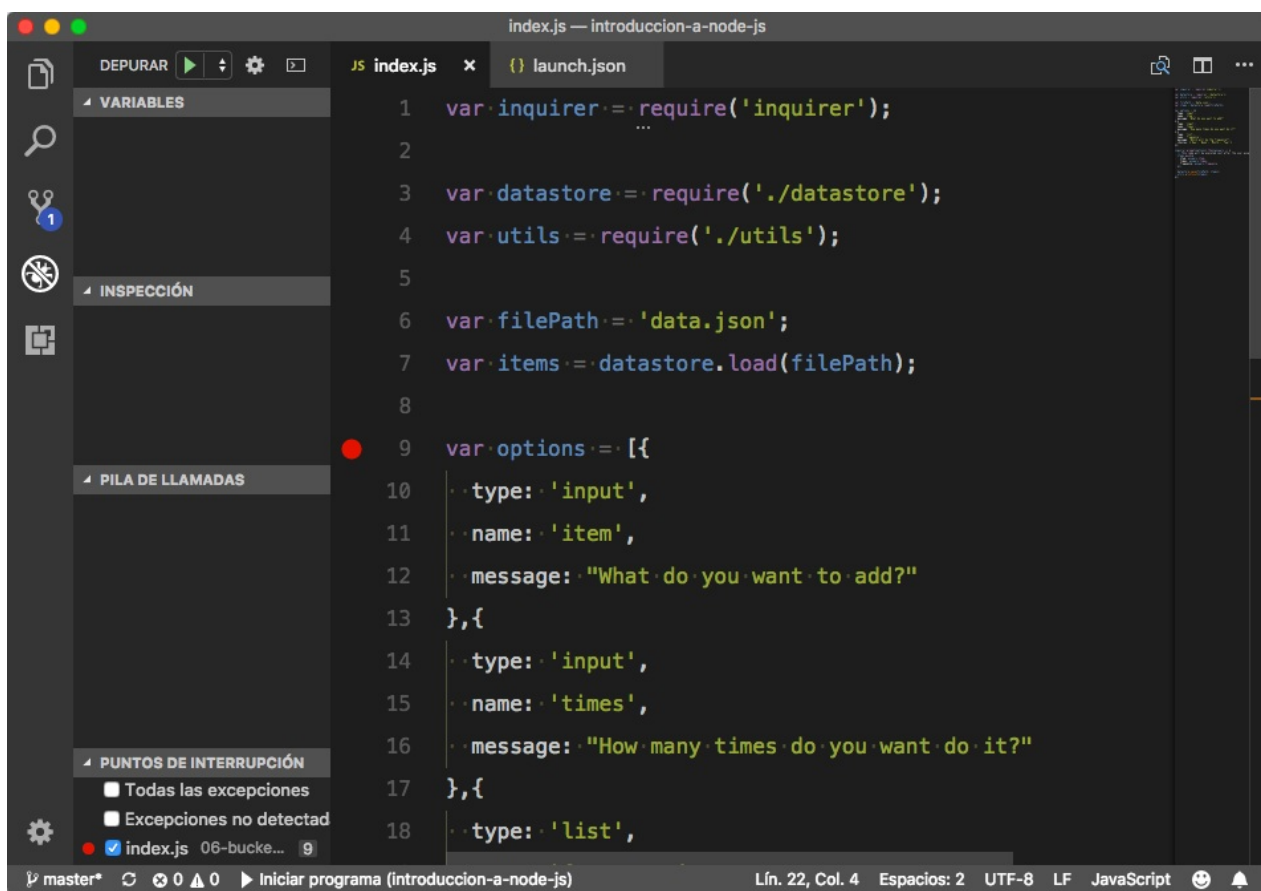
Toda esta configuración será guardada en un archivo llamado `launch.json` que VS Code creará en la raíz del proyecto. Cambiamos el nombre de archivo de entrada para que quede de la siguiente manera:

```

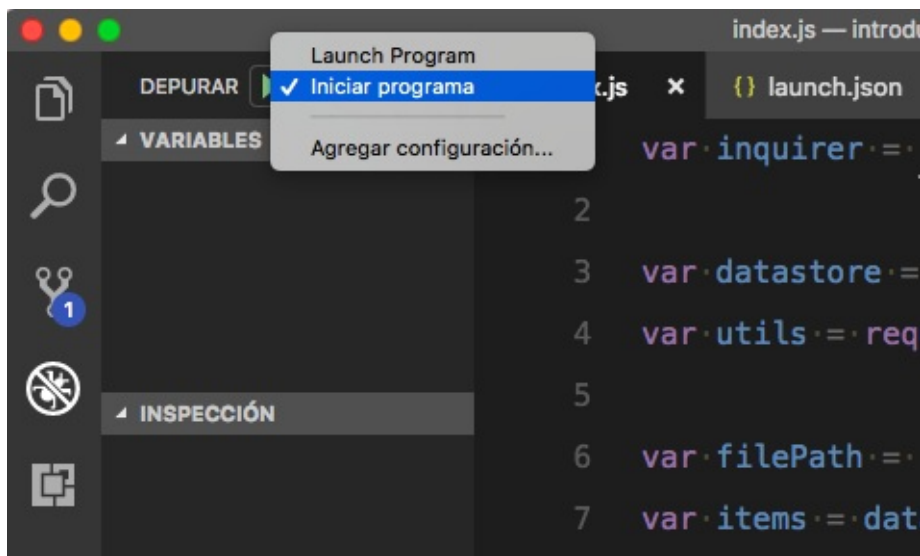
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Iniciar Programa",
      "program": "${workspaceFolder}/index.js"
    }
  ]
}

