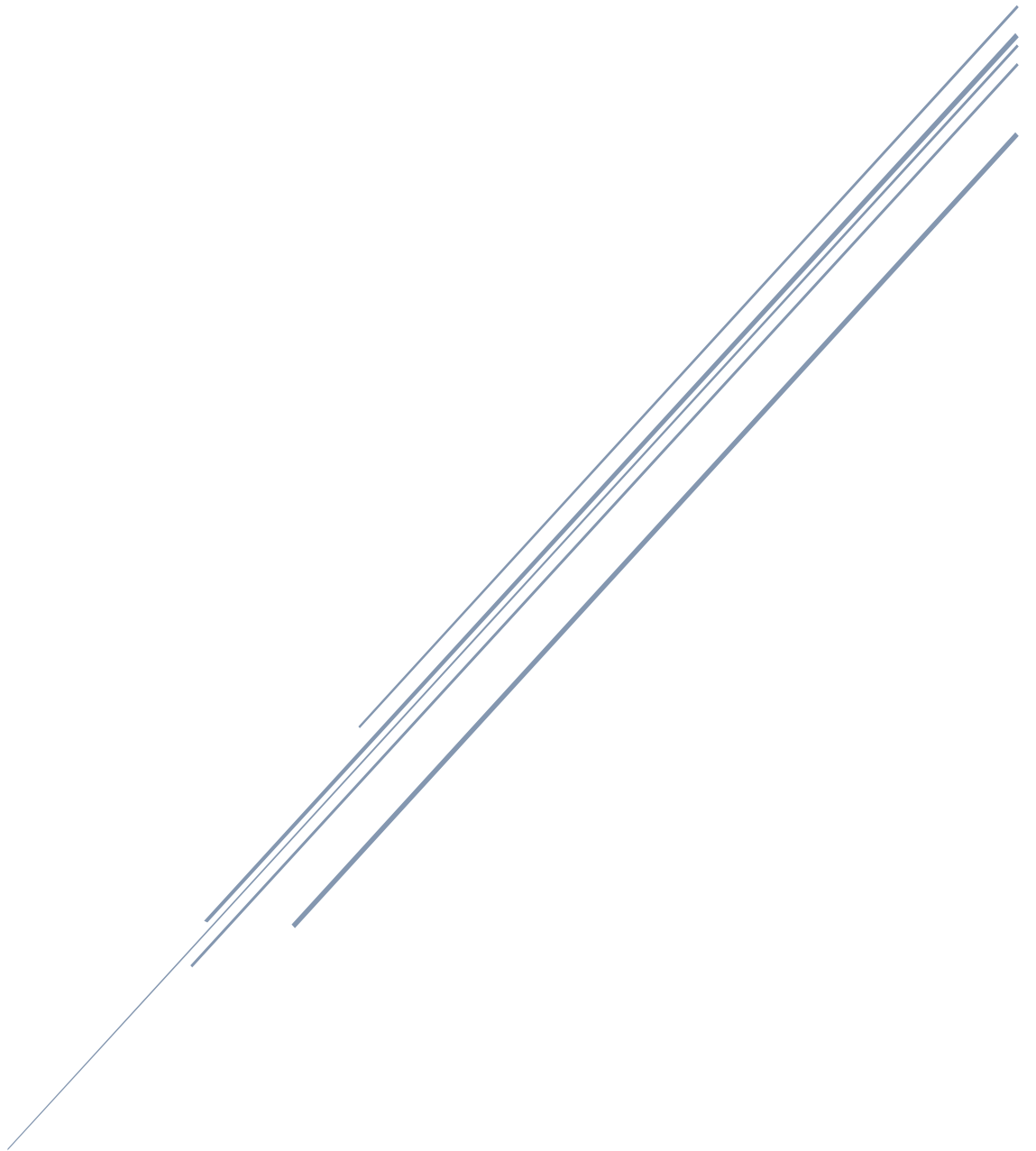


JAVASCRIPT EN SERVIDOR CON NODE.JS



Isabel María Ariza Velasco
Noelia Hinojosa Sánchez

Contenido

1. Introducción.....	2
2. ¿Qué es Node.js?	2
3. Requisitos de la tecnología.....	3
3.1. Instalación en Windows.....	3
.....	4
3.2. Instalación en Linux.	5
3.3. Instalación en Mac.....	5
4. Aplicabilidad y Uso.	5
5. Ventajas y Desventajas.....	6
6. Introducción a Node.js.	6
6.1. El concepto de módulo.	10
6.2. El concepto de Evento	13
6.2.1. Como definir un evento en Node.js.....	13
6.3. Crear Nuestro Propio Servidor HTTP Básico.	14
6.3.1. Peticiones POST	23
6.4. Usando Node.js y NPM.	28
7. Bibliografía.....	30

1. Introducción.

En la actualidad, existen numerosos lenguajes de programación utilizados para la creación de programas de servidor, como podrían ser Java o PHP, a nivel de programación Web.

A pesar de ser lenguajes enfocados al desarrollo de servidor, son totalmente opuestos. Por ejemplo, mientras que Java es estático, corre en múltiples núcleos de mundo transparente, Node es un lenguaje dinámico y directo, que ejecuta un único hilo sin costes de cambio de contexto.

En este tutorial, vamos a centrarnos en conocer Qué es Node.js, el uso de JavaScript en servidor con Node.js, que se necesita para su uso, para que tipo de aplicaciones es útil y veremos algunos ejemplos de su desarrollo.

2. ¿Qué es Node.js?

Por lo que hemos podido ver, encontrar una definición clara de lo que es Node.js es algo bastante difícil y tedioso, pues hemos comprobado que en ninguna parte se da una explicación corta y concisa de lo que realmente es, para que sea fácil de entender o analizar.

Veamos primero la definición que nos da Wikipedia:

“Es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación ECMAScript (lenguaje basado en JavaScript), asíncrono, con entrada/salida de datos en una arquitectura orientada a eventos, y basado en el motor V8 de Google Chrome, un motor de código abierto de JavaScript en C++ que compila código máquina en lugar de ejecutarlo en tiempo real.”

A decir verdad, con esta explicación no hemos conseguido aclarar demasiado, a no ser que sea un experto en Node.js... lo cual nosotras no somos... aun.

Tal vez, una forma más sencilla de entender que es sea viendo cuáles son sus características, ¿no?

Pues bien, profundicemos un poco en el uso de JavaScript en el lado del servidor.

Para empezar, todos sabemos que para que dos dispositivos puedan comunicarse, ya sean ordenadores, móviles, tablets, etc, se necesita una máquina que actúe como servidor y varios clientes que deseen comunicarse.

Como funciona la comunicación entre clientes y servidor es sencilla... Los clientes se comunican con el servidor y este les devuelve la información que han solicitado, ya sea desde un sitio web, o una base de datos, por ejemplo.

JavaScript es un lenguaje que se ejecuta en el lado del cliente, en Web. Una de las principales características de Node.js es que hace que este lenguaje de programación pueda estar del lado del cliente, a estar ejecutando en el lado del servidor.

Otra de las características de Node.js es que soporta una gran cantidad de conexiones simultáneas a un servidor. Emplea un único hilo y un bucle de eventos asíncronos, es decir, permite que sucedan múltiples cosas al mismo tiempo. Por este motivo, Node.js es apropiado para desarrollo y aplicaciones con un gran número de conexión simultáneas y por el contrario, no resulta adecuado para aplicaciones que requieran un número reducido de conexiones con un gran consumo de recursos como aplicaciones de cálculo, acceso intensivo de datos, etc.

Node.js también tiene un gestor de paquetes NPM (Node Package Manager) que nos permite acceder a una gran cantidad de librerías de Código abierto.

Aunque Node.js no es ni un servidor web ni un entorno de desarrollo, se ha convertido en una de las herramientas estándares en el desarrollo Web por su rapidez en ejecutar tareas, sincronización de cambios en el navegador en tiempo real, etc.

