Introducción a Node.js

El lenguaje de programación **Javascript** fue diseñado inicialmente para ser ejecutado en navegadores web, con el objetivo de ofrecer la posibilidad de dinamizar las páginas web en el lado del cliente.

Node.js es un entorno de ejecución diseñado para poder interpretar código Javascript fuera del navegador y en una gran variedad de plataformas, como por ejemplo Windows, Linux, OS X, Arduino, etc.

Está basado en el motor Javascript V8 (https://developers.google.com/v8/) creado por Google para su navegador Chrome, y como veremos más adelante las funciones de **entrada/salida** (I/O) que ofrece, como puede ser tratar con ficheros, escribir por pantalla, leer la entrada, comunicación via protocolos web, etc. son de ejecución **asíncrona**, es decir, la ejecución no se bloquea al realizar la llamada, se sigue ejecutando código y el resultado se recibe mediante una función *callback*. Es por eso que se dice que ofrece una **arquitectura orientada a eventos**. Además, toda la ejecución se realiza en un **único thread** o hilo de ejecución, algo que de primeras puede parecer negativo para el rendimiento, pero que ha demostrado ser más efectivo en la mayoría de escenarios.

Módulos en Node.js

Dado que Node.js está pensado para el desarrollo de aplicaciones, tal y como lo podríamos hacer con C++ o Java, y dado que Javascript no ofrece sintaxis para la gestión de módulos, necesitamos algún sistema para poder organizar nuestras aplicaciones en módulos reutilizables y de fácil mantenibilidad. Aquí es donde node.js nos ofrece 5 herramientas.

Gestor de paquetes/módulos `npm`

NPM Es un programa que se ejecuta desde el terminal y que nos sirve para instalar, actualizar y desinstalar nuevos paquetes (considerad paquete = módulo), ya sea a nivel global (de sistema operativo) o a nivel local (para su uso en un módulo concreto). Los comandos básicos que ofrece son:

- `npm install <nombre_paquete>`. Instala un paquete de forma local en la carpeta X en la que nos encontremos. Los paquetes locales se guardan en una carpeta `node_modules` dentro de X, y sólo pueden ser utilizados por ficheros js que estén en X o en alguna de sus subcarpetas.
- `npm install -g <nombre_paquete>`. Instala un paquete de forma global. Los paquetes globales se instalan en un directorio determinado que depende del sistema operativo, y se pueden usar desde cualquier directorio. A veces estos módulos que instalamos funcionan como una aplicación y también ofrecen una interfície por terminal, es decir, se pueden ejecutar por terminal como cualquier otro programa que instalemos.

La carpeta `node_modules`

En esta carpeta se guardan los módulos que instalamos mediante el comando `npm install <nombre_módulo>`. Por ejemplo, si estando en la carpeta X ejecutamos el comando `npm install modulo-guay`, el módulo `modulo-guay` se instalará en el directorio

X/node_modules/modulo-guay. Esto no se cumple para módulos instalados de forma global con la opción -g (npm install -g modulo-guay), ya que los módulos globales se instalan en una carpeta específica del sistema que no depende de la carpeta en la que nos encontremos.

La variable module.exports y la función require

Si queremos poder construir módulos de javascript reutilizables necesitamos de algún mecanismo para **especificar** que **funciones/constantes/valores** de nuestro código queremos **que se puedan ver desde el exterior**, i.e, desde el punto de vista del programador que utiliza nuestro módulo.