```

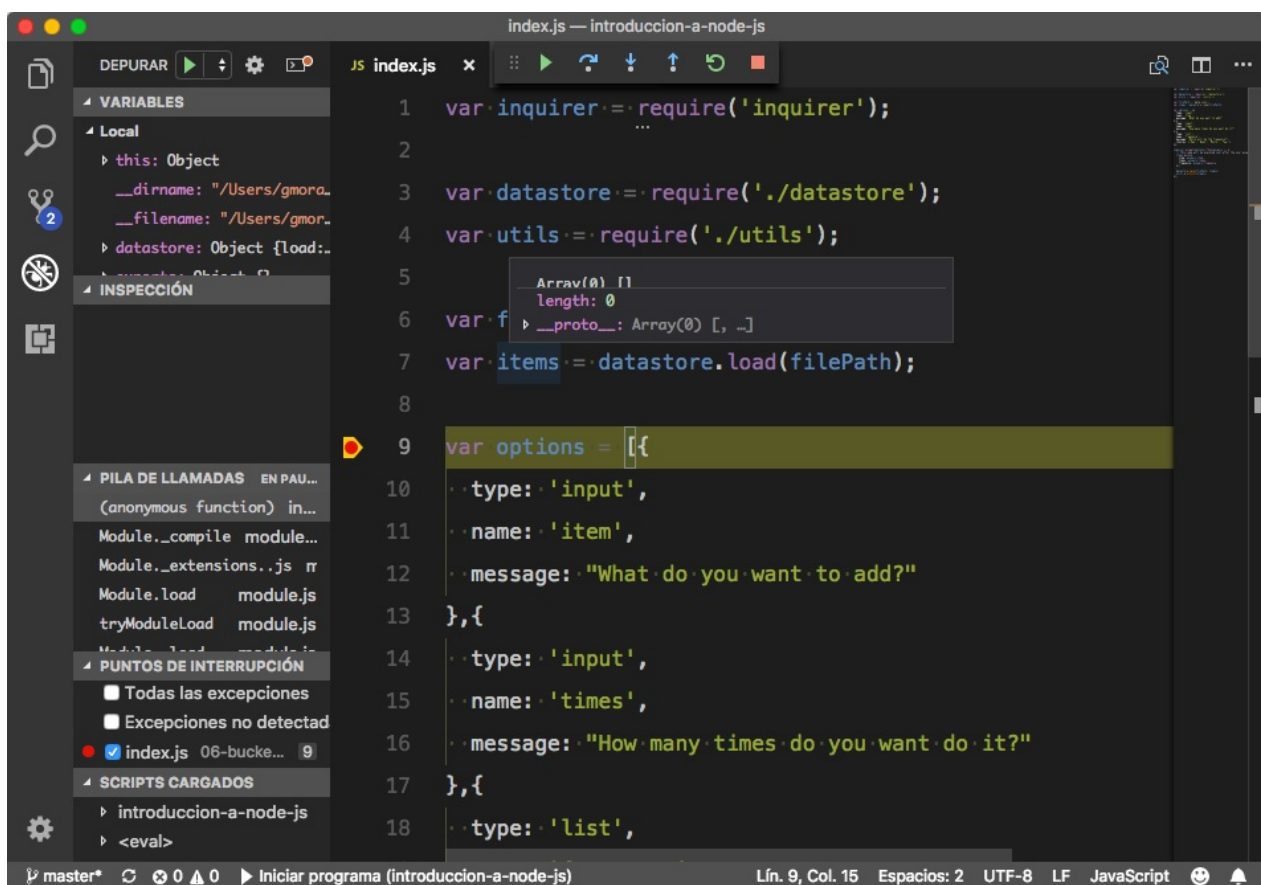
Guardamos el archivo y procedemos a colocar los breakpoints que deseemos haciendo clic en el número de la línea deseada, como muestra el ejemplo de la siguiente captura de pantalla:



Finalmente hacemos clic en el botón de Depuración y seleccionamos la configuración que nos interesa. Es posible tener muchas configuraciones en el mismo archivo de configuración (launch.json) dependiendo del tipo de depuración que se desee hacer para la aplicación:



Visual Studio Code nos permite interactuar con el estado de la aplicación en el punto donde se detuvo la ejecución de la misma, debido a los breakpoints que colocamos.



De igual forma, tiene la barra de comandos que nos da las mismas opciones que tenemos en el Chrome Developer Tools, como Continuar, Next (Step) y demás.

