

Code

Issues

Pull requests 1

Actions

Projects







Security

...

 master ▾

...

[FT-M3](#) / [01-Node](#) /

 nachovip ...	on 7 Apr 
..	
 demo	17 months ago
 homework	16 months ago
 README.json	7 months ago
 README.md	9 months ago



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quizz teórico de esta lecture.

Introducción a Node.JS



¿Qué es Nodejs?

Conceptos

Para poder entender el concepto de node, primero vamos a tener que conocer otros conceptos más básicos sobre estos temas:

- Procesadores
- Lenguaje de máquina
- C++

Podemos pensar a procesador (microprocesador) como una pequeña máquina que recibe impulsos eléctricos como entrada y genera salidas al que nosotros le pasamos *instrucciones*.

☰ README.md

set de instrucciones. Como se imaginarán cada fabricante tiene set de instrucciones distintos y por lo tanto distintos lenguajes, mencionemos algunos:

- IA-32
- x86, x86-64
- ARM
- MIPS

Para pasarle intrucciones al procesador, no escribimos 1's y 0's, sino que usamos un lenguaje que se traduce directamente a esa secuencia, llamado **assembler**.

```

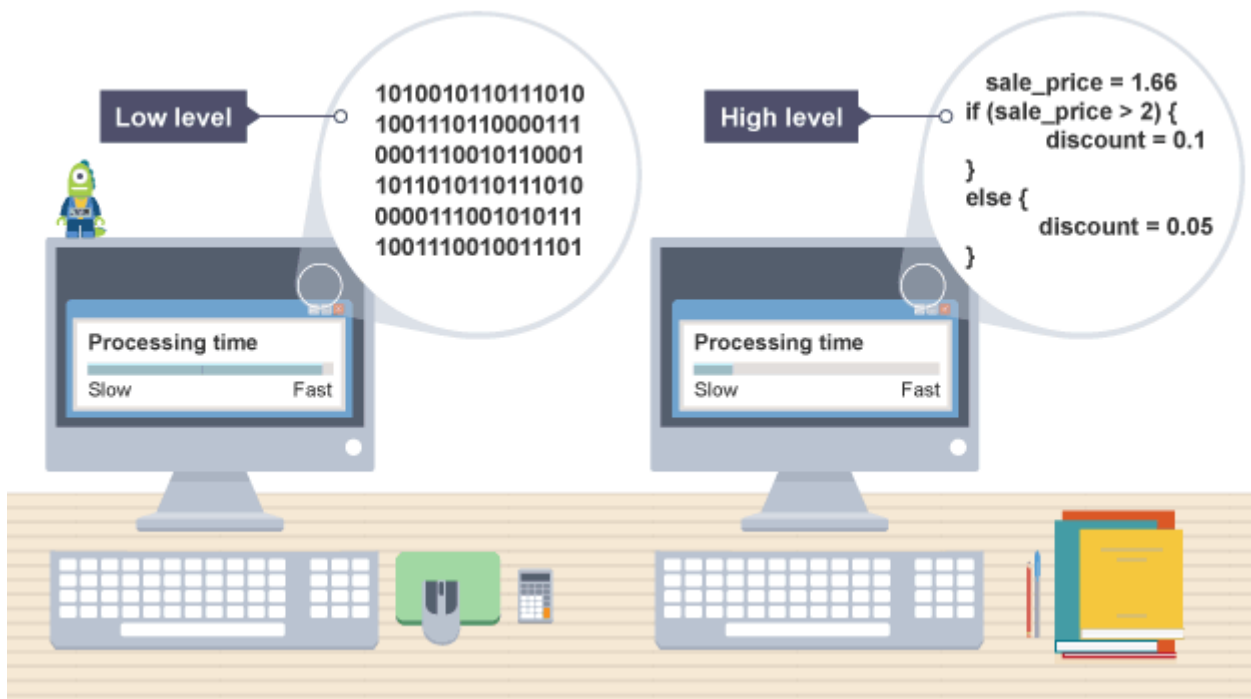
-u 100 1a
OCFD:0100 BAOB01      MOV    DX,010B
OCFD:0103 B409      MOV    AH,09
OCFD:0105 CD21      INT     21
OCFD:0107 B400      MOV    AH,00
OCFD:0109 CD21      INT     21
-d 10b 13f
OCFD:0100                48 6F 6C 61 2C
OCFD:0110                20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67
OCFD:0120                72 61 6D 61 20 68 65 63-68 6F 20 65 6E 20 61 73
OCFD:0130                73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 20
OCFD:0140                57 69 68 69 70 65 64 69-61 24

```

Hola,
este es un prog
rama hecho en as
sembler para la
Wikipedia\$

Hoy en día no se programa en assembler (**lenguaje de bajo nivel**), ya que es muy complejo y hacer un simple 'Hello World' podría llevar muchas líneas de código: Simplemente no escala. Para solucionar esto, se crearon lenguajes más fáciles y rápidos de programar y que compilan a lenguaje de máquina, estos son los conocidos *lenguajes de alto nivel*, JAVA, C++, Javascript son ejemplos de estos lenguajes. Es importante notar, que no importa que lenguaje usemos, en algún momento el código será *traducido* o *compilado* a lenguaje de máquina, que es el único lenguaje que entiende verdaderamente la computadora.

Como pueden pensar mientras nos alejamos del lenguaje de máquina (lenguajes de más alto nivel) y nos abstraemos vamos ganando velocidad para codear, pero también vamos perdiendo performance. Piensen que si codeamos en lenguaje de máquina, podemos controlar cada slot de memoria nosotros mismos y hacerlo de la mejor forma posible. Cuando subimos de nivel, alguien hace ese trabajo por nosotros, y como tiene que ser genérico no puede lograr ser tan óptimo. Es por eso que según la performance y los recursos que se necesite o tengamos vamos a elegir lenguajes de altísimo o bajísimo nivel.



Por ejemplo, los microcontroladores embebidos en lavaropas están programados en C compilado a lenguaje de máquina, esto es porque tienen muy poca memoria y tienen que optimizarla al máximo

C++

C++ es un lenguaje de programación de bajo nivel, estaría justo por arriba de Assembly. Este lenguaje se hizo muy popular porque es fácil para codear, pero a su vez te deja tener bastante control sobre lo que está pasando en nivel hardware. Muchos otros lenguajes se construyeron en base a C++, o siguiendo sus sintaxis o usándolo en su cadena de abstracción.

Justamente *Nodejs* está programado en C++! La razón por la que nodejs está escrito en C++ es porque **V8** está escrito en C++, ahora veamos qué es V8.

Motor de Javascript

Antes que nada es importante entender que el lenguaje JavaScript está basado en un standard que se conoce como **ECMAScript**. Este standard setea las bases de qué cosas deberá hacer el lenguaje y cuales no, y de qué manera. Ahora bien, en el mundo real no se respetan los estándares al 100%, es por eso que hay muchas implementaciones distintas de JavaScript, que va a interpretar (convertir el código a lenguaje que pueda ser corrido por la compu) el código de una manera particular, estas implementaciones son los llamados *motores javascript*.

