Tema 1

Introducción a Node.js

¿Qué es Node.js?

- Node.js, como lo definen sus creadores, es “(…) una plataforma basada en Chrome JavaScript Runtime para crear aplicaciones web de manera fácil, rápida y escalable"

- Node.js utiliza un modelo de no-bloqueo orientado a eventos, que lo hace ligero y eficiente, ideal para aplicaciones de data intensiva en tiempo real que se ejecutan a través de dispositivos distribuidos.

¿Qué es Node.js?

- Se programa del lado del servidor, lo que indica que los procesos para el desarrollo de software se realizan de una manera muy diferente que los de Javascript del lado del cliente.

- Cuando trabajamos con Node.js solamente necesitamos preocuparnos de que el código se ejecute correctamente en tu servidor.

¿Qué es Node.js?

- Otras de las cosas a tener en cuenta cuando trabajas con Node.js es la programación asíncrona y la programación orientada a eventos.

- Con la particularidad que los eventos en esta plataforma son orientados a cosas que suceden del lado del servidor y no del lado del cliente como estamos habituados en Javascript.

- Ya no vamos a responder sobre eventos provocados por parte del usuario al interactuar con los elementos de nuestra web, si no sobre eventos lanzados a partir de sucesos que ocurran en nuestro servidor.

¿Qué es Node.js?

- Además, NodeJS implementa los protocolos de comunicaciones en redes más habituales, de los usados en Internet, como puede ser el HTTP, DNS, TLS, SSL, etc.

- Mención especial al protocolo SPDY, fácilmente implementado en Node, que ha sido desarrollado mayoritariamente por Google y que pretende modernizar el protocolo HTTP, creando un sistema de comunicaciones que es sensiblemente más rápido que el antiguo HTTP (estiman un rendimiento 64% superior).

¿Quién usa NodeJS?

- Importantes empresas han migrado sus servidores a Node.js, quizá el caso más llamativo fue el de **LinkedIn**:

- A finales de 2012 LinkedIn **cambió Rails por Node.js**, lo que le permitió pasar de 30 servidores a 3, y comportarse, en determinados escenarios, hasta 20 veces más rápido.

- Además de LinkedIn, eBay, Microsoft, empresas dedicadas a hosting como Nodester o Nodejitsu, redes sociales como Geekli.st, y muchos más tienen sus servidores en Node.js.

Características

- Programación asíncrona:
  - Toma especial importancia, dado que NodeJS fue pensado desde el primer momento para potenciar los beneficios de la programación asíncrona.
  - En la programación asíncrona eres capaz de crear diferentes hilos, con diferentes procesos que llevarán un tiempo en ejecutarse, de modo que se hagan todos a la vez. Además, podrás especificar código (*callbacks*) que se ejecute al final de cada uno de esos procesos largos.

Características

- Programación asíncrona:
  - La filosofía detrás de Node.js es hacer programas que no bloqueen la línea de ejecución de código con respecto a entradas y salidas, de modo que los ciclos de procesamiento se queden disponibles cuando se está esperando a que se completen tales acciones.
  - Realmente Javascript es síncrono y ejecuta las líneas de código una detrás de otra, pero por la forma de ejecutarse el código hace posible la programación asíncrona.

Características

- Programación asíncrona:
  - Un ejemplo del funcionamiento asíncrono de Node.js sería el siguiente:

```
console.log("inicio");
fs.readFile("x.txt", function(error, archivo){
   console.log("archivo leído");
})
console.log("final");
```

  - La segunda instrucción (que hace la lectura del archivo) tardará un rato en ejecutarse y en ella indicamos además una función con un console.log ("archivo leído"), esa es la función callback que se ejecutará solamente cuando termine la lectura del archivo.

  - Como resultado, primero veremos el mensaje "inicio" en la consola, luego el mensaje "final" y por último, cuando el fichero terminó su lectura, veremos el mensaje "archivo leído".

Características

- Problema del código piramidal:
  - El uso intensivo de callbacks en la programación asíncrona produce el poco deseable efecto de código piramidal.
  - Al utilizarse los callbacks, se meten unas funciones dentro de otras y se va entrando en niveles de profundidad que hacen un código menos sencillo de entender visualmente y de mantener.

Características

- Problema del código piramidal:
  - La solución es hacer un esfuerzo adicional por estructurar nuestro código. Básicamente se trata de **modularizar** el código de cada una de las funciones, escribiéndolas aparte e indicando el nombre de la función callback, en vez de escribir el código.
  - Sería conveniente incluso definir las funciones en archivos aparte y requiriéndolas con **require("nombre_archivo")** en el código de tu aplicación.

Características

- Problema del código piramidal:
  - Al conseguir niveles de indentación menos profundos, estamos ordenando el código, con lo que será más sencillo de entender y mantener.
  - Algunos consejos a la hora de escribir código para que éste sea de mayor calidad:
    - Escribe código modularizado (un archivo con más de 500 líneas de código puede que esté mal planteado)
    - No abuses, no repitas las mismas cosas, mejor diseña funciones reusables.
    - Usa librerías que ayuden al control (como async que te ayuda a ordenar callbacks)

Módulos

- En Node.js el código se organiza por medio de módulos. Son como los paquetes o librerías de otros lenguajes como Java. Por su parte, **NPM** es el nombre del gestor de paquetes (*package manager*) que usamos en Node.js.