Algo muy importante antes de finalizar es que, al contrario del Chrome Developer Tools, esta nos permitirá seguir interactuando con la consola. Esta opción NO nos permite hacerlo. Para solucionar esto podemos utilizar la segunda opción más utilizada de las plantillas

predeterminadas que es "Adjuntar a Proceso", sencillamente ejecutamos nuestra aplicación desde la consola con:

```
node --inspect-brk index
```

Y en vez de seleccionar la opción "Iniciar Programa" en la lista de opciones, seleccionamos "Adjuntar a Proceso" y seleccionamos la aplicación que se está ejecutando para poder depurarla.

Más información:

- [Node.js Debugging in VS Code](#)

Blocking code y Non-Blocking code

¿Qué sucedería si estamos esperando una respuesta de la lectura de un archivo grande o una operación de red? La aplicación tendría que esperar hasta que la operación terminara para poder continuar, pues V8 no tiene ninguna opción para implementar operaciones asíncronas. Esto dejaría muy mal parado a Node.js como implementación del lado del servidor, pero este tiene una solución para ello, llamado el *Event Loop* (el cual mostraremos más en detalle en la próxima sección). Al igual que su contraparte en el lado del cliente, por ejemplo Google Chrome, también utiliza v8 como "engine" y sabemos que la interacción del navegador es a base de eventos, pues los navegadores también implementan su propio *Event Loop*.

Por consiguiente, es importante hacer una distinción entre el "Blocking code" y "Non-Blocking code", sencillamente este primero se ejecuta línea por línea y tiene que esperar a que termine; en cambio el segundo no. Para ilustrar la explicación veamos los siguientes fragmentos de código:

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

const content = fs.readFileSync('data.json', 'utf-8');
console.log(content);
```

Lo importante del fragmento de código anterior es que la instrucción `console.log(content)` NO se ejecuta hasta que la operación de leer el archivo termina, esto es un clásico ejemplo de "Blocking code".

Por otra parte, veamos el siguiente fragmento de código:

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

fs.readFile('data.json', 'utf8', function(err, content){
  if(err) throw err;
  console.log(content);
});

console.log('more work');
```

Sin analizar a profundidad el código, lo importante es que esta vez la aplicación puede continuar y ejecutar la instrucción `console.log('more work');` independientemente si el archivo ha terminado de leer o no, y una vez este haya terminado ejecutará la instrucción `console.log(content)`. A este tipo de código se llama "Non-Blocking code";

La mayoría de la programación con Node.js es con "Non-Blocking code". Son muy particulares los casos en los que es necesario utilizar "Blocking code", adicionalmente no se recomienda combinar estos dos tipos de código.

Más información:

- ["Blocking" vs "Non-Blocking"](#)

El *Event loop*

Antes de hablar del *Event loop* debemos hablar del *Call Stack*. Así, como su nombre lo indica, es una cola de llamados que tiene cada aplicación. Recordemos que la cola es una estructura de datos que se visualiza en forma vertical. Al iniciar cada aplicación se coloca en esta cola una primera función llamada `main`, que contiene el código principal de la aplicación. Para demostrar esto, tomemos como ejemplo la siguiente aplicación:

```
const fs = require('fs');
const content = fs.readFileSync('data.json', 'utf-8');
console.log(content);
```

Miremos qué sucede a medida que se ejecuta la aplicación:

1. Se agrega la función `main()` al tope del *Call Stack*:
 - i. ``main()``
2. Se agrega la función `require()` al tope del *Call Stack*:
 - i. ``require()``
 - ii. ``main()``
3. Termina de ejecutarse la función `require()` y se quita del tope del *Call Stack*:
 - i. ``main()``
4. Se agrega la función `fs.readFileSync()` al tope del *Call Stack*:
 - i. ``fs.readFileSync()``
 - ii. ``main()``
5. Termina de ejecutarse la función `fs.readFileSync()` y se quita del tope del *Call Stack*:
 - i. ``main()``
6. Se agrega la función `console.log()` al tope del *Call Stack*:
 - i. ``console.log()``
 - ii. ``main()``
7. Termina de ejecutarse la función `console.log()` y se quita del tope del *Call Stack*:
 - i. ``main()``
8. No hay más líneas que ejecutar, termina de ejecutarse la función `main()` y se quita del tope del *Call Stack*.
9. Termina la ejecución de la aplicación

La librería V8 de JavaScript, que es la encargada de interpretar y ejecutar todo el código de JavaScript en Node.js, es *single thread*; y de esta manera al usar el *Call Stack* es que ejecuta todas las aplicaciones. Pero, ¿qué hay de los llamados asíncronos? ¿Cómo se ejecutan estos en el *Call Stack*?

La respuesta a las preguntas anteriores es: el *Event loop*. Esta implementación es lo que permite a Node.js realizar, entre otras cosas, las operaciones asíncronas (*Non-blocking*). A pesar de que este se ejecute como un *single thread*, lo hace delegando operaciones al kernel del sistema.