Ve ocho es el motor de javascript creado por **Google** para su browser *Chrome*. Es un proyecto Open Source así que podemos investigar su **código**, en su página Google define a v8 como:

- V8 JavaScript Engine: *ya sabemos*.
- V8 is Google's open source JavaScript engine: *really?*.
- V8 implements ECMAScript as specified in ECMA-262: *ah!, sigue estándares, bien!*.
- V8 is written in C++ and is used in Google Chrome, the open source browser from Google: *Alguien conoce ese browser Chrome, es bueno?*.
- V8 can run standalone, or can be embedded into any C++ application: *Atención con esto, se puede embeber (usar) en cualquier aplicación C++*

Releamos el último bullet en detalle: Es Standalone, eso quiere decir que puedo bajar V8 y correrlo en mi compu, no necesariamente dentro del browser, genial!; puede ser embebido, es decir que puedo codear una aplicación en C++ y agregarle todas las funciones de V8, copado!! Ya se imaginan donde empezó a nacer *Nodejs*??

¡Sí! Nodejs es justamente una aplicación escrita en C++ que embebe el motor V8 en su código. Por lo tanto puede interpretar código javascript, 'traducirlo' a lenguaje de máquina y finalmente ejecutarlo. Pero eso no es lo mejor de Nodejs, lo mejor es que el creador agregó algunos features que no están definidos en el estándar. Javascript originalmente estaba diseñado para correr en el browser, o sea que nadie pensó en que pudiera leer archivos, o conectarse a una base de datos, etc... Justamente estas features son las que Nodejs agrega usando código C++. O sea, crea nuevas funciones Javascript que envuelven en realidad funciones de C++, como por ejemplo la función de leer un archivo del filesystem.

Esto hace que NodeJs sea muy poderoso! De hecho, gracias a esto NodeJs tiene todas las features necesarias para poder manejar un servidor.

- Maneras de organizar nuestro código para que sea reusable
- Poder leer y escribir archivos (input/output)
- Leer y escribir en Bases de Datos
- Poder enviar y recibir datos de internet
- Poder interpretar los formatos estándares
- Alguna forma de manejar procesos que lleven mucho tiempo

Instalar Nodejs

Para instalar node vamos a ir [acá](#) y seguir las instrucciones según el sistema operativo que estés usando.

Una vez terminada la instalación, podemos probar si funciona correctamente escribiendo el siguiente comando en la consola:

```
node -v
```

Si el resultado es algo de la forma: `v6.3.1` entonces habremos instalado Node de manera correcta!

Core libraries

Nodejs cuenta con un set de librerías (les vamos a llamar módulos) que vienen por defecto en la instalación. Básicamente es código escrito para hacer tareas muy comunes. Podemos separar estas librerías en las que están escritas en C++ y las que son nativas o están escritas en *javascript*.

C++

Como dijimos antes, usando el motor V8 los desarrolladores de Nodejs agregaron funcionalidad que ECMAScript no tenia en el standard. La mayoría de estas funciones tiene que ver con tareas que también involucran al Sistema operativo, como leer archivos, mandar y recibir datos por la red, comprimir y descomprimir archivos y otras de manejo de streams.

Javascript

Estas librerías están escritas en Javascript, algunas de ellas implementan la funcionalidad usando javascript, pero la mayoría son sólo envoltorios a las funciones escritas en C++ para que puedan ser fácilmente utilizadas por los desarrolladores de Nodejs. Éstas son el tipo de librerías que podrías haber escrito vos mismo, por suerte alguien ya se tomó el trabajo!

Organizando nuestro código en Nodejs

Una de las features que hizo que Nodejs creciera tanto, es justamente el ecosistema de librerías desarrolladas por los usuarios, que sumadas a las librerías *Core* hacen que sea muy potente. Ahora vamos a ver cómo poder utilizar módulos en nuestro código, y luego veremos cómo instalar módulos externos.

Módulos

Def: Un bloque de código reusable, cuya existencia no altera accidentalmente el comportamiento de otros bloques de código. Este concepto ya existía en otros lenguajes de programación y era muy usado para estructurar proyectos. De todos modos, los módulos no eran parte del standard en ECMAScript (lo agregaron en la última versión), pero Nodejs lo introdujo bajo el nombre de *CommonJs Modules*, que era básicamente un acuerdo (un standard) sobre cómo deberían estar estructurados los módulos.

Cómo funciona CommonJs Modules?

Básicamente, el standard dice lo siguiente:

- Cada archivo es un módulo, y cada módulo es un archivo separado.
- Todo lo que queremos exportar va a ser expuesto desde un único punto.

Para entender el standard tenemos que tener dos conceptos en claro:

- **First-Class Functions:** Las funciones en javascript son tratadas igual que cualquier otro objeto, es decir que podés guardarlas en variables, pasarlas como argumentos, guardarlas en arreglos, etc...
- **Function Expressions:** Como las funciones son first-class, al escribir una expresión de función estoy creando la misma, eso quiere decir que puedo crear funciones en cualquier parte del código.

Vamos a empezar construyendo nuestro propio módulo! una vez que lo tengamos, vamos a ver como usarlo. Entendiendo así todo el ciclo.

Construyendo un módulo

Empezamos con un un archivo `js` vacío. Lo llamaremos `hola.js`, esté será nuestro primer módulo. Lo único que hará este módulo es saludar:

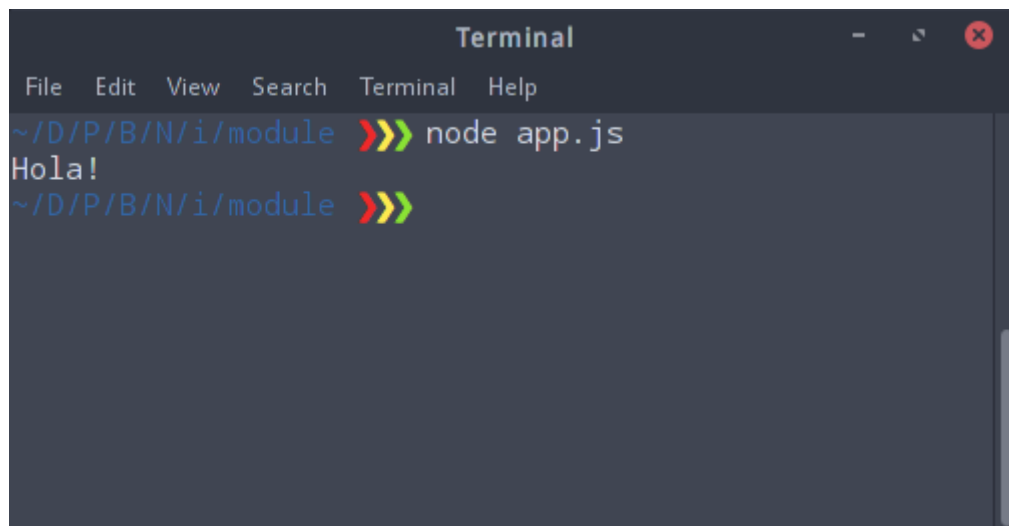
```
console.log('Hola!');
```

Tambien vamos a crear un archivo `js` que se llame `app.js`, en donde vamos a utilizar el código de `hola.js`. Dentro de `app.js` vamos a llamar a la función **require** que es parte de las core libraries de Nodejs:

```
require('./hola.js');
```

Require recibe como argumento, un string que es el path en donde encontrará el código o el módulo que queremos agregar, en este caso como `hola.js` está en la misma carpeta el path es: `./hola.js`.