- El gestor de paquetes npm es un poco distinto a otros gestores de paquetes que podemos conocer, porque los instala localmente en los proyectos. Es decir, al descargarse un módulo, se agrega a un proyecto local, que es el que lo tendrá disponible para incluir.

- Aunque cabe decir que también existe la posibilidad de instalar los paquetes de manera global en nuestro sistema.

Módulos

- Javascript nativo no da soporte a los módulos. Esto es algo que se ha agregado en NodeJS y se realiza con la sentencia **require()**, que está inspirada en la variante propuesta por CommonJS.

- La instrucción require() recibe como parámetro el nombre del paquete que queremos incluir e inicia una búsqueda en el sistema de archivos, en la carpeta "node_modules" y sus hijos, que contienen todos los módulos que podrían ser requeridos.

- Por ejemplo, si deseamos traernos la librería para hacer un servidor web, que escuche solicitudes http, haríamos lo siguiente:

```
var http = require("http");
```

Módulos

- Existen distintos módulos que están disponibles de manera predeterminada en cualquier proyecto Node.js y que por tanto no necesitamos instalar previamente con npm.

- Éstos toman el nombre de "**Módulos nativos**" y ejemplos de ellos tenemos:

  - **http** para atender solicitudes HTTP.
  - **fs** para el acceso al sistema de archivos.
  - **net** es un módulo para conexiones de red de más bajo nivel.
  - **url** permite realizar operaciones sobre url.
  - **util** es un conjunto de utilidades.
  - **child_process** da herramientas para ejecutar sobre el sistema.
  - **domain** permite manejar errores.

Módulos

- Podemos escribir nuestros propios módulos, para ello:

1. Escribimos el módulo con nuestras funciones:

```
function suma(a,b){
   return a + b;
}

function multiplicar(a,b){
   return a * b;
}
```

2. Exportamos nuestras funciones con module.exports:

```
module.exports = {
    suma: suma,
   multiplicar: multiplicar
}
```

3. Suponiendo que el archivo con las operaciones se llama *operaciones.js* desde otro fichero podríamos acceder a ellas:

```
var operaciones = require('./operaciones');
operaciones.suma(2,3);
```

Módulos

- Volviendo a *npm*, es una operación que funciona desde la línea de comandos de Node.js. Por tanto lo tenemos que invocar con *npm* seguido de la operación que queramos realizar.

```
npm install async
```

- Esto instalará el paquete *async* dentro de mi proyecto. Lo instalará dentro de la carpeta *node_modules* y a partir de ese momento estará disponible en mi proyecto y podré incluirlo por medio de *require*:

```
require("async");
```

Módulos

- Npm instala los paquetes para un proyecto en concreto, sin embargo existen muchos paquetes de Node.js que te facilitan tareas relacionadas con el sistema operativo

- Estos paquetes, una vez instalados, se convierten en comandos disponibles en terminal. Existen cada vez más módulos de Node.js que nos ofrecen muchas utilidades, accesibles por línea de comandos, como Bower, Grunt, etc.

- Las instrucciones para la instalación de paquetes de manera global son prácticamente las mismas que para la instalación de paquetes en proyectos pero ene este caso utilizando *-g*

```
npm install -g grunt-cli
```

Ejemplo de Node.js

- Para crear un servidor web que trabaje con el protocolo HTTP deberíamos hacer lo siguiente:
  - Para usar el módulo http:

```
var http = require("http");
```

  - Ahora tenemos una variable http que es un objeto, sobre el que podemos invocar métodos que están en el módulo requerido. Uno de los requisitos en un servidor HTTP es recibir peticiones:

```
var server = http.createServer(function (peticion, respuesta){
   respuesta.end("He recibido una petición");
});
```

  - La función *callback* que enviamos a *createServer()* recibe dos parámetros que son la petición y la respuesta. La petición contiene datos de la petición realizada. La respuesta la usaremos para enviarle datos al cliente que hizo la petición. De modo que *respuesta.end()* sirve para terminar la petición y enviar los datos al cliente.

Ejemplo de Node.js

- Es momento de decirle al servidor que se ponga en marcha. Porque hasta el momento solo hemos creado el servidor y escrito el código a ejecutar cuando se produzca una petición, pero no lo hemos iniciado:

```
server.listen(3000, function(){
   console.log("tu servidor está listo en " + this.address().port);
});
```

- Con esto el servidor escucha en el puerto 3000, aunque podríamos haber puesto cualquier otro.

Ejemplo de Node.js

- Este sería el código completo del servidor:

```
var http = require("http");
var server = http.createServer(function (peticion, respuesta){
   respuesta.end("He recibido una petición");
});
server.listen(3000, function(){
   console.log("tu servidor está listo en " + this.address().port);
});
```

- Debemos guardar ese archivo en cualquier lugar con extensión *.js*, por ejemplo servidor.js.

Ejemplo de Node.js

- Para ejecutar el servidor únicamente debemos acceder con la terminal a la carpeta en la que se encuentre y ejecutar:

```
node servidor.js
```

- En consola debe aparecer el mensaje que informa que nuestro servidor está escuchando en el puerto 3000.

- Para comprobar si realmente el servidor está escuchando a solicitudes de clientes en dicho puerto podemos abrir el navegador y acceder a: **http://localhost:3000**