Para ser más descriptivos, el *Event Loop* es una cola de eventos suscritos que Node.js está revisando cada vez el *Call Stack* está vacío para saber cuál de ellos ha terminado y, si es el caso, invocar el código asociado a este. Revisemos nuevamente el código de ejemplo de la sección anterior con una pequeña modificación:

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

function fileReader(err, content){
  if(err) throw err;
  console.log(content);
}

fs.readFile('data.json', 'utf8', fileReader);

console.log('more work');
```

Para mejorar la legibilidad del ejemplo hemos creado la función `fileReader` (el nombre no es tan importante).

Repasemos qué sucede:

1. Se agrega la función `main()` al tope del *Call Stack*.
2. Se agrega la función `require()` al tope del *Call Stack*.
3. Termina de ejecutarse la función `require()` y se quita del tope del *Call Stack*.
4. La función `fileReader()` es un *function statement*, así esta solamente se registra en la memoria.
5. Se agrega la función `readFile()` al tope del *Call Stack*.
6. Se ejecuta la función `readFile()` pero esta registra un "evento" en el *Event loop* con su respectivo callback (`fileReader`). Una vez que finalice la lectura del archivo (no es posible determinar cuándo terminará pues es un operación asíncronica) se marcará como finalizado el evento.
7. Termina de ejecutarse la función `readFile()` y se quita del tope del *Call Stack*.
8. Se agrega la función `console.log()` al tope del *Call Stack*.
9. Termina de ejecutarse la función `console.log()` y se quita del tope del *Call Stack*.
10. El *Call Stack* está vacío, así que el *Event loop* verifica, en su cola, cuáles eventos han terminado, digamos que aún no se ha terminado de leer el archivo de la función `readFile()` .
11. El *Call Stack* está vacío, así que el *Event loop* verifica en su cola cuáles eventos han terminado. Digamos que esta vez sí se terminó de leer el archivo de la función

`readFile()` , entonces el evento ha finalizado y el *Event loop* saca de la cola este evento. Por lo tanto el *Event loop* coloca la función *callback* asociada a este evento en el tope del *Call Stack*, en este caso la función `fileReader()` .

12. Termina de ejecutarse la función `fileReader()` y se quita del tope del *Call Stack*.
13. No hay más eventos en el *Event Loop*. No hay más funciones en el *Call Stack*, entonces termina la ejecución de la función `main()` y, por lo tanto, la ejecución de la aplicación .

Para conocer en detalle qué verifica el *Event Loop* para saber si termina o no su ejecución, se puede consultar el siguiente enlace:

- [Node JS - Event loop - Phase overview](#)

En el *Event Loop* no solamente se pueden registrar eventos, también otro tipo de operaciones. Vea el siguiente código como ejemplo:

```
setInterval( function(){  
    console.log("Hello");  
}, 1000);  
  
console.log("World");
```

El resultado de la aplicación sera:

```
World  
Hello
```

Ya que la función `setInterval()` registra en el *Event Loop* un *callback* después de *n* milisegundos. En este caso el *callback* es una función anónima, esta función `setInterval()` hace parte del API de Node.js y, por lo tanto, utiliza sus librerías internas para registrar estos eventos asociados. En este caso delega la operación para el procesamiento de esta función especial al kernel del sistema, pero para nuestra aplicación principal sigue siendo *single thread*.

En la documentación de Node.js está descrito cada una de las funciones "especiales", sus respectivos eventos y los *callbacks*. Además, Node.js también permite crear eventos personalizados, como veremos en la próxima sección.

Para concluir, la programación en Node.js está más orientada a eventos que pensar en programar de manera lineal.

Más información:

- [Event loop](#)

- [Timers](#)

El patrón *observer* y *Event Emitter*

Otro patrón fundamental de diseño, presente en Node.js, es el patrón *observer*. Es uno de los pilares de Node.js y es muy utilizado, tanto por el mismo como por los usuarios.

El patrón de *observer* es ideal para modelar flujos reactivos junto con los *callbacks*. Este define un objeto llamado sujeto (*subject*) que una vez realice cierta acción notifica a todos los observadores (*observers*) de lo que ha sucedido.

La clase de *Event Emitter*

El patrón de observer está construido en el *core* de Node.js y disponible a través de la clase `EventEmitter`, la cual permite registrar una o más funciones observadoras cuando un evento en particular suceda. El `EventEmitter` es un prototipo y está disponible en el módulo de Eventos. Está compuesto de los siguientes métodos:

- `on(event, listener)` : Este método permite registrar un *listener* (función) a un evento específico (nombre del evento).
- `once(event, listener)` : Este método es similar al anterior, pero solo se realiza una sola vez y después la asociación es removida.
- `emit(event, [arg1], [...])` : Este método permite disparar o ejecutar un evento, pasando todos los parámetros, que el usuario desee, a las funciones *listener*.
- `removeListener(event, listener)` : Este método remueve la asociación entre un evento y una función *listener*.

Todos estos métodos a su vez retornan una instancia de `EventEmitter` permitiendo así encadenar unos eventos con otros. Esto es una gran diferencia con el uso solo de los *callbacks*; además, en particular el primer argumento, no es un error como lo habíamos visto en los patrones del *callback*, pero puede ser cualquier dato que se pase una vez se emita el evento.

Utilizar la clase *Event Emitter*

La mejor forma de entenderlo es con un ejemplo, inicializamos un proyecto llamado `find-pattern`, creamos un directorio llamado `libs` y dentro un archivo `index.js` con el siguiente contenido:

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');

module.exports.findPattern = function (files, pattern){
  const emitter = new EventEmitter();

  files.forEach((file) => {
    fs.readFile(file, 'utf8', (err, content) => {
      if(err){
        return emitter.emit('file-error', err);
      } else{
        emitter.emit('file-read', file);
      }

      let matches = content.match(pattern);
      if(matches){
        matches.forEach((element) =>{
          emitter.emit('file-match-found', file, element);
        });
      }
    });
  });
  return emitter;
}
```