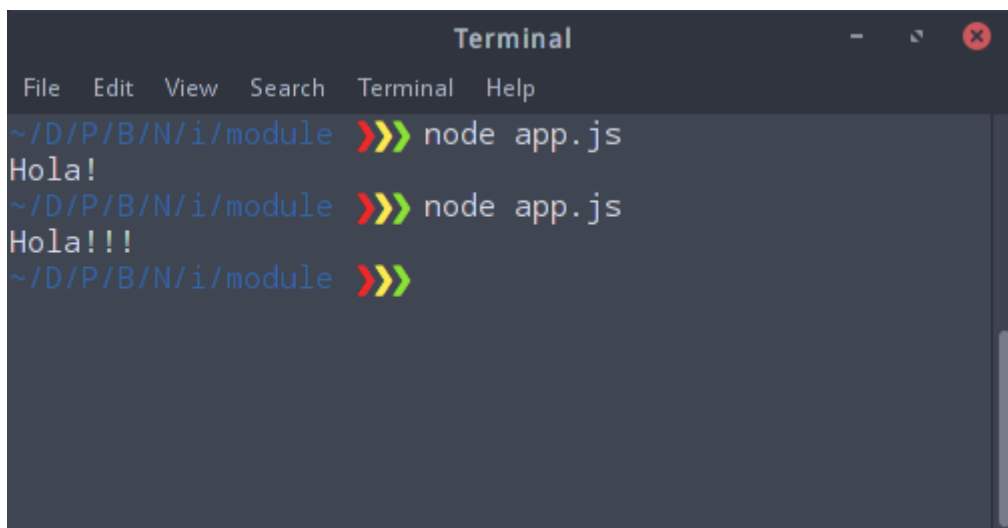
Ahora, si corremos el archivo `app.js` usando `node app.js` en la terminal vamos a ver que se ejecutó el código que escribimos en `hola.js`:

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command `node app.js` being executed, which results in the output `Hola!`. The prompt `~/D/P/B/N/i/module` is visible before and after the command.

Hagamos algo más interesante, vamos a `hola.js` y creemos una función y luego la usemos para saludar:

```
var saludar = function() {  
  console.log('Hola!!!');  
}  
saludar();
```

Corramos de nuevo `app.js`:

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the following commands and output:

```
~/D/P/B/N/i/module >>> node app.js
Hola!
~/D/P/B/N/i/module >>> node app.js
Hola!!!
~/D/P/B/N/i/module >>>
```

El resultado es el mismo! Ahora, si no invocamos `saludar()` dentro de `hola.js`, creen que la podremos invocar (usar) en `app.js`? Hagamos la prueba!

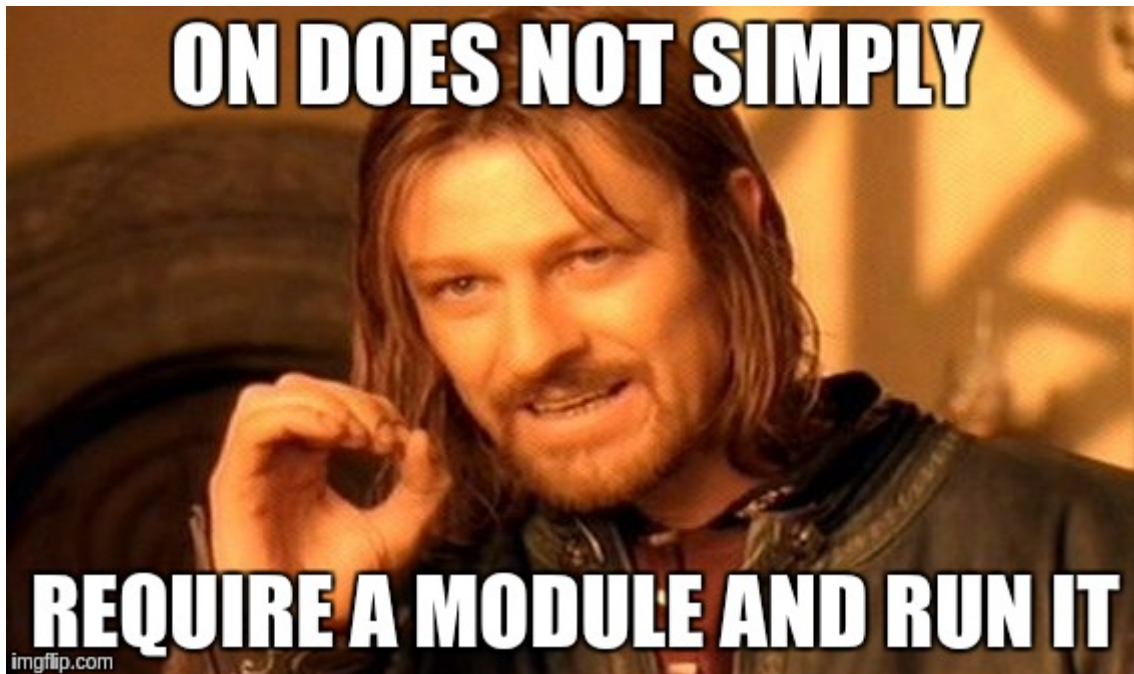
Comentamos la invocación en `hola.js`:

```
var saludar = function() {
  console.log('Hola!!!');
}
//saludar();
```

Invacamos en `app.js`:

```
require('./hola.js');
saludar();
```

`saludar` no está definido! recibimos un error. De hecho, esto está bien que suceda. Se acuerdan que dijimos que un módulo no debería afectar otro código accidentalmente? Eso quiere decir que el código está protegido y que no podemos simplemente usarlo y acceder a los objetos fuera de ese módulo.



Para usarlo afuera, vamos a tener que explicitarlo, veamos como hacer que podamos usar `saludar()` desde `app.js`.