En conclusión, ahora podemos decir que Node.js es un entorno JavaScript del lado del servidor, basado en eventos, que complica y ejecuta a velocidades increíbles

3. Requisitos de la tecnología.

La instalación de Node.js es un proceso bastante sencillo, pues no se necesitan unos requisitos exactos para poder usarlos.

Para su instalación Windows, tan solo debemos descargar la última versión en el enlace que se proporcionara continuación.

Para Linux, necesitamos una cuenta de usuario independiente que no sea root, con privilegios sudo.

Para Mac, necesitamos tener instalado Hombrew, ya que de esta forma, la instalación de Node.js es más sencilla.

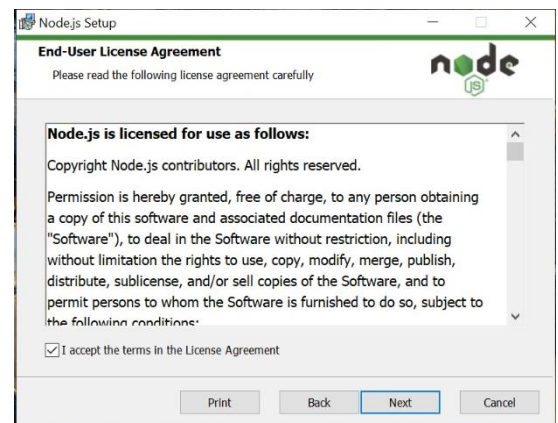
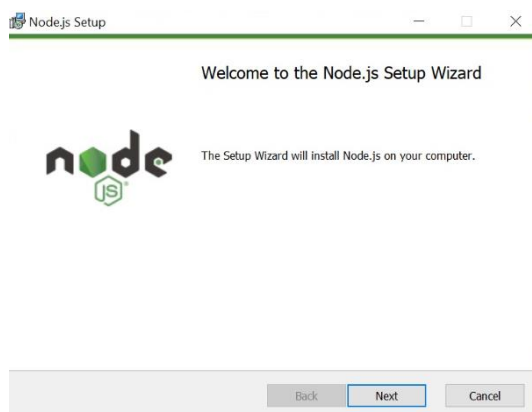
A continuación, veremos cómo realizar la instalación en diferentes sistemas operativos.

3.1. Instalación en Windows.

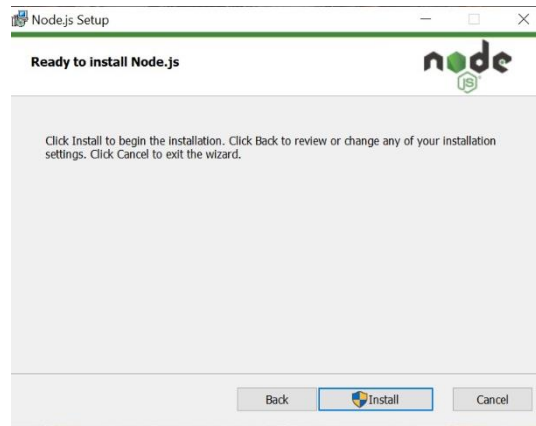
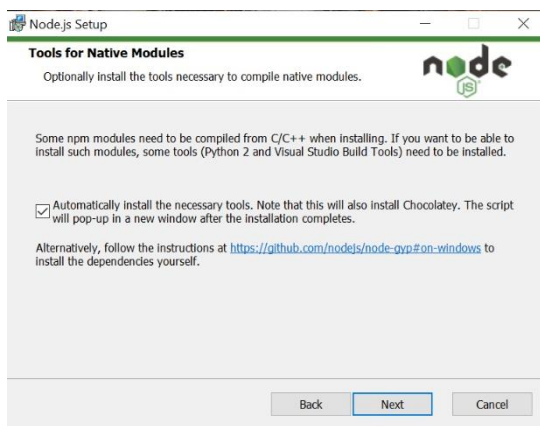
Lo primero que tenemos que hacer es dirigirnos al enlace siguiente: <https://nodejs.org/es/> donde directamente encontrarás la versión adecuada según tu sistema operativo. Una vez descargada, hacemos doble clic sobre el paquete, y comenzamos con su instalación.

Tan solo tenemos que seguir los pasos mostrados en las imágenes siguientes.

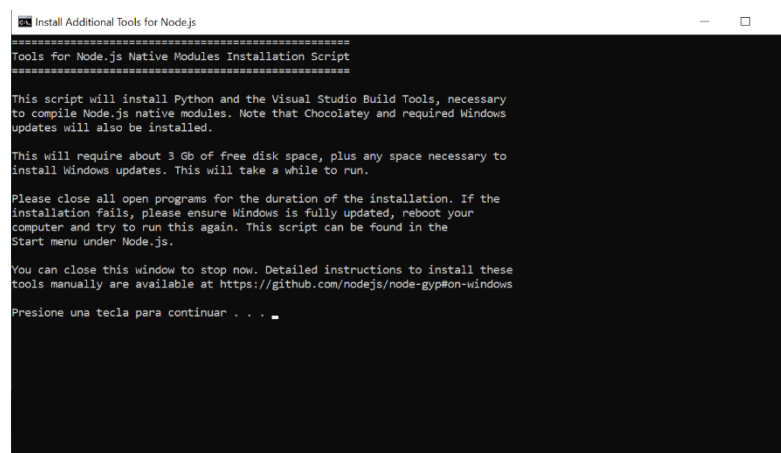
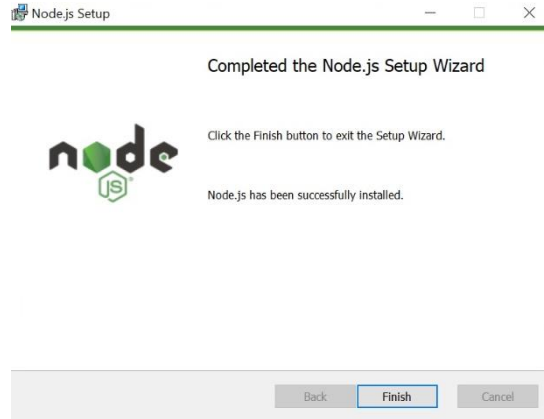
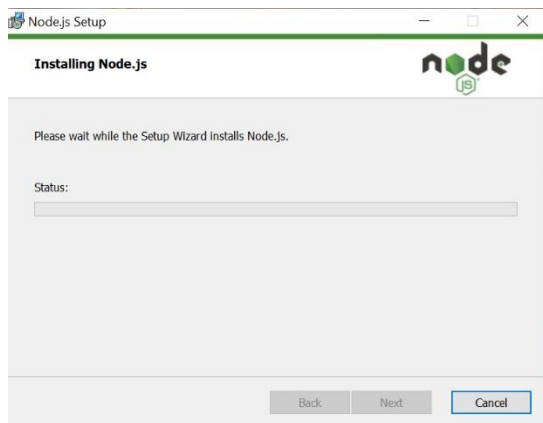
Hacemos "Next" en la primera ventana que nos muestra y aceptamos la licencia en la segunda.



Elegimos la ruta donde queremos que se instale y marcamos los ajustes, en este caso se han elegido para no tener que modificar las variables de entorno de Windows.



Y continuamos hasta finalizar la instalación.