Podemos ver cómo se incluye el módulo de EventEmitter y se emiten los siguientes eventos:

- `file-read` : El evento ocurre cuando el archivo es leído.
- `file-match-found` : El evento ocurre cuando se encuentra dentro del archivo el patrón enviado.
- `file-error` : El evento ocurre cuando existe algún error leyendo el archivo

A continuación se implementará el código; para ello, cree un archivo llamado `index.js` en la raíz del proyecto, con el siguiente contenido:

```
const { findPattern } = require('./libs');

const pattern = new RegExp(process.argv[2] || '');

findPattern(['logs/log1.txt', 'logs/log2.txt'], pattern)
.on('file-read', function(file){
  console.log(`File: ${file}`);
})
.on('file-match-found', function(file, match){
  console.log(`Match: ${match}`);
})
.on('file-error', function(err){
  console.log(`Error: ${err.message}`);
});
```

En el código anterior, creamos una expresión regular a partir del argumento suministrado por la línea de comandos; luego suscribimos los eventos con sus respectivas funciones *listener*. En el código fuente se encuentra el directorio `logs` y sus respectivos archivos de muestra. Ejecute el código con el siguiente comando:

```
npm start -- "ERROR \w+"
```

En el caso anterior, se buscará cualquier cadena que comience con la palabra `ERROR` y devolverá la cadena con las palabras subsiguientes.

[Código fuente de la aplicación](#)

Propagar los errores

El `EventEmitter`, así como sucede en los *callbacks*, no se pueden simplemente lanzar una excepción, ya que se perdería en el Event Loop si los eventos son emitidos asincrónicamente. En cambio la convención es emitir un evento especial llamado `error` y pasar el objeto `Error` como argumento; justamente esto fue lo que hicimos en el código anterior de `findPattern`.

Siempre es una buena práctica registrar un evento de error, ya que Node.js lo tratará en una forma especial y arroja una excepción cuando el programa termine su ejecución.

Renderizar páginas Web

Habíamos nombrado que Node.js puede tener muchos usos y uno de ellos es el de Servidor Web; es un simple proceso que reciba una petición desde la URL y envíe como respuesta el archivo indicado si este existe.

Node.js posee varios módulos nativos que pueden ayudar a cumplir este propósito. Cree un nuevo proyecto llamado `web-server` y proceda a iniciarlo (tal y como lo hemos hecho en capítulos anteriores).

Editamos el archivo inicial `index.js` y colocamos el siguiente contenido:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Este código fue extraído de la página oficial de Node.js, de la sección [Usage & Example](#). Se analizará su contenido antes de proceder a ejecutarlo:

- El módulo nativo `http` de Node.js contiene todas las funciones asociadas a este protocolo.
- Se crean las variables con la configuración para no repetir el código.
- Se invoca la función `createServer` del módulo `http` que recibe como parámetro otra función; esta función tiene dos parámetros especiales: *request* (`req`) y *response* (`res`), que equivalen a toda la información de la petición que ha llegado a nuestro servidor Web y el objeto que llevara la respuesta al cliente respectivamente. Dentro de esta función SIN IMPORTAR qué tipo de petición sea, SIEMPRE estaremos devolviendo una respuesta válida 200 (OK). Se establece el formato de esta respuesta como texto plano (*text/plain*) y, finalmente, terminamos la respuesta con un texto (*Hello World*).
- Luego, iniciamos el servidor para que escuche todas las peticiones entrantes en el

puerto y dirección indicados con la función `listen` ; esta acepta adicionalmente otro parámetro que es otra función que se ejecuta cuando este se inicia de manera correcta.

Ahora procedemos a ejecutar nuestro servidor:

```
npm start
```

Para ejecutar el comando anterior es necesario configurar el script correspondiente en la sección *scripts* del archivo `package.json` , que se generó con la iniciación del proyecto o el comando alternativo `node index`

Si accedemos en el navegador Web a la dirección: `http://localhost:3000` veremos la respuesta de `Hello World` .

Antes de procesar nuestras peticiones para saber qué archivo requiere nuestro usuario, creamos dentro de nuestro proyecto el directorio `public` (el nombre no es estricto, pero es uno muy utilizado para colocar los archivos estáticos):

```
mkdir public
```

Allí dentro del directorio creado colocaremos dos archivos `index.html` y `about.html` con el respectivo contenido:

```
<html>
  <head>
    <title>Index</title>
  </head>
  <body>
    <h1>Index</h1>
  </body>
</html>
```

```
<html>
  <head>
    <title>About</title>
  </head>
  <body>
    <h1>About</h1>
  </body>
</html>
```

Muy básico el contenido, pues el objetivo del ejercicio es cómo procesar las peticiones y mostrar los respectivos archivos.