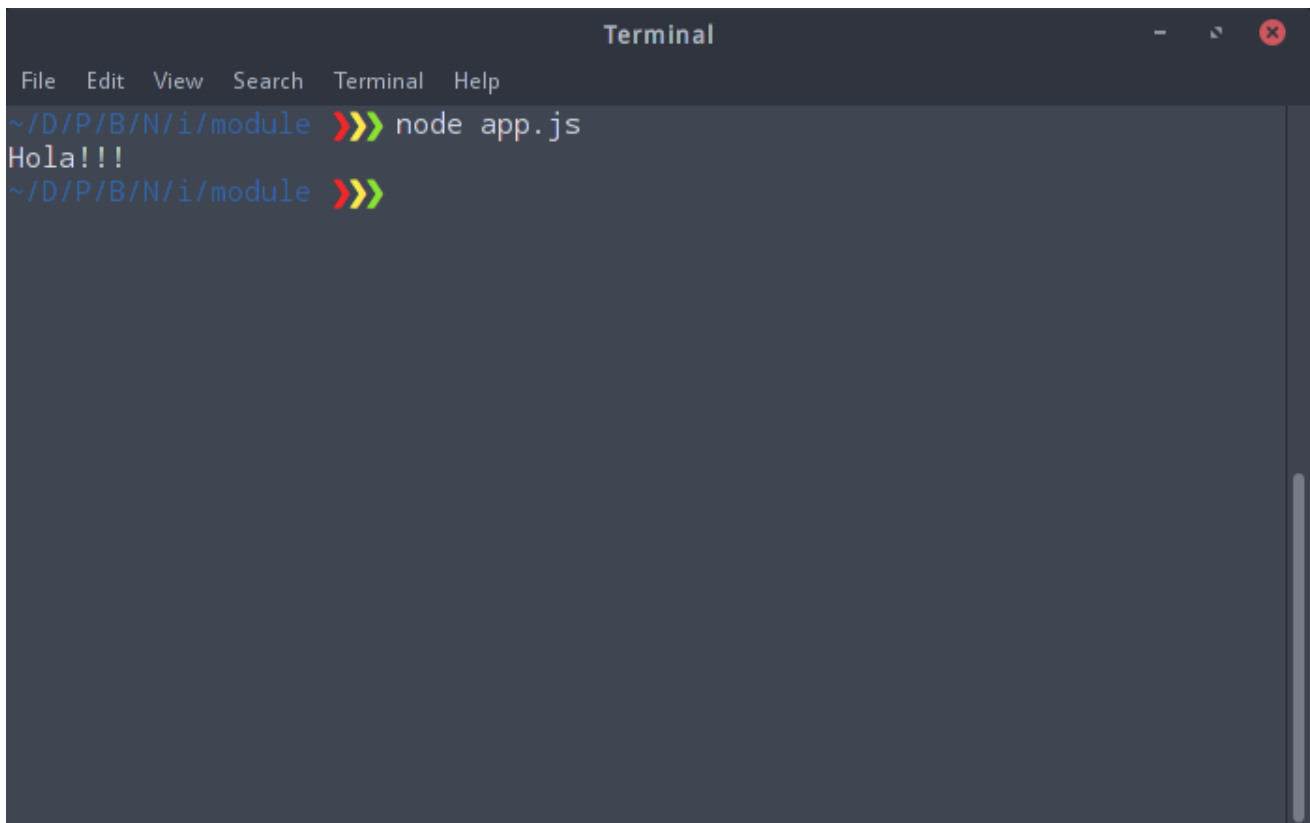
Nodejs nos permite hacerlo usando `module.exports`, que es una variable especial cuyo objetivo es pasar su contenido a otro pedazo de código cuando llamamos a `require`:

```
var saludar = function() {  
  console.log('Hola!!!');  
};  
  
module.exports = saludar;
```

Ahora este módulo está exponiendo el objeto `saludar`. Para usarlo en nuestro módulo tenemos que guardar lo que devuelve `require` en una variable (puedo llamar a la nueva variable como quiera):

```
var hola = require('./hola.js');  
  
hola();
```

Ahora voy a poder invocar `hola()`, porque lo hemos pasado a través de `module.exports`. Si ejecuto `app.js`:



```
Terminal
File Edit View Search Terminal Help
~/D/P/B/N/i/module >>> node app.js
Hola!!!
~/D/P/B/N/i/module >>>
```

Ahora pudimos acceder al código de `hola.js` , porque lo expusimos a propósito.

Resumiendo:

- **Require** es una función que recibe un *path*.
- **module.exports** es lo que retorna la función *require*.

Algunos patrones de Require

Como siempre en Nodejs, hay muchas formas de hacer lo mismo (esto puede ser bueno o malo - según cómo lo usemos) y crear módulos no es la excepción. Veamos algunos patrones comunes en la creación de módulos!

Múltiples imports

La función *require* busca primero un archivo con el nombre que le pasamos en la carpeta que le pasamos (de hecho no es necesario poner la extensión '.js'). Si no encuentra ese archivo, entonces busca una carpeta con ese nombre y busca un archivo `index.js` el cual importa. Esta funcionalidad la podemos usar como patrón para importar varios archivos: dentro de `index.js` importamos los demás archivos.

Imaginemos que queremos hacer una función que salude en distintos idiomas, vamos a tener un `app.js` de esta forma:

```
var saludos = require('./saludos');
```

```
saludos.english();  
saludos.spanish();
```

Estamos importando solamente el path `./saludos`, como no hay ningún archivo `.js` con ese nombre, *require* busca una carpeta, por lo tanto vamos a crear una carpeta `saludos` con los siguientes archivos:

index.js

```
var english = require('./english');  
var spanish = require('./spanish');  
  
module.exports = {  
  english: english,  
  spanish: spanish  
};
```

En este archivo estamos importando otros dos módulos, uno por cada idioma en el que queremos saludar. Lo bueno de esto, es que va a ser muy fácil agregar y sacar idiomas de nuestra aplicación, ya que solo debemos agregar o borrar los idiomas que exportamos!

Ahora veamos como sería cada idioma:

spanish.js

```
var saludos = require('./greetings.json');  
  
var greet = function() {  
  console.log(saludos.es);  
}  
  
module.exports = greet;
```