3.2. Instalación en Linux.

La instalación en Linux es más corta y sencilla que la de Windows, pues solo debemos ejecutar los siguientes comandos desde la terminal.

```
➤ sudo apt-get install curl  
➤ curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -  
➤ sudo apt-get install nodejs  
➤ node -v  
➤ npm -v
```

3.3. Instalación en Mac

Una forma de instalar Node.js en Mac es haciendo uso de Homebrew desde la terminal usando los siguientes comandos:

```
➤ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Una vez tenemos Homebrew instalado, pasamos a Node.js:

```
➤ brew install node
```

Y para comprobar que la instalación se ha realizado de forma correcta:

```
➤ node -v ó npm -v
```

4. Aplicabilidad y Uso.

Los usos principales de Node.js se basan en sistemas que respondan en tiempo real de ejecución. En este ámbito podemos encontrar diferentes opciones como son:

- La programación de servicios web, tanto en la creación de *APIs* como para páginas web tradicionales, y con tradicionales me refiero a aquellas páginas web que no permiten una interacción con los usuarios que la visitan sino que simplemente son páginas informativas.
- Aplicaciones de escritorio como NW y *electron*
- Scripts de administración o monitorización para sistemas informáticos o incluso aplicaciones.
- Además de crear los scripts anteriores, se pueden crear proyectos de ayuda para los desarrolladores, los cuales pueden estar integrados en los flujos de trabajos de desarrollo, que cada vez más empresas de programación usan para cualquier lenguaje.

Un ejemplo de uso es la creación de un chat. Aunque es una aplicación con un tráfico de datos bastante intensivo, es ligera, ya que no se requiere una capacidad de procesamiento alta y funciona en dispositivos distribuidos.

Analizando lo anterior vemos que cubre la mayoría de herramientas que se pueden utilizar en una aplicación con Node.js. Aunque como cualquier aplicación, se puede mejorar su mecanismo añadiendo una única cola de mensajes para la gestión del enrutamiento de mensajes a los clientes y controlar la entrega de mensajes de forma

que se pueda cubrir pérdidas de conexión temporal o almacenar mensajes para clientes registrados que estén desconectados en ese momento.

Otro ejemplo, menos común, es el uso de Node.js con Express.js (web application framework) para crear aplicaciones web clásicas en el servidor. De este modo, si la aplicación no tiene cálculos intensivos en la CPU y no se cuenta con una base relacional no se bloqueará la receptividad de Node.js

5. Ventajas y Desventajas.

Las ventajas principales que tiene Node.js de cara al uso son:

- Mayor velocidad, ya que como se ha dicho varias veces en este trabajo, es en tiempo real de ejecución. - La entrada/salida que se realiza con el servidor no se bloquea.
- En la transmisión de datos, solicitudes y respuestas HTTP, se reconoce como un único evento.
- El usar el mismo lenguaje en la parte del cliente y en el servidor ayuda a una comunicación más fluida.
- Se unifican las consultas de base de datos en el formato JSON.
- Es fácil de codificar.
- Cuenta con una buena gestión de paquetes con NPM.
- Cuenta con la capacidad de servidor proxy, por lo que no es necesario un proxy externo.
- Los ciclos de desarrollo son rápidos.
- El propio servidor tiene la lógica del negocio.

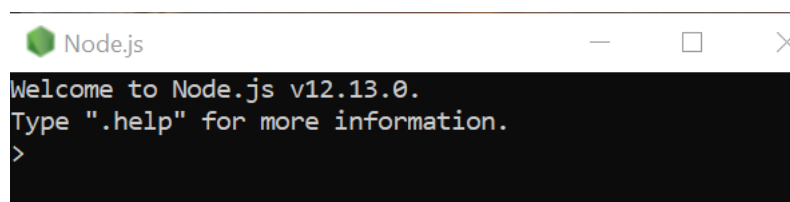
Las principales desventajas son:

- En el caso de que se realice un uso excesivo de CPU, la receptividad de Node.js se bloqueará.
- Es muy difícil trabajar con una base relacional.

6. Introducción a Node.js.

Para comenzar con el tutorial sobre JavaScript con Node.js, en primer lugar veremos cómo podemos hacer uso de Node.js después de haber realizado su instalación.

La aplicación de Node.js se puede abrir de dos formas: la primera, si estamos trabajando desde Windows, como es nuestro caso, podemos ir a nuestra barra de tareas y escribir *node.js*.



La segunda forma, que es la que nosotros vamos a utilizar, es accediendo a través del símbolo del sistema.

En primer lugar, vamos a ver que node funciona como un lector de código. Si escribimos *node* en nuestro cmd, vemos que aparece una "flecha", llamada *prompt*. Lo que esto indica, es que node está esperando o está escuchando a que escribamos nuestro código *JavaScript*.

```
Símbolo del sistema - node
Microsoft Windows [Versión 10.0.18362.476]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\isa_a>node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
>
```

Para comenzar, vamos a ver el funcionamiento de forma simple. Por ejemplo, si queremos ejecutar un pequeño código, como por ejemplo, una suma:

Si escribimos `5 + 3`, y presionamos *enter*, vemos que nos devuelve un 9. Node.js no es una calculadora, pero como cualquier lenguaje de programación, realiza cálculos, por lo que lee la expresión introducida.

La consola de node lee tipos de datos. En el ejemplo anterior le indicamos dos números y una operación. En el caso de querer introducir una palabra o frase,

esta debe ir entre comas simples o entre dobles comillas, lo que indicaría que se trata de un *string* o cadena, puesto que si no, nos lanzaría un error.

```
Símbolo del sistema - node
Microsoft Windows [Versión 10.0.18362.476]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\isa_a>node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
> 5 + 3
8
> 'Hola Mundo'
'Hola Mundo'
> "Hola Mundo"
'Hola Mundo'
> Hola Mundo
Thrown:
Hola Mundo
  ^^^^^
SyntaxError: Unexpected identifier
>
```

Si lo que queremos es concatenar dos palabras o cadenas, es tan simple como añadir entre ambas el símbolo de la suma, puesto que si simplemente indicamos una detrás de la otra también no saltaría un error.


```
Símbolo del sistema - node
C:\Users\isa_a>node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
> "Hola" "Mundo"
Thrown:
"Hola" "Mundo"
^^^^^^
SyntaxError: Unexpected string
> "Hola " + "Mundo"
'Hola Mundo'
>
```

Hasta aquí tan solo hemos visto de una forma muy breve y simple como es el funcionamiento de Node.js, pero ahora vamos a comenzar a explicar su uso junto a *JavaScript*.

Para comenzar, vamos a empezar con la primera sentencia:

```
console.log()
```

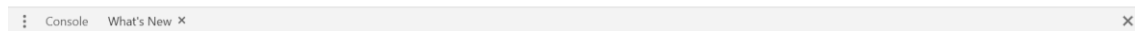
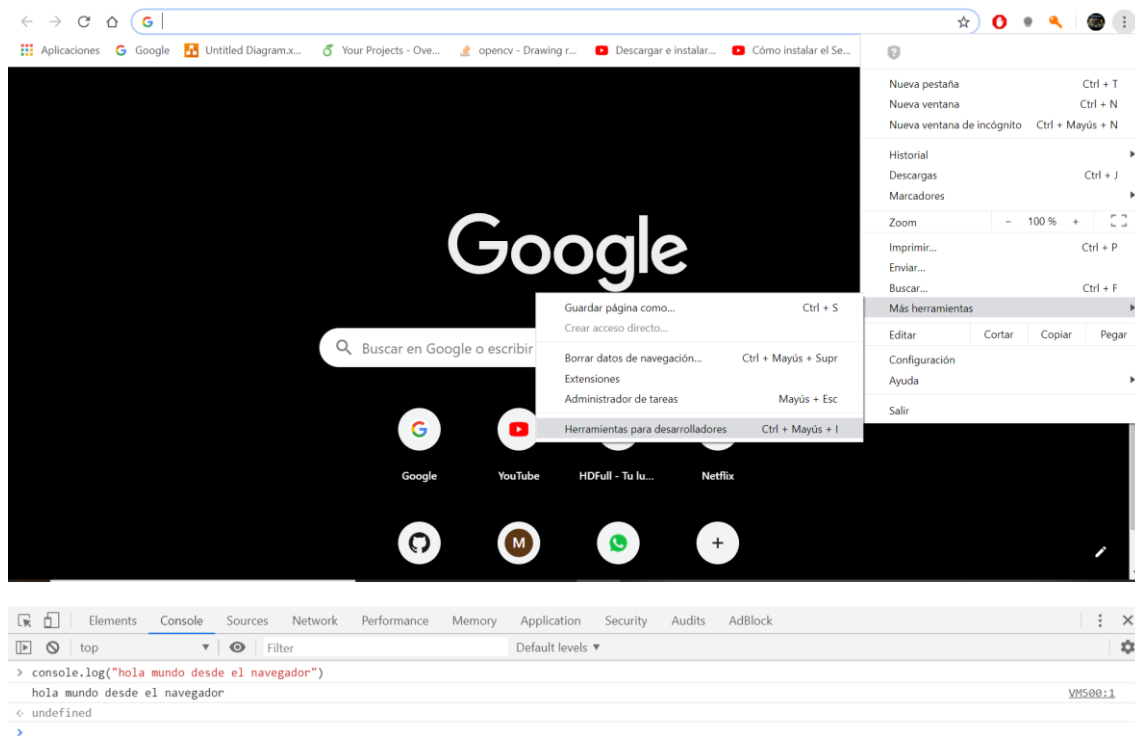
Esta sentencia lo que hace es mostrar por la consola el mensaje que nosotros queremos pasar. Por ejemplo, si escribimos en la cmd,