Realizemos este proceso paso a paso. Comencemos primero por añadir los módulos para procesar la dirección url y leer el archivo guardado en el sistema:

```
const http = require('http');
const url = require('url');
const fs = require('fs');
const path = require('path');

const hostname = '127.0.0.1';
const port = 3000;

...
```

Todos los anteriores módulos nativos de Node.js, dentro de la función que procesa las peticiones de nuestro servidor Web. Procesamos la dirección url de la petición:

```
...

const server = http.createServer((req, res) => {
  const pathurl = url.parse(req.url, true);
  const filepath = `${path.resolve('.')}/public${pathurl.pathname}`;

  ...
});

...
```

- El módulo `url` nos permite procesar (`parse`) la petición (`req`) del usuario y se almacena su resultado en la variable `pathurl` .
- Luego, utiliza el módulo *core* de `path` para extraer la ruta del directorio actual. El directorio donde se están almacenando los archivos (`public`) y, finalmente, cuál es el archivo que estamos requiriendo, se extrae del objeto `pathurl` el nombre de la ruta de la propiedad `pathname` y se crea la dirección absoluta del archivo en el sistema. Por último, lo se guarda en la variable `filepath` .

A continuación se comprobará que el archivo existe, y si es así se procede a leerlo:

```
...

const server = http.createServer((req, res) => {
  const pathurl = url.parse(req.url, true);
  const filepath = `${path.resolve('.')}/public${pathurl.pathname}`;

  fs.readFile(filepath, 'utf-8', (err, file) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    res.end(file);
  });
});

...
```

- Se ejecuta la función para leer el archivo independientemente si existe o no.
- Si el archivo que buscándose busca existe, procedemos a leerlo del sistema. Si la operación es exitosa establecemos la respuesta de la petición como 200 (OK). El tipo de contenido esta vez es html. Se escribe el contenido del archivo en la respuesta y se finaliza esta misma.

Notesé que estamos utilizando el patrón de *callbacks*, las dos funciones que utilizamos son asincrónicas, por lo tanto, hasta que no finalizemos la petición (u ocurra un *timeout*) no se termina la petición del usuario.

Procedemos a reiniciar nuestro servidor Web y a probar las diferentes direcciones:

1. `http://localhost:3000/index.html`
2. `http://localhost:3000/about.html`

Una última modificación a este código será dar una respuesta apropiada en caso de que el archivo que estamos requiriendo no exista:

```
...

const server = http.createServer((req, res) => {
  const pathurl = url.parse(req.url, true);
  const filepath = `${path.resolve('.')}/public${pathurl.pathname}`;

  fs.readFile(filepath, 'utf-8', (err, file) => {
    if (err) {
      res.statusCode = 404;
      res.setHeader('Content-Type', 'text/plain');
      res.write('404 Not Found');
      res.end();
      return;
    }
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    res.end(file);
  });
});

...
```

Con esto, al menos estamos garantizamos dar una respuesta correcta en caso de solicitar un archivo que no se encuentre en el sistema.

Unas modificaciones finales a este archivo podrían ser:

1. Devolver archivos solo con el verbo HTTP correcto; por ejemplo, para leer un archivo es el método GET, esto se puede hacer con `req.method` .
2. Detectar que si solicita un directorio y, si dentro de este existe el archivo `index.html` , lo devolvamos por defecto.

[Código fuente de la aplicación](#)

Más información:

- [HTTP headers](#)
- [HTTP request methods](#)
- [HTTP status codes](#)

Renderizar vistas dinámicas

Como vimos en la sección anterior, podemos renderizar todas las páginas Web HTML que deseemos, pero una de las ventajas de estar del lado del servidor es poder procesar la solicitud antes de responder al cliente. Por ejemplo, hacer cálculos, consultar una base de datos o preprocesar cada página Web.

Una de las librerías más utilizadas para manejar vistas dinámicas es *pug*, que nos permite bajo su sintaxis: crear una versión más corta del mismo HTML, renderizar variables enviadas desde nuestro servidor, tener condicionales para mostrar u ocultar información, hacer ciclos repetitivos para imprimir información y muchas otras opciones.

Basado en el proyecto anterior de `web-server` instalamos la librería:

```
npm i -S pug
```

La incluimos junto con las demás librerías en el encabezado de nuestro archivo principal de entrada `index.js` :

```
const http = require('http');
const url = require('url');
const fs = require('fs');
const path = require('path');
const pug = require('pug');

...
```

Renombramos el directorio `public` por `views` , pues este ya no contendrá archivos estáticos y, además, este último es un nombre más apropiado para nuestros archivos de vista dinámicas:

```
mv public views
```

Adicionalmente, renombramos nuestros archivos estáticos de `.html` a `.pug` :

```
mv views/index.html views/index.pug
mv views/about.html views/about.pug
```

Actualizamos su contenido con la sintaxis de pug del `index.pug` y `about.pug` respectivamente:

```
html
  head
    title Index
  body
    h1 Index
```

```
html
  head
    title About
  body
    h1 About
```

Al final el resultado será el mismo, pero *pug* nos simplifica de alguna manera su escritura.