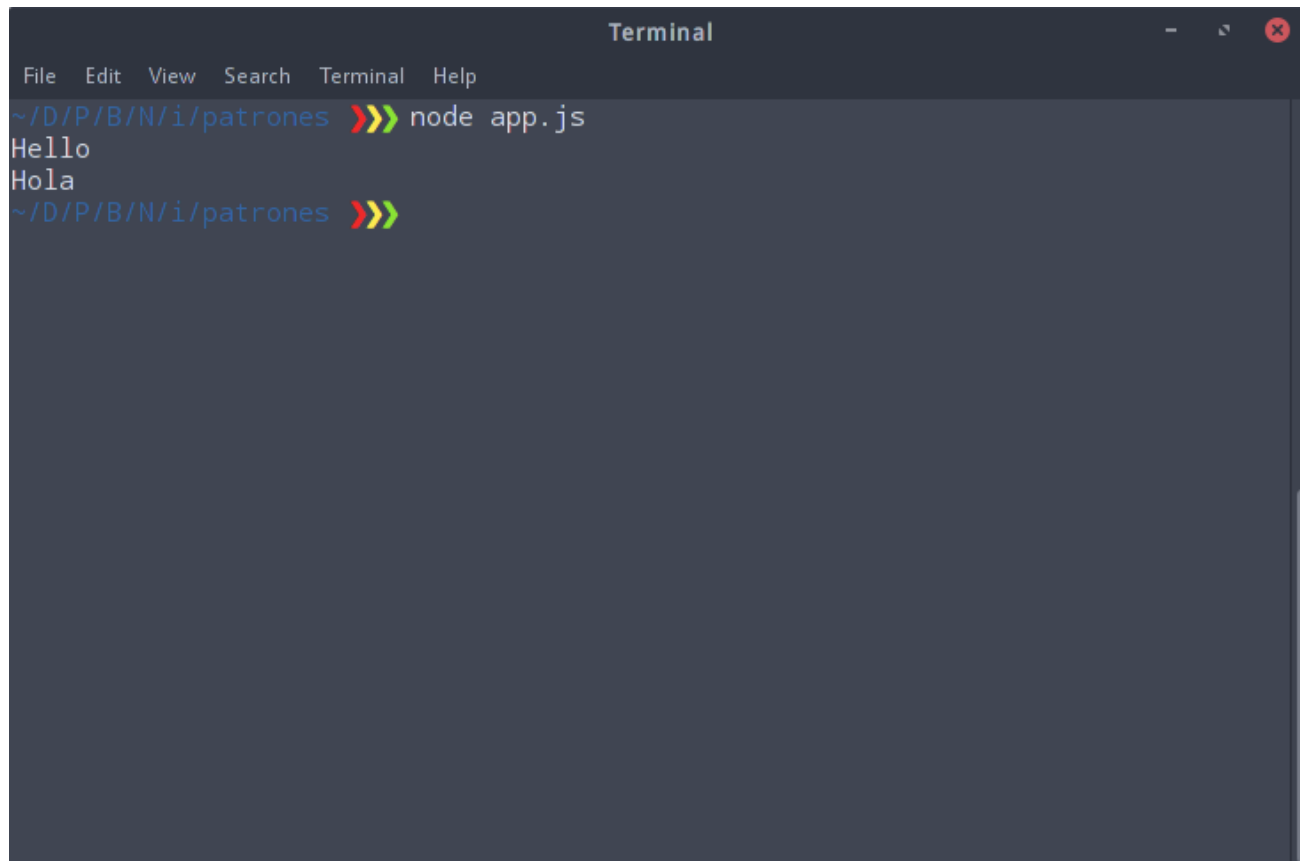
english.js

```
var saludos = require('./greetings.json');  
  
var greet = function() {  
  console.log(saludos.en);  
}  
  
module.exports = greet;
```

y en `greetings.json` vamos a tener los saludos per se:

```
{  
  "en": "Hello",  
  "es": "Hola"  
}
```

Si ejecutamos nuestra `app.js` :



The image shows a terminal window titled "Terminal" with a dark background. The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows the command `node app.js` being executed in the directory `~/D/P/B/N/i/patrones`. The output of the script is `Hello` followed by `Hola` on the next line. The prompt `~/D/P/B/N/i/patrones` is visible again on the third line.

Más Patrones

En la carpeta `./patrones/otro` hemos puesto varios files distintos llamados `saludos` que van del uno al cinco, y luego los importamos en el archivo `app.js`. En cada módulo exportamos lo que necesitamos de manera distinta, cada una de esas formas constituye un patrón.

Dentro de cada archivo en los comentarios está explicado en más detalle el patrón.

De nuevo, **no hay una mejor forma, prueben todos los patrones y vean cual es el que les gusta y cual los hace felices!**

Require con modulos Core o nativos

También podemos importar módulos CORE (los que hablamos al principio) usando `require`. Esta función está preparada para que si no encuentra un archivo con el nombre que le pasamos, busca una carpeta, y si no encuentra busca en los módulos nativos. Podemos ir a la [documentación](#) y ver los módulos que podemos usar.

Veamos como podemos importar algo de esa funcionalidad. Como ejemplo vamos a importar el módulo llamado `utilities`, en la doc vemos que el módulo se llama `util`. Por lo tanto lo vamos a importar escribiendo

```
var util = require('util'); // No usamos ./ porque es un modulo core

var nombre = 'Toni';
var saludo = util.format('Hola, %s', nombre);
util.log(saludo);
```

¿Que hace este código? Busquen en la [documentación](#) por la función `format()` y `log()` e intenten predecir que hará ese código.

Eventos: Events emitter

Una gran porción del core de Nodejs está construida usando como base este concepto. Un *Evento* es algo que ha ocurrido en nuestra aplicación y que dispara una acción. Este concepto no es exclusivo a *Nodejs*, si no que aparece en muchos lenguajes y arquitecturas de software. En Node hay dos tipo de eventos:

- Eventos del sistemas: Estos vienen del código en C++, gracias a una librería llamada *libuv* y manejan eventos que vienen del sistema operativo como por ejemplo: Terminé de leer una archivo, o recibí datos de la red, etc...
- Eventos Customs: Estos eventos estan dentro de la parte Javascript del Core. Maneja eventos que podemos crear nosotros mismos, para eso vamos a usar el *Event Emitter*. Varias veces cuando un evento de *libuv* sucede, genera un evento usando el *event emitter*.

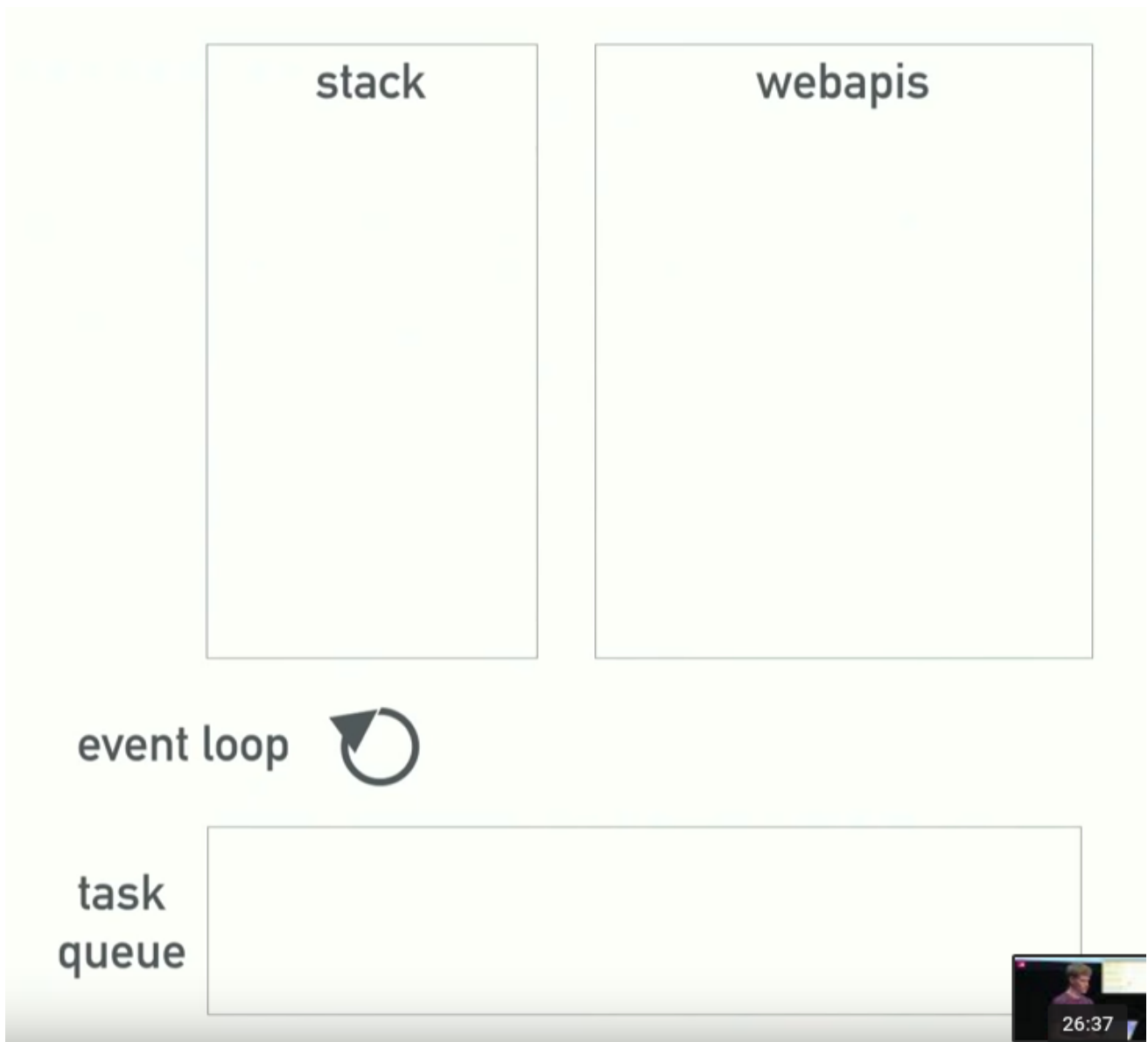
Ejercicio Crear un forma simple de emitir eventos y capturarlos.