```
console.log("Hola Mundo")
```

vemos como nos devuelve el mensaje, como vemos, también no aparece la palabra *undefined*, que por ahora no vamos a explicar.

```
Símbolo del sistema - node
C:\Users\isa_a>node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
> console.log("Hola Mundo")
Hola Mundo
undefined
>
```

Si ahora nos vamos a nuestro navegador → Más herramientas → Herramientas del navegador y en Console escribimos la misma sentencia, vemos que obtenemos el mismo resultado:



La diferencia, es que en la cmd, estábamos ejecutando código *JavaScript* desde el servidor, y desde el navegador ejecutamos código *JavaScript*, que solo servirá para ese ámbito. Lo que vamos a ver aquí, es la diferencia entre ambos.

A continuación, en lugar de continuar ejecutando directamente sentencias simples desde la cmd, vamos a trabajar con archivos externos.

Para crear los ficheros, podemos utilizar cualquier tipo de editor, incluso el propio bloc de notas del ordenador. Nosotros vamos a utilizar la aplicación Notepad++. Crearemos un fichero llamado "ejemplo.js", que contendrá el código JavaScript.

Desde la consola, nos dirigiremos a la carpeta donde hemos creado nuestro archivo.

Una vez situados en el directorio correcto, en nuestro "ejemplo.js" vamos a escribir la típica sentencia de "Hola Mundo". Abrimos nuestro archivo y simplemente escribimos:

```
console.log("Hola Mundo");
```

Para ejecutar este fichero desde la consola, tan solo tendremos que escribir *node ejemplo.js* (la extensión se puede omitir).



```
C:\Users\isa_a\Desktop\nodejsTutorial\codigoEjemplo>node ejemplo.js
Hola Mundo
```

Ahora que sabemos cómo ejecutar de forma correcta un fichero *JavaScript* en *node.js*, vamos a profundizar un poco más.

6.1. El concepto de módulo.

Una gran diferencia que encontramos en *JavaScript* con *node.js*, es el concepto de módulo.

Los módulos son simplemente trozos de código, dispuestos en diferentes ficheros, que puede contener funciones, objetos, variables, etc. Esto se usa para tener nuestro código más claro y ordenado en diferentes ficheros.

Para ver esto de forma más clara, vamos a crearnos un módulo llamado *operaciones.js*, que contendrá un programa sencillo que realizará diferentes operaciones y las mostrará por consola.

El código de nuestro modulo será el siguiente:

```
function sumar(x, y){
    return x+y;
}

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

function dividir(x, y){
    if(y==0){
        error();
    }else{
        return x/y;
    }
}

function error(){
    console.log("No se puede dividir por cero");
}

const NUM=4.44;

exports.sumar=sumar;

exports.restar=restar;
```

```
exports.dividir=dividir;  
exports.NUM=NUM;
```

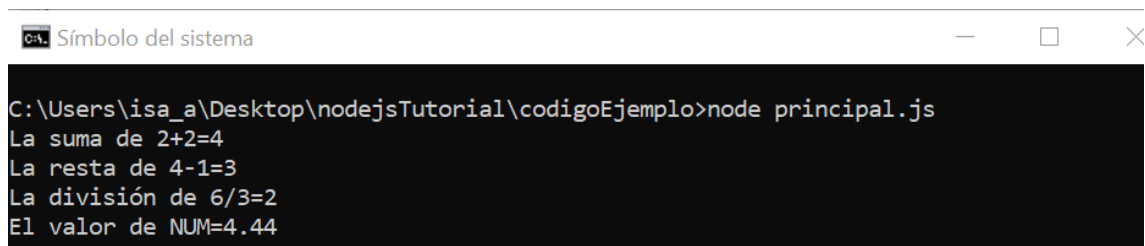
Para poder acceder a las funciones, variables, objetos, etc, definidos en un módulo, desde otro exterior, tenemos que añadirlos al objeto *exports*, que es un objeto que el módulo actual devuelve cuando es "requerido" en otro programa o módulo.

A continuación, crearemos nuestro programa principal, al que llamaremos *principal.js*, y añadiremos el siguiente código;

```
const op= require('./operaciones.js');  
  
console.log('La suma de 2+2='+mat.sumar(2,2));  
console.log('La resta de 4-1='+mat.restar(4,1));  
console.log('La división de 6/3='+mat.dividir(6,3));  
console.log('El valor de NUM='+mat.NUM);
```

Cuando hacemos uso de *JavaScript* desde el navegador, no podemos referenciar a otro archivo, en cambio, en *node.js*, esto puede hacerse usando la función *require* e indicando la dirección del fichero *principal.js*.

Para poder acceder a todas las variables, funciones y objetos exportados, creamos la constante *op*, desde la cual accederemos a todo lo anteriormente enumerado. Ahora, cuando ejecutamos nuestro programa principal en la consola, obtendremos como resultado:



```
C:\Users\isa_a\Desktop\nodejsTutorial\codigoEjemplo>node principal.js  
La suma de 2+2=4  
La resta de 4-1=3  
La división de 6/3=2  
El valor de NUM=4.44
```

Conocer lo que son los módulos, como se crean y como se utilizan es muy útil en *node.js*, que puede llegar a ser muy útil conforma la funcionalidad de un programa aumente su complejidad.

Node.js también tiene módulo del núcleo, que son una serie de módulos de uso común en los programas, como pueden ser: *os*, *fs*, *http*, *url*, *path*, etc.

Vamos a ver algunos de estos módulos propios de *Node.js*:

➤ *Módulo Operative System:*

```
var os= require("os");  
console.log("Sistema Operativo: " + os.platform());  
console.log("Versión del OS: " + os.release());
```

```
console.log("Memoria Total: " + os.totalmem()+"bytes");  
console.log("Memoria Libre: " + os.freemem()+"bytes");
```

```
root@DESKTOP-874LG08:/mnt/c/Users/isa_a/Desktop/nodejsTutorial/cod  
igoEjemplo# node moduloS0.js  
Sistema Operativo: linux  
Versión del OS: 4.4.0-18362-Microsoft  
Memoria Total: 17066909696bytes  
Memoria Libre: 11172364288bytes
```

➤ *Módulo File System:*

Permite acceder al sistema de archivos para leer sus contenidos y crear otros archivos o carpetas.

○ ***writeFile:***

```
var fs= require("fs");  
fs.writeFile("./ejemplo/archivo.txt",  
    "linea 1\nLinea 2",  
    function(error){  
        if(error) console.log(error);  
        else console.log("El archivo fue creado");  
    });  
console.log("Ultima linea del programa");
```

○ ***readFile:***

```
function leer(error, datos){  
    if(error) console.log(error);  
    else console.log(datos.toString());  
}  
  
fs.readFile("./ejemplo/archivo.txt", leer);  
  
console.log("Ultima linea del programa");
```

Nombre	Fecha de modificación	Tipo
archivo.txt	21/11/2019 16:27	Doc

```

root@DESKTOP-874LG08:/mnt/c/Users/isa_a/Desktop/nodejsTutorial/cod
igoEjemplo# node moduloFS.js
Ultima linea del programa
El archivo fue creado
linea 1
linea 2

```

Hasta ahora, todo puede parecer, muy lógico, incluso se hace un poco aburrido. Lo que queremos es comenzar a ver programas un poco más en profundidad, así que vamos a comenzar creando nuestro propio servidor HTTP Básico.

6.2. El concepto de Evento

Los eventos en node.js, que no tienen nada que ver con los eventos JavaScript. Aquí los eventos se producen en el servidor y pueden ser de distintos tipos, dependiendo de la librería o clase que estemos trabajando.

Un ejemplo de evento sería el evento de recibir una solicitud en un servidor HTTP.

Los eventos se encuentran en un módulo independiente que tenemos que requerir en nuestros programas. Esto se hace con la ya conocida secuencia *require*.