Para eso en node se define la variable **module.exports** como la contenedora de todo **aquello que queramos que el módulo ofrezca**. Más adelante veremos un ejemplo de esto, pero antes, **¿como podemos 'obtener' lo que un modulo ofrece?**, i.e ¿Cómo podemos hacer un #include (C++) o import (Java/Python) en Node? Aquí viene la **función require**. require es una función que recibe como parámetro un string, que puede ser:

- En caso de ser un módulo predeterminado de node.js o un módulo que hayamos instalado con npm install modulo-guay (ya sea con opción -g o no, eso no importa), el string que le pasaremos como parámetro a la llamada a require será el **nombre del módulo en cuestión**. Si por ejemplo utilizamos el modulo-guay, para importarlo en nuestro código escribiríamos:

```
var mi_modulo_guay = require('modulo-guay');
```

- En caso de ser un módulo que hayamos creado nosotros, es decir, un fichero .js de nuestra aplicación, especificaremos el **path relativo** del fichero js del módulo que queramos incluir. Si está en la misma carpeta hay que indicar obligatoriamente el path empezando con ./, ya que si no lo ponemos node interpreta que es un nombre de módulo y no un path. Por ejemplo, si en un directorio tenemos el fichero A.js, y la carpeta mi-carpeta que contiene el fichero B.js, para importar el módulo B desde A escribiríamos en A:

```
var B = require('./mi-carpeta/B')
```

Y para importar el módulo A desde B escribiríamos en B:

```
var A = require('../A')
```

Notad que **no es necesario especificar la extensión del fichero**. La función require automáticamente detecta y trata adecuadamente ficheros con extensión .js (código Javascript) y .json formato JSON (Javascript Object Notation).

En ambos casos, en la variable a la que asignamos el resultado de la llamada a require tendremos el valor de la variable module.exports que haya asignado el programador del módulo que importamos. Por otro lado, qué pasa si en vez indicar el path de un fichero indicamos el de una carpeta? Entonces la llamada a require será equivalente a incluir el fichero index.js que haya en esa carpeta.

Volviendo a la variable module.exports, tenemos que tener en cuenta que node siempre ejecuta la siguiente línea de código justo antes de empezar a ejecutar nuestro código javascript:

```
exports = module.exports = {}
```

Por lo tanto, `module.exports` es por defecto un objeto, y tiene una variable *alias* llamada `exports`, pensada simplemente para que no haya que escribir repetidamente un nombre tan largo como `module.exports`. Veamos algunos ejemplos de cómo podemos usar estas dos variables para exponer funciones y valores de nuestros módulos/ficheros javascript:

```
// Esta variable no se podrá ver desde fuera del módulo
var variableNoExportada = 'secret'

module.exports = {
  diHola: function() {
    console.log("Hola!");
  },
  letrasAbecedario: 27,
  meVanAVerDesdeFuera: true
}
```

que es equivalente a:

```
var variableNoExportada = 'secret'

module.exports.diHola = function() {
                          console.log("Hola!");
                        };

module.exports.letrasAbecedario = 27,
module.exports.meVanAVerDesdeFuera = true
```

o a:

```
var variableNoExportada = 'secret'

exports.diHola = function() {
                          console.log("Hola!");
                        };

exports.letrasAbecedario = 27,
exports.meVanAVerDesdeFuera = true
```

Hay que tener en cuenta, sin embargo, que el siguiente código **NO** es equivalente:

```
var variableNoExportada = 'secret'

exports = {
  diHola: function() {
    console.log("Hola!");
  },
  letrasAbecedario: 27,
  meVanAVerDesdeFuera: true
}
```

Fijaos que en este caso la variable que estamos asignando es `exports`. Pero `exports`, como hemos visto antes, simplemente era una variable que apuntava a la misma dirección que `module.exports`, que es la variable que realmente usa node para saber que es lo que

queremos exportar. Si ahora a esta variable `exports` le asignamos otro objeto, entonces ya no apuntará a `module.exports` y será como cualquier otra variable normal privada de nuestro código, por lo tanto no vamos a exportar nada más que `{}`, que es el valor por defecto que le asigna node a `module.exports`.

Notad también que lo que exportemos no tiene porque ser un objeto de javascript, puede ser cualquier valor posible, una función, un valor constante, etc. pero normalmente se usa un objeto para poder exportar más de un valor o función.