Event Listener

Dijimos que cuando ocurre un evento, queremos capturarlo y responder de alguna forma, para eso vamos a hacer uso de los *Events Listeners*, básicamente es el código que *escucha* por un evento y que hace algo (ejecuta código) cuando ese evento sucede. Podemos tener varios listeners escuchando el mismo evento.

Event Loop

Repasemos como funcionaba V8 internamente y veamos qué es exactamente el **event loop**:



Los componentes que vemos en la figura son:

- Stack: es el runtime de V8, éste va leyendo el código del programa que esté corriendo y va poniendo cada instrucción en el stack para ejecutarla. **Javascript es Single threaded, o sea que sólo puede ejecutar una instrucción a la vez**
- SO / Webapis: Esta pila es manejada por el sistema Operativo. Básicamente V8 le delega tareas al SO, por ejemplo: Bajame esto de internet, o leeme este archivo, o comprímame esta imagen, etc... **El SO puede hacer estas tareas en un thread o varios, es transparente para nosotros.** Lo único que nos importa es cuando el SO *completó* la tarea que se le pidió. Cuando lo hizo, nos avisa y pasa el *callback* al task queue.
- Task Queue: Cuando el SO nos indica que terminó una tarea, no podemos simplemente pasar el *callback* al stack de JS (no sabemos que está haciendo y podría correr código en un momento inoportuno), por lo tanto, lo que hace es dejar el callback en el *Task Queue* (es una cola tipo FIFO). Ahora, cuando el Stack JS está vacío el TANTAN TATAN... **EVENT LOOP** agarra el *callback* del Queue y lo pasa al Stack JS para ser ejecutado!

Toda esta interacción con el SO, el manejo del Task Queue y el Event loop está implementado en la librería **libuv** que es la piedra angular de Nodejs. De hecho, su logo es un Unicornio T-rex, es demasiado copada:



¿Qué es un gestor de paquetes?

Primero definamos lo que es un paquete. Básicamente es.. código ! Es cualquier pieza de **código** manejada y mantenida por un gestor de paquetes.

Ahora, un gestor de paquetes es un software que automatiza la instalación y actualización de paquetes. Maneja qué versión de los paquetes tenes o necesitas y maneja sus *dependencias*. Una **dependencia** es código del cual dependen uno o más pedazos de código para funcionar. Por ejemplo, si usás `fs` en tu servidor, entonces `fs` es una dependencia de tu server. Sin el código de `fs` tu servidor no podría ejecutarse. De hecho, cada paquete podría tener sus propias dependencias, es por esto que manejarlos a mano se podría volver un poco engorroso, por suerte tenemos los gestores de paquetes que nos solucionan este problema.

NPM: Node Package Manager

Npm es el gestor de paquetes que viene con Nodejs y que gestiona los paquetes para este. Para probar si tenés npm podés hacer `npm -v` en tu consola, y si recibis algo como esto:
`3.10.5` quiere decir que lo tenés instalado.

Para ver el registro de paquetes podés ir a [esta](#) página. Acá podés buscar paquetes individuales, ver su información y bajarlos. Cualquiera puede subir sus paquetes en npm, así que tenés que tener cuidado con qué paquetes bajar.

Para instalar un paquete se utiliza el comando `npm` con el argumento `install` y el nombre del paquete. Por ejemplo, para instalar el paquete 'express' vamos a la consola y tipeamos:

```
npm install express .
```

Para poder trackear las dependencias y los paquetes que tenemos instalados, npm hace uso de un archivo de *configuración* al que llama **package.json**. Este es básicamente un archivo de texto en formato JSON con el listado de dependencias de tu aplicación, de esta forma con sólo compartir ese archivo cualquiera sabrá qué paquetes se deben instalar, e incluso hacerlo de forma automática.

Para crear este archivo npm nos da el comando `npm init`, que es una forma interactiva de crear el 'package.json'.

- entry point: Indica cuál es el archivo javascript que Node debe correr para arrancar la aplicación.

En la imagen vemos los datos que nos piden y a continuación vemos el archivo `package.json` que creó el comando:

```
{
  "name": "test-app",
  "version": "1.0.0",
  "description": "Esta es una aplicación de prueba",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Prueba"
  ],
  "author": "Toni Tralice",
  "license": "ISC"
}
```

Para que veamos como funcionan las dependencias, vamos a instalar algunos paquetes y requerirlos dentro de nuestra aplicación de prueba.

*Los paquetes instalados de forma local serán guardados en una carpeta llamada **node_modules** creada dentro de la carpeta donde ejecuté el comando*

moment

Es una librería de javascript para manejar fechas. Para instalarlo hacemos: `npm install moment --save`.

usamos el argumento `--save` para indicar a npm que además de instalar el paquete, lo agregue a la lista de dependencias en el archivo `package.json`.

Luego de ejecutar el comando, vemos que en `package.json` hay una nueva propiedad llamada 'dependencias' que es un objeto que contiene los nombres y las versiones de los paquetes que hemos instalado (siempre que instalemos usando `--save`).

```
{
  "name": "test-app",
  "version": "1.0.0",
  "description": "Esta es una aplicación de prueba",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Prueba"
  ],
  "author": "Toni Tralice",
  "license": "ISC",
  "dependencies": {
    "moment": "^2.14.1"
  }
}
```

Ahora creamos un archivo llamado `index.js` (el entry point de nuestra app), y dentro de él vamos a incorporar el nuevo módulo instalado.

```
var moment = require('moment');
console.log(moment().format("ddd/MM/YYYY, hA"));
```

Como verán, Nodejs ya sabe en qué carpetas buscar el módulo 'moment' y no tenemos que explicitarlo diciendo en qué carpeta está, de la forma `./node_modulos/moment`.