```
var eventos = require("events");
```

De esta forma tendremos acceso a una serie de utilidades para trabajar con eventos.

6.2.1. Como definir un evento en Node.js.

En primer lugar, vamos a ver el emisor de eventos.

```
var EmisorEventos = eventos.EventEmitter;
```

A continuación debemos instanciar el objeto de la clase *EventEmitter* que hemos guardado en *EmisorEventos*:

```
var ee= new EmisorEventos();
```

Para emitir un evento en JavaScript usaremos el método *emit()*. Por ejemplo, si queremos emitir un evento llamado "datos", haríamos:

```
ee.emit("datos", Date.now());
```

Para manejar este evento, creamos una función manejadora de eventos haciendo uso del método *on()*:

```
ee.on("datos", funcion(fecha){
    console.log(fecha);
});
```

6.3. Crear Nuestro Propio Servidor HTTP Básico.

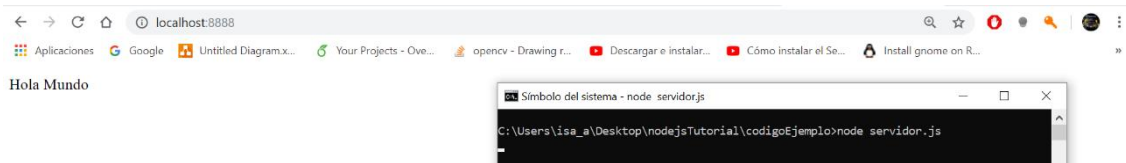
Para comenzar, vamos a crear un archivo main, llamado index.js, el cual será usado para iniciar nuestra aplicación, y un archivo de módulo, servidor.js, que contendrá el código de nuestro servidor HTTP.

En servidor.js, tendremos el siguiente código:

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

http.createServer(function(request, response) {
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("Hola Mundo");
    response.end();
}).listen(8888);
```

Para comprobar que nuestro servidor funciona, tan solo tendremos que ejecutarlo en la consola, y abrir en nuestro navegador apuntándolo a <http://localhost:8888/>, y veremos que nos muestra una página web con el texto "Hola Mundo"



Vamos a analizar un poco el código que hemos utilizado;

-*require* -> requiere al módulo http, que es un módulo de node.js, donde al igualarlo a la variable http, lo hacemos accesible.

- *createServer* -> es una de las funciones que ofrece el módulo http. Esta retorna un objeto, que a su vez tiene un método llamado listen, que nos indica el número de puerto en que nuestro servidor HTTP va a escuchar.

Como podemos observar, a createServer se le están pasando como parámetros, *request* y *response*, donde *request* no se trata de una variable, si no de una función anónima.

De esta forma, nuestro código también podría quedar de la siguiente manera:

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

function onRequest(request, response) {
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("Hola Mundo");
    response.end();
} http.createServer(onRequest).listen(8888);
```

-*onRequest* -> se trata de una función de *callback*, que son funciones que se pasan como parámetros a otras funciones y se ejecutan dentro de éstas.

Cuando la función *callback* es disparada y nuestra función *onRequest* es gatillada, se les pasa dos parámetros: *request* y *response*, que pueden ser usados para enviar algo de vuelta al navegador que hizo la petición a tu servidor.

En resumen, lo que nuestro código hace es que cada vez que una petición es recibida, mediante la función *responde.writeHead()*, envía un estatus HTTP 200 y un *content-type*(tipo de contenido) en el encabezado de la respuesta HTTP, con la función *response.write()* envía el mensaje "hola mundo" en el cuerpo de la respuesta y para terminar, con la función *responde.end()* finalizamos la respuesta,

Lo que estamos haciendo es refactorizar nuestro código. Esto se hace de esta manera, ya que es la naturaleza con la que node.js trabaja, está orientado al evento, razón por la que es tan rápido. Como node.js trabaja de manera asíncrona, esto nos va a ayudar a poder manipular las peticiones que entran a nuestro servidor.

Por ejemplo, vamos a probar si nuestro código continuo después de haber creado el servidor, incluso si no ha sucedido ninguna petición HTTP y la función que pasamos no ha sido llamada:

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

function onRequest(request, response) {
    console.log("Petición Recibida.");
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("Hola Mundo");
    response.end();
}

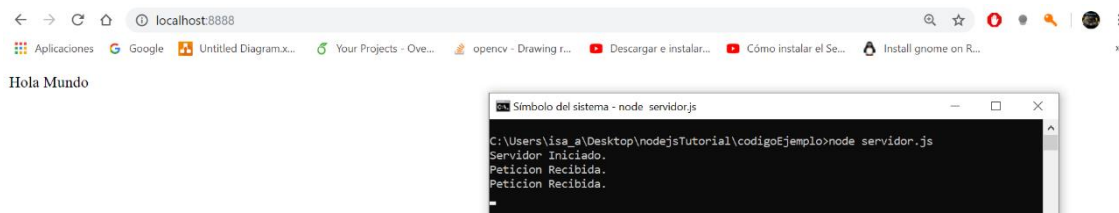
http.createServer(onRequest).listen(8888);

console.log("Servidor Iniciado.");
```

Si ejecutamos este nuevo código, vemos que nada más iniciar, en la consola nos escribirá "Servidor Iniciado", y que cada vez que hagamos una petición al servidor, abriendo el navegador con la dirección <http://localhost:8888/> mostrará el mensaje "Petición Recibida".

Esto es lo que se llama *JavaScript* del lado del servidor asíncrono y orientado al evento.

(Toma en cuenta que nuestro servidor probablemente escribirá "Petición Recibida." a STDOUT dos veces al abrir la página en el navegador. Esto es porque la mayoría de los browsers van a tratar de cargar el favicon mediante la petición <http://localhost:8888/favicon.ico> cada vez que abras <http://localhost:8888/>).



Ahora que ya sabemos la forma de crear el servidor y conocemos, más o menos, su funcionamiento, vamos a ver cómo organizar nuestra aplicación.

Lo normal para que nuestro programa este modularizado y sea más fácil y cómo de entender, es tener un archivo principal, index.js que creamos al comenzar con este tutorial. Este archivo será usado para arrancar los distintos módulos que pueda tener nuestra aplicación, como el módulo de servidor HTTP que ya hemos creado en el servidor.js.

Primero veremos cómo podemos hacer que nuestro servidor.js sea un verdadero módulo de Node.js para poder usarlo desde nuestro index.js.

Como hemos comentado anteriormente, ya hemos usado módulos propios de Node.js en nuestro código, como pueden ser:

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

Pues bien, ahora que ya sabemos cómo usar los módulos internos de Node.js, vamos a ver cómo podemos crear nuestros propios módulos y cómo podemos utilizarlos.

Hacer que algún código sea un módulo, no es complicado, significa que tenemos que exportar las partes de su funcionalidad que queremos proveer a otros scripts que requieran nuestro módulo.