Ahora, con cualquiera de los códigos anteriores, y suponiendo que están en un fichero llamado `salutacion.js`, si queremos importar el módulo `salutacion.js` desde otro fichero `main.js` que esté en el mismo directorio, haríamos:

```
// Si especificamos la extensión .js también funciona, pero no es necesario
var modulosalutacion = require('./salutacion');

modulosalutacion.diHola(); // Escribe por consola "Hola!"

// Escribe por consola "27"
if (moduloSalutacion.meVanAVerDesdeFuera)
console.log(modulosalutacion.letrasAbecedario);

// Da ReferenceError 'modulosalutacion.variableNoExportada' is not defined
console.log(modulosalutacion.variableNoExportada);
```

Si sólo hubiesemos necesitado la función `diHola` del módulo, podríamos haber hecho:

```
var diHola = require('./salutacion').diHola;

diHola(); // Escribe por consola "Hola!"
```

El fichero `package.json`

El fichero `package.json` es un fichero que contiene la **información sobre nuestro proyecto/módulo/aplicación**, como su nombre, versión, autor, etc... Y la información más importante, las dependencias.

```
{
  "name": "example",
  "version": "0.0.1",
  "description": "first example",
  "dependencies": {
    "body-parser": "^1.12.2",
    "express": "^4.12.3",
    "winston": "^0.9.0"
  }
}
```

Las dependencias son los módulos de los que depende la aplicación o módulo que esté especificando el `package.json`. En el objeto de dependencias que contiene el fichero `package.json`, podemos ver todos los módulos que queremos que nuestro programa tenga, con su versión corresponiente. Como ya hemos dicho anteriormente, estos módulos estarán en la carpeta `node_modules`.

El fichero package.json es muy útil por ejemplo cuando tenemos que subir nuestro proyecto a Github y no queremos subir la carpeta node-modules, para que el proyecto no pese tanto. Cuando alguien se descargue nuestro proyecto, simplemente tendrá que ejecutar el comando `npm installen` el terminal, y automáticamente se instalarán todos los módulos que hay indicados en el objeto de dependencias del `package.json` con la versión que este indique.

En el caso de que queramos instalar un nuevo módulo anuestro proyecto/aplicación/módulo, y además añadirlo como dependencia de éste en el package.json, simplemente tendremos que usar el comando

```
npm install <modulo que queremos> --save
```

situados en la carpeta de nuestro proyecto. Automáticamente se agregará el módulo como dependencia en el objeto de dependencias del fichero `package.json`, y se instalará en la carpeta `node_modules`, listo para ser incluido desde cualquier fichero javascript de nuestra aplicación con `var modulo = require('nombre-modulo');`.

Para poder crear fácilmente un fichero `package.json` en la carpeta que os encontréis, podéis ejecutar el comando `npm init`, que os hará ciertas preguntas sobre vuestro proyecto y irá rellenando el fichero `package.json` con la información.

Si queréis saber el significado de cada uno de los atributos del fichero package.json, en [este (http://browsenpm.org/package.json)](http://browsenpm.org/package.json) enlace tenéis la explicación de cada atributo y un listado de todos los atributos disponibles.

Ejemplo

En este ejemplo podréis encontrar un programa `gestor.js` que escribe por pantalla el contenido de la carpeta ficheros (todos sus archivos) y que escribe su propio código.

Para poder usarlo deberéis ejecutar `npm install` para instalar sus dependencias, y luego ejecutar `node gestor.js` para ejecutar el programa.

Como podréis ver, el programa también usa un módulo que hemos definido nosotros, y que está en la carpeta `gestor_ficheros`. Este módulo es el que implementa las funciones para leer un fichero dado su camino y para listar los fichero de una carpeta. Además, usa un módulo que lleva por defecto node y que sirve para gestionar ficheros. Se llama `fs` (filesystem) y lo utilizamos para implementar las funciones del módulo `gestor_ficheros`.