NPM paquete globales

Con lo anterior hemos instalado el paquete 'moment' al que podremos acceder desde la aplicación test. Ahora vamos a ver que hay paquetes de npm que nos son útiles en todas las aplicaciones que hacemos, por lo tanto los queremos instalar de manera *global*. Para hacerlos usamos el argumento `-g` en `npm install`.

Según el sistema operativo pueden llegar a tener algún problema con permisos, si es el caso pueden buscar soluciones en [esta página](#). La carpeta donde se instalan los módulos globales también dependen del SO.

Nodemon

Se acuerdan cuando hicimos el servidor web con Node? Cada vez que cambiamos algo en el código, teníamos que reiniciar el servidor para que los cambios sean reflejados. Con *Nodemon* podemos olvidarnos de eso, ya que hace eso por nosotros.

```
nodemon will watch the files in the directory in which nodemon was
started, and if any files change, nodemon will automatically
restart your node application.
```

Como verán este es un paquete que vamos a usar en casi todos los proyectos de node que hagamos, por lo tanto es un buen ejemplo de algo que instalariamos globalmente. Para hacerlo hacemos:

```
npm install -g nodemon
```

o en Linux o Mac:

```
sudo npm install -g nodemon
```

Una vez instalado globalmente, puedo simplemente utilizarlo desde la línea de comandos, ya que viene con una interfaz CLI (*command line interface*). De hecho, varios paquetes de npm vienen con una interfaz CLI para ser usados desde la consola o terminal.

En nuestro ejemplo al ejecutar `nodemon` obtenemos:

```
[nodemon] 1.10.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
```

Para probar si funciona, hagamos un cambio en el archivo `index.js` y salvemos. Para mi ejemplo voy a agregar un `console.log()`.

```
var moment = require('moment');
console.log('Salgo en la consola!!');
console.log(moment().format("ddd/MM/YYYY, hA"));
```

```

xterm-256color
File Edit View Search Terminal Tabs Help
xterm-256color xterm-256color
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
^C%
~/D/P/B/N/npm >>> nodemon
[nodemon] 1.10.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Salgo en la consola!!
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart

```

Como vemos, no fue necesario volver a correr el archivo `index.js` a mano para ver los cambios, Nodemon hizo todo el trabajo por nosotros!

Actualizar paquetes

Como dijimos, `npm` también nos sirve para mantener actualizados los paquetes. Para hacerlo sólo tenemos que escribir el siguiente comando:

```
npm update
```

Para mantener la compatibilidad con las aplicaciones, `npm` sólo actualiza automáticamente los *patches* y *minor changes* ([Semantic Versioning](#)) de un paquete, por defecto.

De hecho, el `^` antes del número de versión en 'dependencies' indica qué puede actualizar los *minors* automáticamente, si quisieramos restringir ese comportamiento para que actualice sólo los *patches* deberíamos usar el carácter `~` antes de la versión:

```

"dependencies": {
  "moment": "^2.14.1" //Actualiza sólo Minors
}

"dependencies": {
  "moment": "~2.14.1" //Actualiza sólo Patches
}

```

Leyendo un archivo con Node

Para trabajar con archivos en Node, vamos a usar el módulo `fs` que tiene la funcionalidad para leer y escribir archivos, como se encuentra en las librerías core de node, vamos a requerirlo usando `require('fs')`.

Ahora vamos a crear un archivo llamado `greet.txt` y vamos a escribir un saludo dentro, lo vamos a dejar en la misma carpeta que nuestro archivo `.js`.

Ahora vamos a leer el contenido de ese archivo y mostrarlo por consola:

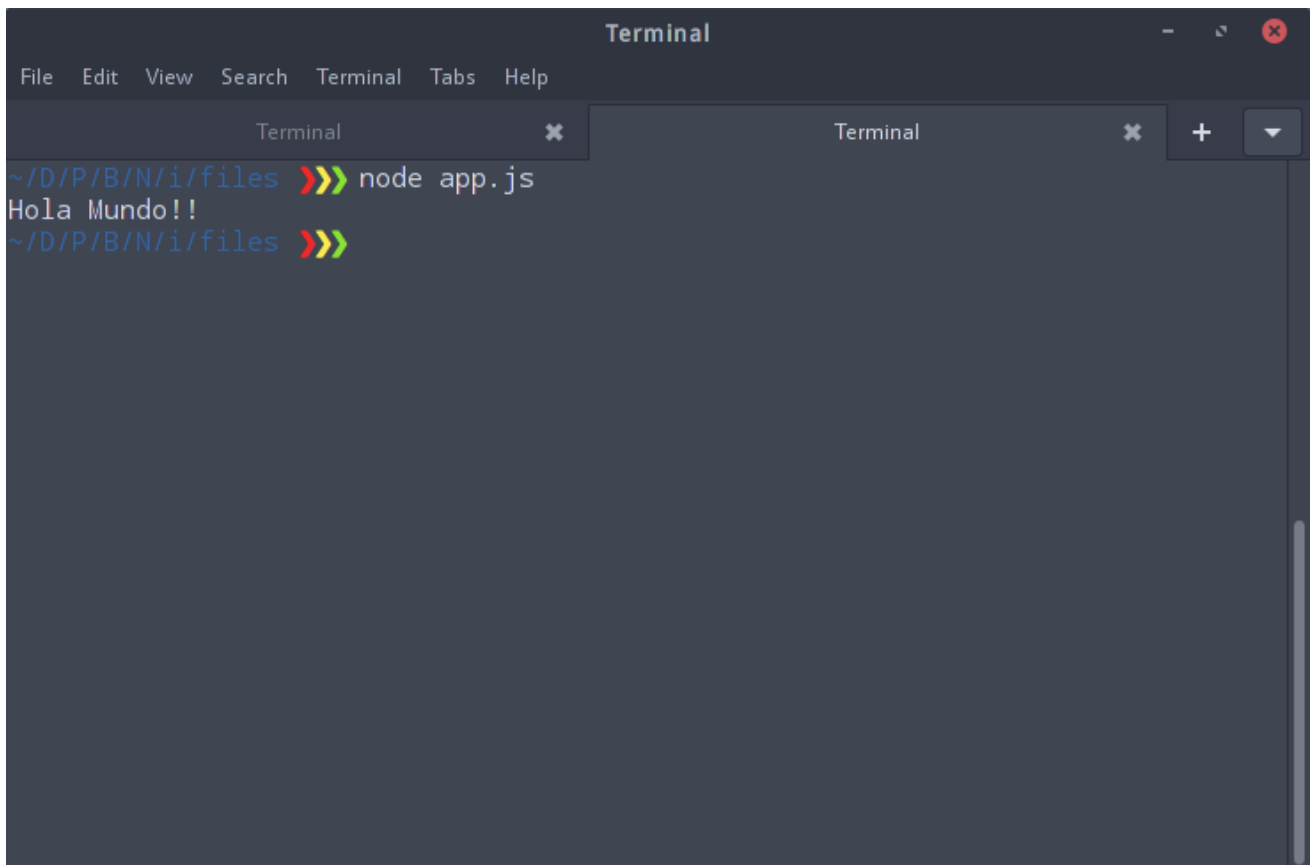
```
var fs = require('fs');

var saludo = fs.readFileSync(__dirname + '/greet.txt', 'utf8');
console.log(saludo);
```

Como podemos ver en la [documentación](#) de `fs`, la función `readFileSync`, recibe un path como parametro y el encoding del file.