Además, procedemos a realizar ciertos cambios en nuestro archivo de `index.js` para poder compilar y renderizar nuestra vista:

```
...

const server = http.createServer((req, res) => {
  const pathurl = url.parse(req.url, true);
  const filename = path.basename(pathurl.pathname, '.html');
  const filepath = `${path.resolve('.')}/views/${filename}.pug`;

  fs.readFile(filepath, 'utf-8', (err, file) => {
    if (err) {
      res.statusCode = 404;
      res.setHeader('Content-Type', 'text/plain');
      res.write('404 Not Found');
      res.end();
      return;
    }
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    const html = pug.render(file);
    res.end(html);
  });
});

...
```

- Almacenamos en la variable `filename` el nombre del archivo sin la extensión, pues el usuario realizará la petición del archivo en `index.html`, pero ahora lo tenemos almacenado con extensión `.pug`.
- Cambiamos el lugar donde buscamos los archivos, pues ahora es en el directorio `views` y anexamos la extensión `.pug`.

- Finalmente, almacenamos el contenido de la compilación de la vista en la variable `html`, que es la enviada como respuesta al usuario.

Si ejecutamos nuestro servidor Web y probamos las direcciones:

1. `http://localhost:3000/index.html`
2. `http://localhost:3000/about.html`

Todo debería estar funcionando como la versión anterior del proyecto. Hasta el momento, sólo hemos logrado acortar la escritura de la sintaxis de los archivos resultantes HTML; pero *pug* tiene muchas otras ventajas, como enviar variables y renderizarlas o utilizarlas para tomar decisiones de mostrar bloques o no.

Imaginemos que el mismo título es para todas las páginas y queremos enviarle el menú de navegación de manera dinámica. Modificamos nuevamente nuestro archivo `index.js`:

```
...

fs.readFile(filepath, 'utf-8', (err, file) => {
  if (err) {
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain');
    res.write('404 Not Found');
    res.end();
    return;
  }
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  const html = pug.render(file, {
    pageTitle: 'Our great Website',
    menu: [{
      name: 'Home',
      link: 'index.html'
    }, {
      name: 'About',
      link: 'about.html'
    }]
  });
  res.end(html);
});

...
```

Enviamos dos variables a la vista: El título de las páginas y un Array que contiene las diferentes opciones del menú de navegación. Ahora procedemos a modificar las vistas utilizando la sintaxis de *pug* para escribir variables y hacer ciclos repetitivos para imprimir una estructura como un Array. Primero hacemos el proceso con nuestro `views/index.pug`:

```
html
  head
    title= pageTitle
  body
    h1 Index
    ul
      each item in menu
        li
          a(href=item.link)= item.name
```

- En vez de escribir un texto estático en la etiqueta `title` , ahora estamos imprimiendo el contenido de la variable `pageTitle` . Para ello, *pug* utiliza el signo igual (`=`) justo después del nombre de la etiqueta, para indicar que lo que va a procesar es una variable y no un texto.
- En una lista no ordenada (`ul`) utilizamos la directriz `each` para iterar el Array `menu` y en cada iteración almacena en la variable `item` el elemento correspondiente. Luego procedemos a crear un enlace con la etiqueta `a` dentro de cada elemento de la lista (`li`). El enlace lo extraemos de la propiedad `link` y el nombre, que va dentro del enlace, de la propiedad `name` , ambas del respectivo `item` .

Realizamos este mismo proceso con el archivo `views/about.pug` pero en el `h1` dejamos el texto `About` para poder distinguir los archivos.

Reiniciamos el servidor, y esta vez podemos utilizar los enlaces del menú para navegar entre las diferentes páginas.

Código fuente de la aplicación

Más información:

- [Pug - Getting Started](#)
- [Pug - Github](#)

Utilizar Express JS

En este capítulo de servidor Web comenzamos a realizar las operaciones con solo módulos nativos de Node.js. Luego incorporamos *pug* como librería para compilar plantillas dinámicas. La siguiente es Express JS, que es un framework para Node.js, muy popular en la comunidad, tanto que la fundación de Node.js la acogió para su desarrollo y crecimiento. Contiene muchas de las funcionalidades que hemos implementado de manera nativa, por lo tanto, nuestro código se verá bastante simplificado. Realizaremos este proceso paso a paso, primero vamos a instalar la librería:

```
npm i -S express
```

Luego incorporamos la librería y removemos las que no necesitaremos en el encabezado de nuestro archivo `index.js` :

```
const express = require('express');
const pug = require('pug');

const hostname = '127.0.0.1';
const port = 3000;
const menu = [{
  name: 'Home',
  link: 'index.html'
}, {
  name: 'About',
  link: 'about.html'
}];

...
```