A continuación, incluiremos en nuestro código del servidor una función llamada inicio, y esta función será la que vamos a exportar:

```
var http = require("http");  
  
function iniciar() {  
    function onRequest(request, response) {  
        console.log("Petición Recibida.");  
        response.writeHead(200, {"Content-Type": "text/html"});  
        response.write("Hola Mundo");  
        response.end();  
    }  
    http.createServer(onRequest).listen(8888);  
}
```

```
console.log("Servidor Iniciado.");  
  
}  
  
exports.iniciar = iniciar;
```

Como ya te habrás dado cuenta, esto no es nuevo para nosotros. Como explicamos el concepto de módulo, ya vimos lo que era el objeto exports y como se utilizaba.

Aun así, para comprobar que todo funciona de forma correcta, en nuestro index.js vamos a añadir el siguiente código:

```
var server = require("./servidor");  
  
server.iniciar();
```

Esta vez, en lugar de ver el resultado en nuestra consola, lo veremos en el mismo navegador, ejecutando node index.js

Ya tenemos nuestra primera parte de la aplicación, podemos recibir peticiones HTTP, pero... ¿qué hacemos ahora con ellas? necesitamos reaccionar de manera diferente dependiendo de que URL el navegador requiera de nuestro servidor.

Si tenemos una aplicación muy simple, esto se podría hacer dentro de la función *callback OnRequest()*, pero nosotros queremos tener más abstracción y hacer nuestra aplicación más interesante.

Hacer distintas peticiones HTTP e ir a partes diferentes de nuestro código se llama "ruteo", por lo que vamos a crear un módulo nuevo llamado router.js.

Para poder rutear peticiones tenemos que ser capaces de entregar la URL requerida y los parámetros GET o POST adicionales a nuestro *router*. Dependiendo de estos, nuestro *router* debe ser capaz de decidir qué código va a ejecutar.

Toda esta información está disponible en el objeto *request*, aquel que si recordamos, es pasado como primer parámetro a nuestra función *OnRequest()*.

Para poder interpretar esta información, necesitamos algunos módulos de Node.js, como son el módulo url, que nos permite extraer las diferentes partes de una URL y el módulo *querystring*, que puede ser usado para *parsear* el *string* de consulta para los parámetros requeridos.

Ahora vamos a añadir a nuestra función *onRequest()* la lógica requerida para encontrar que ruta URL solicitó el navegador:

```
var http = require("http");  
  
var url = require("url");  
  
function iniciar() {  
  
    function onRequest(request, response) {  
  
        var pathname = url.parse(request.url).pathname;  
  
        console.log("Petición para " + pathname + " recibida.");
```

```

        response.writeHead(200, {"Content-Type": "text/html"});
        response.write("Hola Mundo");
        response.end();
    }

    http.createServer(onRequest).listen(8888);
    console.log("Servidor Iniciado.");
}

exports.iniciar = iniciar;

```

Ahora, nuestra aplicación puede distinguir peticiones basadas en la ruta URL requerida. Esto significa que seremos capaces de tener peticiones para las URLs /iniciar y /subir, manejadas por partes diferentes de nuestro código.

Muy, ahora vamos a pasar a escribir nuestro código. En nuestro archivo router.js vamos a escribir el siguiente contenido:

```

function route(pathname) {
    console.log("A punto de rutear una petición para " + pathname);
}

exports.route = route;

```

De momento, este código no va a hacer nada, pero ahora veremos cómo vamos a encajar este *router* con nuestro servidor antes de poner más lógica en el *router*.

Lo primero que vamos a hacer es extender nuestra función *iniciar()*, para permitir pasar la función de *router* a ser usada como parámetro:

```

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

function iniciar(route) {
    function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log("Petición para " + pathname + " recibida.");
        route(pathname);
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write("Hola Mundo");
        response.end();
    }
}

```

```

    }

    http.createServer(onRequest).listen(8888);

    console.log("Servidor Iniciado.");
  }

  exports.iniciar = iniciar;

```

Y añadimos a nuestro index.js la función de ruteo de nuestro *router* en el servidor.

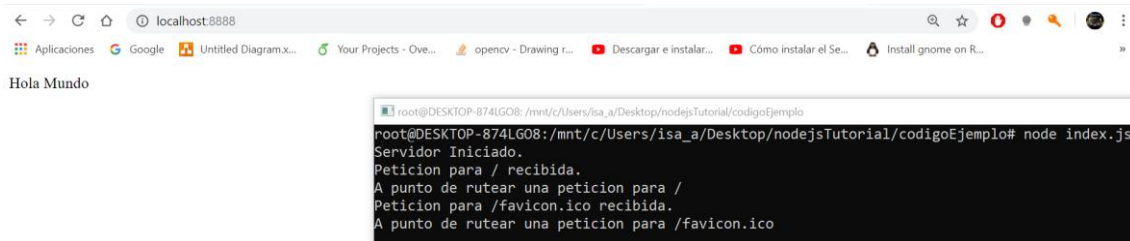
```

var server = require("./servidor");
var router = require("./router");

server.iniciar(router.route);

```

Si ahora arrancamos nuestra aplicación y hacemos una petición para una URL, puedes ver ahora por las respuestas de la aplicación, que nuestro servidor HTTP hace uno de nuestro *router* y le entrega el nombre de ruta requerido.



Nuestro servidor HTTP y nuestro *router* ahora se comunican entre ellos, tal y como pretendíamos.

Para poder continuar, y darle verdadera funcionalidad a todo lo que estamos haciendo, vamos a crear un nuevo módulo llamado requestHandlers.js en el que vamos a agregar una función de ubicación para cada manipulador de petición, y los exportamos como métodos para el módulo:

```

function iniciar() {
    console.log("Manipulador de petición 'iniciar' ha sido llamado.");
}

function subir() {
    console.log("Manipulador de petición 'subir' ha sido llamado.");
}

exports.iniciar = iniciar;
exports.subir = subir;

```

Esto nos permitirá atar los manipuladores de petición al *router*, dándole a nuestro *router* algo que rutear.

Hay varias formas de pasar los manipuladores de petición, ya sea desde directamente dentro del código del *router* o pasándolos al servidor, y desde este hacia el *router*. Aunque la segunda forma parece más larga, si llega un momento que tenemos muchos manipuladores, tendríamos que estar mapeando peticiones cada vez que una nueva URL o manipulador de petición sea agregado.

Bien, ya que los objetos en JavaScript son solo colecciones de pares nombre/valor, y estos valores pueden ser *strings*, números e incluso funciones, vamos a pasar la lista de *requestHandlers* (manipuladores de petición) como un objeto y vamos a usar la técnica de inyectar este objeto en la *route()*.

Vamos a comenzar con poner el objeto en nuestro *index.js*:

```
var server = require("./servidor");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.iniciar;
handle["/iniciar"] = requestHandlers.iniciar;
handle["/subir"] = requestHandlers.subir;

server.iniciar(router.route, handle);
```

Como podemos ver, con tan solo añadir un par llave/valor de "/" y *requestHandlers.iniciar*, podemos expresar de forma limpia que no solo pueden ser manejadas peticiones a */start*, si no también peticiones a */*.