__dirname: esta variable tiene guardado el path completo del directorio donde está el archivo que está ejecutando el código

Por lo tanto le estamos diciendo que lea el archivo `greet` que creamos *DE MANERA SINCRONICA* (esto quiere decir que no va avanzar hasta que no lea de manera completa) y que guarde el resultado en la variable `saludo`, luego que muestre el contenido de `saludo` por consola. Como vemos, el resultado es que en `saludo` se guarda el contenido del archivo `greet.txt` !!

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", "Tabs", and "Help". Below the menu bar, there are two tabs, both labeled "Terminal". The first tab is active and shows the following text:

```
~/D/P/B/N/i/files >>> node app.js
Hola Mundo!!
~/D/P/B/N/i/files >>>
```

Al hacerlo sincrónico, el programa se *bloquea* hasta que no termine de leer el archivo completo, si el archivo fuera muy grande, veríamos que el programa queda *trabado* y causa una mala experiencia de uso.

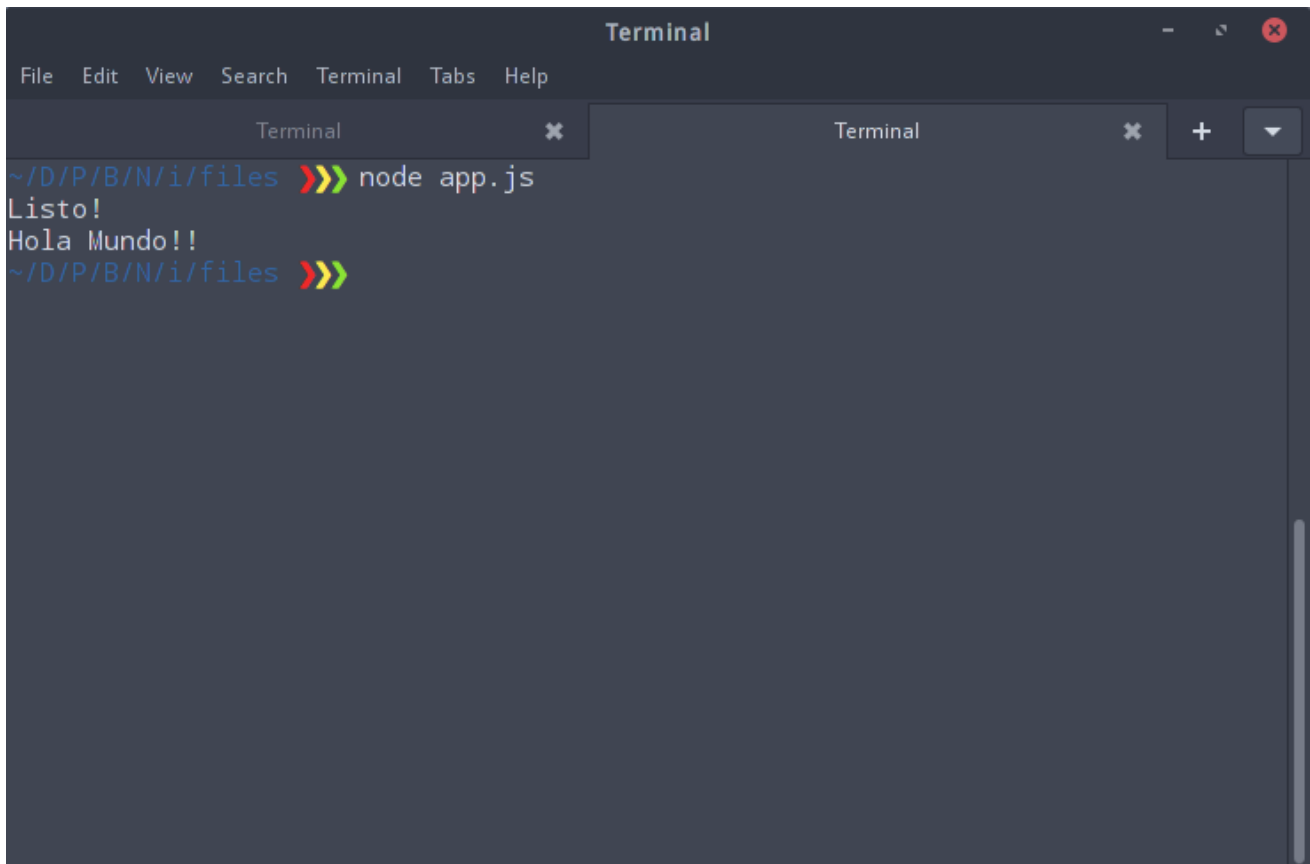
Para resolver ese problema, vamos a hacer lo mismo, pero haciendo que la lectura del archivo sea en forma asíncrona. Para hacerlo, `fs` nos da la función `readFile`, que recibe el path del archivo a leer, el encoding, y además recibe una función, que será el `callback` para cuando se termine de leer el archivo:

```
var fs = require('fs');
fs.readFile(__dirname + '/greet.txt', 'utf8', function(err, data) {
  console.log(data);
});
console.log('Listo!');
```

Hemos agregado al final el `console.log('Listo!')` para que vean en qué momento se ejecuta esa línea de código y en qué momento se ejecuta el `callback`.

Es super importante que comprendan el porqué del orden en el que aparecen los `console.logs`, no avances antes de comprenderlo. Pista: Mirá más arriba como funciona el event loop de JS.

Otra cosa a notar es que la función anónima que le pasamos tiene dos parámetros: *err* y *data*. Esto se debe a que existe un standard llamado **error-first callback** que dice que cada vez que escribas una función que ejecute un callback, le pases a ese *cb* como **primer** parámetro una variable de *Error*. En el caso que no haya habido errores en la ejecución, entonces esa variable tendrá *null*, en caso contrario tendrá algo que informe el error.



```
Terminal
File Edit View Search Terminal Tabs Help
~/D/P/B/N/i/files >>> node app.js
Listo!
Hola Mundo!!
~/D/P/B/N/i/files >>>
```

Como vemos, primero se ejecutó el segundo console log, luego el console log del *callback*.

Combinando Todo

Bien! Ahora estamos listos para empezar a crear nuestro propio servidor web usando estos conceptos!

Homework

Completa la tarea descrita en el archivo [README](#)