Como podemos observar, solo necesitaremos `express` y `pug` . Adicionalmente creamos la variable `menu` para almacenar el menú que enviamos a las plantillas dinámicas. A continuación vamos a iniciar la aplicación de `express` , establecer su configuración y rutas:

```
...

const app = express();
app.set('view engine', 'pug');

app.get('/index.html', function (req, res) {
  res.render('index', {
    pageTitle: 'Our great website - Index',
    menu: menu
  })
});

app.get('/about.html', function (req, res) {
  res.render('about', {
    pageTitle: 'Our great website - About',
    menu: menu
  })
});

app.use((req, res) => {
  res.status(404);
  res.set('Content-Type', 'text/html');
  res.end('Error. Route not found');
});

app.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Como podemos ver en el fragmento de código anterior, toda nuestra aplicación se redujo drásticamente a unas cuantas líneas, esto debido a lo que habíamos mencionado anteriormente, pues *express* contiene muy buenas prácticas y resume la mayoría de funcionalidades comúnmente usadas. A continuación el detalle de los cambios:

- Iniciamos la aplicación de *express* y almacenamos su instancia en la variable `app`.
- Establecemos que el procesador de vistas dinámicas será *pug* en la configuración de la aplicación. Vemos como *express* se integra muy fácilmente con *pug*, e inclusive con otras librerías similares.
- Ahora veamos el concepto de "ruta" introducido por *express*. Su definición consta de un verbo HTTP, en este caso GET, que es el utilizado cuando se solicita un recurso en la Web, es decir, para obtener una página desde la dirección url, como parámetros recibe: primero una cadena que establece la "ruta", la cual se comparará con la solicitada por el usuario (esta debe ser exacta) y finalmente una función *middleware* (este concepto también introducido por *express*, que a su vez recibe dos parámetros: la petición (`req`) y la respuesta (`res`) como habíamos visto en la primera sección de este capítulo).
- Dentro de la función *middleware* sencillamente renderizamos la vista correspondiente y

enviamos los parámetros a dicha vista. Detallemos esto un poco más: *express* añade la función `render` al objeto `res`, conoce la ubicación de las vistas que por defecto es `views/` (esto se puede cambiar en la configuración) y finalmente la función `render` finaliza la petición.

- Lo mismo hacemos con la ruta de `about`, renderizando la vista correspondiente con sus respectivos parámetros.
- Si ninguna de las rutas anteriores hace coincidencia, entonces agregamos un *middleware* por defecto con la función `use`. Dentro de esta función establecemos el código de respuesta como 404 (*Not found*), el tipo de contenido que va a tener la respuesta (`text/html`) y la información de esta misma.
- Finalmente, cambiamos la variable `server` por `app`, ya que *express* también contiene la función `listen` que realiza exactamente lo mismo que la homónima en el módulo `http`.

Hemos logrado conseguir la misma funcionalidad con menos líneas de código, adoptando mejores prácticas y habilitando muchas más opciones, como se verá a continuación.

Mostrar archivos estáticos

Hasta el momento solo tenemos plantillas dinámicas, pero ¿qué pasa con los archivos estáticos como por ejemplo: imágenes, hojas de estilo, documentos, etc.? *express* nos permite "servir" muy fácilmente todo un directorio (o varios directorios) de archivos estáticos. Como lo hicimos en la primera sección de este capítulo, nos tocaría comprobar si el archivo solicitado existe, luego leerlo del sistema y finalmente enviarlo como respuesta. *express* realiza todas estas operaciones automáticamente, creemos (nuevamente) nuestro directorio `public`:

```
mkdir public
```

Dentro de este directorio creamos una hoja de estilos llamada `styles.css` con el siguiente contenido:

```
body {
  font-family: Arial, Helvetica, sans-serif;
}

h1 {
  text-align: center;
}

.menu {
  padding: 0;
  text-align: center;
}

.menu li {
  display: inline-block;
  list-style-type: none;
  margin: 8px;
}

.menu li a {
  background-color: navy;
  border-radius: 24px;
  color: white;
  padding: 12px 24px;
  text-decoration: none;
}

p {
  text-align: center;
}
```

Unos simples estilos para darle un poco de apariencia a nuestra página. Creamos la clase `.menu`, la cual asignaremos más adelante en la plantilla dinámica. A continuación, configuramos express para que sirva el directorio de archivos estáticos:

```
...

const app = express();
app.set('view engine', 'pug');
app.use(express.static('public'));

...
```

Esta simple línea que acabamos de añadir, se encargará de todo el trabajo; es decir, que podríamos acceder mediante la dirección url a `http://localhost:3000/styles.css` y ver el archivo. *Express* también permite cambiar de punto de montaje (no necesariamente todos los archivos estáticos tendrían que estar en la raíz de la aplicación), podría ser `assets/styles.css`, depende de la configuración.

Procedemos a cambiar las plantillas para cargar la hoja de estilos y asignarle la clase de menu que creamos. Ante todo, primero comenzamos con `views/index.pug` :

```
html
  head
    title= pageTitle
    link(href='styles.css', rel='stylesheet')
  body
    h1 Index
    ul.menu
      each item in menu
        li
          a(href=item.link)= item.name
    p Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor i
ncididunt ut labore et dolore magna aliqua.
```

Los cambios son con la sintaxis de *pug*. Adicionalmente colocamos un párrafo con texto de ejemplo para darle un poco más de contenido a la página. Los mismos cambios pueden ser aplicados a la plantilla dinámica de `about` .

Completamos un proyecto para un servidor Web utilizando las librerías de *express* y *pug*.

[Código fuente de la aplicación](#)

Más información:

- [Express JS](#)
- [Using templates engines with Express](#)
- [Serving static files in Express](#)
- [Express routing](#)

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 asíncronos y código no bloqueante.

En la página de Node.js se provee una excelente [guía sobre principales los conceptos](#) y en [la documentación del API](#) 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