Una vez definido nuestro objeto, se lo pasamos al servidor como un parámetro adicional. En nuestro *servidor.js* añadimos:

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

function iniciar(route, handle) {
    function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
```

```

        console.log("Petición para " + pathname + " recibida.");
        route(handle, pathname);
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write("Hola Mundo");
        response.end();
    }

    http.createServer(onRequest).listen(8888);
    console.log("Servidor Iniciado.");
}

exports.iniciar = iniciar;

```

Lo que hacemos aquí, es comprobar que un manipulador de peticiones para una ruta dada existe, y en el caso de hacerlo, llamamos a la función adecuada.

Esto es todo lo que necesitamos para unir nuestro servidor, *router* y manipuladores de peticiones.

Ahora, si los manipuladores de petición pudieran enviar algo de vuelta al navegador sería mucho mejor.

En este momento, nuestra aplicación es capaz de transportar el contenido desde los manipuladores de petición al servidor HTTP.

Nuestro objetivo ahora, que en vez de llevar el contenido al servidor, llevaremos el servidor al contenido. Inyectaremos el objeto *response* a través de nuestro *router* a los manipuladores de petición. Los manipuladores serán capaces de usar las funciones de este objeto para responder a las peticiones ellos mismos.

Para esto, vamos a pasar a cambiar nuestra aplicación.

Vamos a empezar por `servidor.js`:

```

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

function iniciar(route, handle) {
    function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log("Petición para " + pathname + " recibida.");

        route(handle, pathname, response);
    }
}

```

```
http.createServer(onRequest).listen(8888);  
console.log("Servidor Iniciado.");  
}  
exports.iniciar = iniciar;
```

En nuestro router.js:

```
function route(handle, pathname, response) {  
  console.log("A punto de rutear una petición para " + pathname);  
  if (typeof handle[pathname] === 'function') {  
    handle[pathname](response);  
  } else {  
    console.log("No hay manipulador de petición para " + pathname);  
    response.writeHead(404, {"Content-Type": "text/html"});  
    response.write("404 No Encontrado");  
    response.end();  
  }  
}  
exports.route = route;
```

Y para acabar, modificamos a requestHandlers.js:

```
var exec = require("child_process").exec;  
  
function iniciar(response) {  
  console.log("Manipulador de petición 'iniciar' fue llamado.");  
  
  exec("ls -lah", function (error, stdout, stderr) {  
    response.writeHead(200, {"Content-Type": "text/html"});  
    response.write(stdout);  
    response.end();  
  });  
}
```

```

});
}

function subir(response) {
  console.log("Manipulador de petición 'subir' fue llamado.");
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("Hola Subir");
  response.end();
}

exports.iniciar = iniciar;
exports.subir = subir;

```

Las funciones manipuladoras necesitan aceptar el parámetro de respuesta `response` para poder hacer uso de él para responder a la petición directamente.

El manipulador `iniciar` responderá con el *callback* anónimo `exec()`, y el manipulador `subir` replicará simplemente con "Hola Subir", pero esta vez, haciendo uso del objeto `response`.

A continuación, arrancamos nuestra aplicación de vemos que debería funcionar de acuerdo con lo esperado.

Ya que tenemos nuestro servidor, *router* y manipuladores de petición en su lugar, podemos empezar a agregar contenido a nuestro sitio que permitirá a nuestros usuarios interactuar y andar a través de los casos de uso de elegir un archivo, subirlo y ver el contenido en el navegador.

Primero veremos cómo manejar peticiones POST entrantes y haremos uso de un módulo externo de Node.js para la manipulación de archivos.

6.3.1. Peticiones POST

Vamos a realizar algo muy simple. Tendremos un área de texto, que podría ser rellenada por el usuario y enviada al servidor en una petición POST. Una vez recibida, mostraremos el contenido en un *text* área.

Para ello, agregaremos el HTML a nuestro manipulador de petición `/iniciar`, en el archivo `requestHandlers.js` añadimos:

```

function iniciar(response) {
  console.log("Manipulador de peticiones 'iniciar' fue llamado.");
  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html;'+
    'charset=UTF-8" />'+

```



```

'</head>'+
'<body>'+
'<form action="/subir" method="post">'+
'<textarea name="text" rows="20" cols="60"></textarea>'+
'<input type="submit" value="Enviar texto" />'+
'</form>'+
'</body>'+
'</html>';

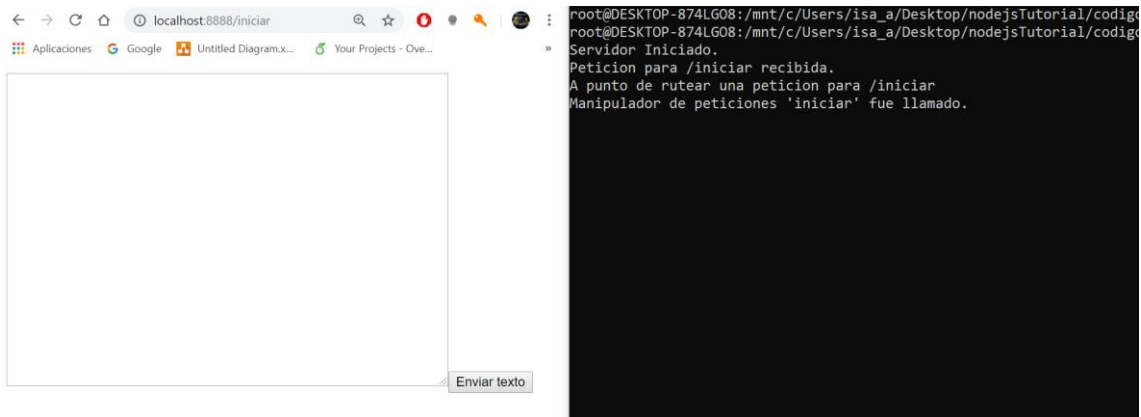
response.writeHead(200, {"Content-Type": "text/html"});
response.write(body);
response.end();
}

function subir(response) {
  console.log("Manipulador de peticiones 'subir' fue llamado.");
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("Hola Subir");
  response.end();
}

exports.iniciar = iniciar;
exports.subir = subir;

```

Si ahora ejecutamos de nuevo nuestro index.js y hacemos una petición a nuestro navegador con <http://localhost:8888/iniciar> deberíamos ver nuestro formulario.



Node.js le entrega a nuestro código la información POST en pequeños trozos con *callbacks* que son llamadas ante determinados eventos, que son *data* y *end*. Necesitamos decirle a Node.js que funciones llamar de vuelta cuando estos eventos ocurran.

Esto se consigue agregando *listeners* al objeto *request* que es pasado a nuestro *callback onRequest* cada vez que una petición HTTP es recibida.

Para esto, tenemos que modificar nuestro *servidor.js*, añadiendo:

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

function iniciar(route, handle) {
  function onRequest(request, response) {
    var dataPosteada = "";
    var pathname = url.parse(request.url).pathname;
    console.log("Petición para " + pathname + " recibida.");

    request.setEncoding("utf8");

    request.addListener("data", function(trozoPosteado) {
      dataPosteada += trozoPosteado;
      console.log("Recibido trozo POST '" + trozoPosteado + "'.");
    });

    request.addListener("end", function() {
      route(handle, pathname, response, dataPosteada);
    });
  }
}
```

```

    }

    http.createServer(onRequest).listen(8888);

    console.log("Servidor Iniciado");
  }

  exports.iniciar = iniciar;

```

Lo que hemos hecho así ha sido, definir que esperamos que la codificación de la información recibida sea UTF-8, agregamos un *listeners* de eventos para "data", el cual llena paso a paso nuestra variable *dataPosteada* cada vez que un nuevo trozo de información POST llega y movemos la llamada desde nuestro *router* al *callback* del evento *end* de manera de asegurarnos que sólo sea llamado cuando toda la información POST sea reunida. Por último, pasamos la información POST al *router*, ya que la vamos a necesitar en nuestros manipuladores de eventos.

Ahora, en la página /subir, desplegaremos el contenido recibido, por lo que tenemos que modificar *router.js*:

```

function route(handle, pathname, response, postData) {
  console.log("A punto de rutear una petición para " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, postData);
  } else {
    console.log("No se ha encontrado manipulador para " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 No encontrado");
    response.end();
  }
}

exports.route = route;

```

Y en requestHandlers.js, incluimos la información de nuestro manipulador de petición subir:

```
function iniciar(response, postData) {
  console.log("Manipulador de Peticion 'iniciar' fue llamado.");

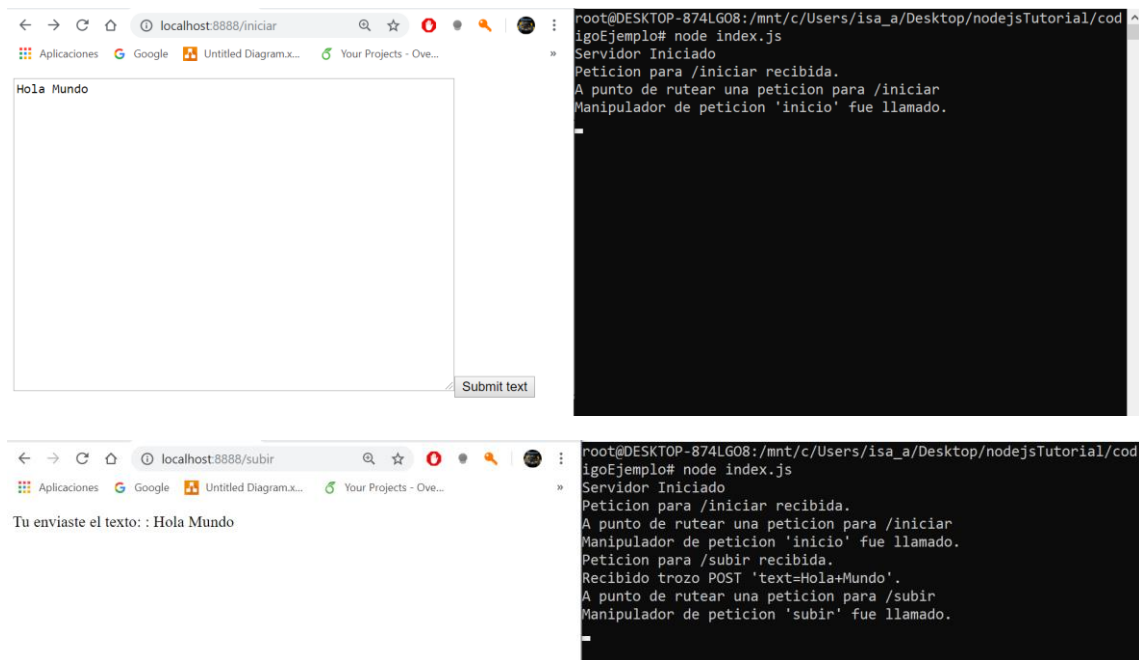
  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/subir" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Enviar texto" />'+
    '</form>'+
    '</body>'+
    '</html>';

  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(body);
  response.end();
}

function subir(response, dataPosteada) {
  console.log("Manipulador de Peticion 'subir' fue llamado.");
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("Tu enviaste: " + dataPosteada);
  response.end();
}

exports.iniciar = iniciar;
exports.subir = subir;
```

Con esto, ahora somos capaces de recibir información POST y usarla en nuestros manipuladores de petición.



6.4. Usando Node.js y NPM.

Como ya explicamos al principio de este tutorial, NPM es un gestor de paquetes, que tiene una gran cantidad de módulos disponibles y nos facilita la instalación de dependencias necesarias para nuestro proyecto con una serie simple de comandos.

Un paquete de Node.js contiene todos los archivos que necesita para un módulo, que son bibliotecas de JavaScript que se pueden incluir en un proyecto.

La descarga de paquetes es muy sencilla. Mediante línea de comandos, tan solo había que escribir:

```
npm install <paquete>
```

Por ejemplo, si queremos instalar un paquete llamado “mayúsculas, tan solo tendríamos que hacer:

```
npm install upper-case
```

NPM nos creará una carpeta llamada *node_modules* en el directorio de nuestra aplicación, donde se colocará este paquete y todos los que se instalen en un futuro.

Estos paquetes se usan de la misma forma que usamos cualquier otro módulo. Por ejemplo, en nuestro, si en nuestro ejemplo anterior, en el archivo *requestHandlers.js*, añadimos justo al inicio:

```
Var uc = require("uper-case");
```

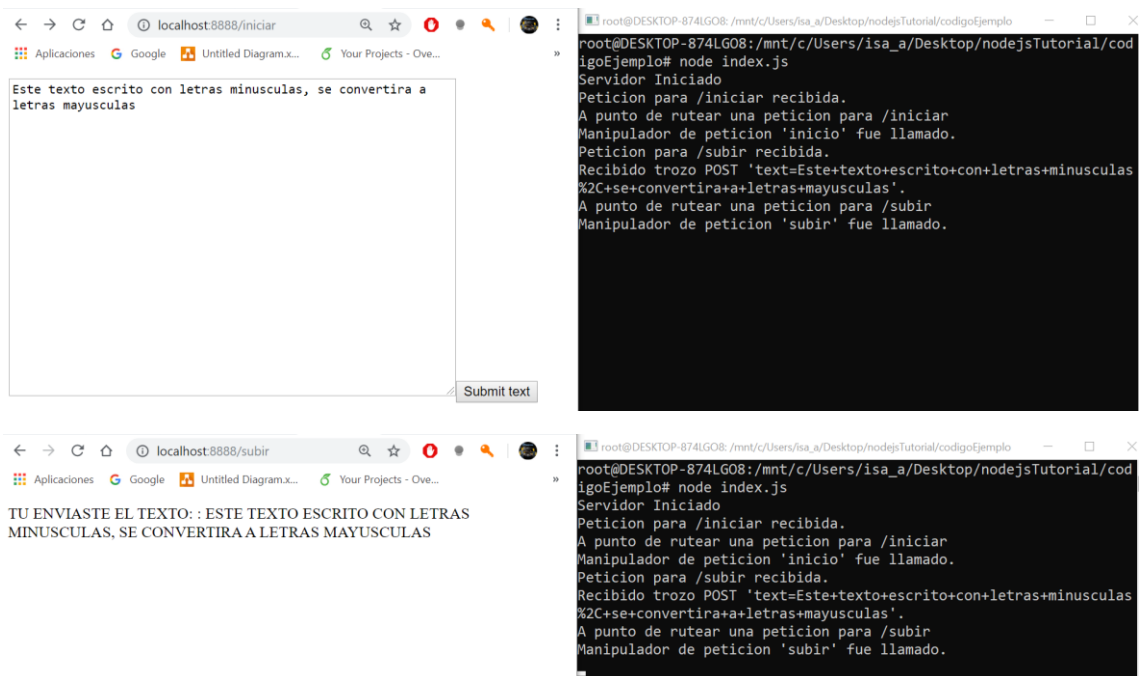
La línea:

```
response.write("Tu enviaste el texto: " + querystring.parse(dataPosteada)["text"]);
```

La editamos para que quede de esta forma:

```
response.write(uc("Tu enviaste el texto: " + querystring.parse(dataPosteada)["text"]));
```

Y volvemos a ejecutar nuestro index.js, abriendo después el navegador, veremos cómo esto funciona.



7. Bibliografía

- Autentia. (22 de Enero de 2016). *Autentia*. Obtenido de <https://www.autentia.com/2016/01/22/java-vs-node-js-el-encuentro/>
- Llamas, L. (29 de Agosto de 2017). *Luis Llamas*. Obtenido de <https://www.luisllamas.es/que-es-node-js/>
- Manuel Kiessling, H. A. (s.f.). *Node Beginner*. Obtenido de <https://www.nodebeginner.org/index-es.html>
- tutorialesprogramacionya*. (s.f.). Obtenido de <https://www.tutorialesprogramacionya.com/javascriptya/nodejsya/index.php?inicio=0>
- um.es*. (2016/2017). Obtenido de <https://www.um.es/docencia/barzana/DAWEB/2017-18/daweb-NodeJS.pdf>
- Vazquez, J. R. (10 de Febrero de 2016). *Medium*. Obtenido de <https://medium.com/javascript-comunidad/cómo-instalar-node-js-y-npm-en-mac-9d80f26fb88d>
- w3schools.com*. (s.f.). Obtenido de <https://www.w3schools.com/nodejs/>
- Wikipedia*. (17 de Octubre de 2019). Obtenido de <https://es.wikipedia.org/wiki/Node.js>
- Wikipedia*. (s.f.). *Wikipedia*. Obtenido de Wikipedia.