https://www.nodebeginner.org/index-es.html

El Libro para Principiantes en Node.js

Un tutorial de Node.js por: Manuel Kiessling & Herman A. Junge

Sobre el Tutorial

El objetivo de este documento es ayudarte a empezar con el desarrollo de aplicaciones para Node.js, enseñándote todo lo que necesites saber acerca de JavaScript "avanzado" sobre la marcha. Este tutorial va mucho más allá del típico manual "Hola Mundo".

Status

Estás leyendo la versión final de este libro, es decir, las actualizaciones solo serán hechas para corregir errores o para reflejar cambiar en nuevas versiones de Node.js.

Las muestras de código de este libro están probadas para funcionar con la versión 0.6.11 de Node.js.

Audiencia Objetivo

Este documento probablemente será mejor entendido por los lectores que tengan un trasfondo similar al mío: Programadores experimentados en al menos un lenguaje orientado al objeto, como Ruby, Python, PHP o Java; poca experiencia con JavaScript, y ninguna experiencia en Node.js.

El que este documento esté orientado a desarrolladores que ya tienen experiencia con otros lenguajes de programación significa que no vamos a cubrir temas realmente básicos como tipos de datos, variables, estructuras de control y similares. Debes saber acerca de estos tópicos para entender este documento.

Sin embargo, dado que las funciones y objetos en JavaScript son diferentes de sus contrapartes en la mayoría de los lenguajes, estos serán explicados con más detalle.

Estructura de este documento

Al Término de este documento, habrás creado una aplicación Web completa, que permita a los usuarios de ésta el ver páginas web y subir archivos.

La cual, por supuesto, no va ser nada como la "aplicación que va a cambiar el mundo", no obstante eso, nosotros haremos algo más y no vamos sólo a codificar una aplicación lo "suficientemente simple" para hacer estos casos de uso posible, sino que crearemos un framework sencillo, pero completo, a fin de poder separar los distintos aspectos de nuestra aplicación. Verás lo que esto significa en poco tiempo.

Empezaremos por mirar cómo el desarrollo en JavaScript en Node.js es diferente del desarrollo en JavaScript en un browser.

Luego, nos mantendremos con la vieja tradición de escribir una aplicación "Hola Mundo", la cual es la aplicación más básica de Node.js que "hace" algo.

Enseguida, discutiremos que tipo de "aplicación del mundo real" queremos construir, disectaremos las diferentes partes que necesitan ser implementadas para ensamblar esta aplicación, y empezaremos trabajando en cada una de estas partes paso a paso.

Según lo prometido, aprenderemos sobre la marcha acerca de algunos de los muchos conceptos avanzados de JavaScript, como hacer uso de ellos, y ver el por qué tiene sentido el hacer uso de estos conceptos en vez de los que ya conocemos por otros lenguajes de programación.

Tabla de Contenidos

JavaScript y Node.js

JavaScript y Tú

Antes que hablemos de toda la parte técnica, tomémonos un minuto y hablemos acerca de ti y tu relación con JavaScript. Este capítulo está aquí para permitirte estimar si tiene sentido el que sigas o no leyendo este documento.

Si eres como yo, empezaste con el "desarrollo" HTML hace bastante tiempo, escribiendo documentos HTML. Te encontraste en el camino con esta cosa simpática llamada JavaScript, pero solo la usabas en una forma muy básica, agregando interactividad a tus páginas de cuando en cuando.

Lo que realmente quisiste era "la cosa real", querías saber cómo construir sitios web complejos - Aprendiste un lenguaje de programación como PHP, Ruby, Java, y empezaste a escribir código "backend".

No obstante, mantuviste un ojo en JavaScript, y te diste cuenta que con la introducción de jQuery, Prototype y otros, las cosas se fueron poniendo más avanzadas en las Tierras de JavaScript, y que este lenguaje era realmente más que hacer un *window.open()*.

Sin embargo, esto era todo cosa del *frontend*, y aunque era agradable contar con jQuery a tu disposición en cualquier momento que te sintieras con ánimo de sazonar una página web, al final del día, lo que eras a lo más, era un usuario de JavaScript, pero no, un desarrollador de JavaScript.

Y entonces llegó Node.js. JavaScript en el servidor, ¿Qué hay con eso?

Decidiste que era ya tiempo de revisar el nuevo JavaScript. Pero espera: Escribir aplicaciones Node.js es una cosa; Entender el por qué ellas necesitan ser escritas en la manera que lo son significa entender JavaScript! Y esta vez es en serio.

Y aquí está el problema: Ya que JavaScript realmente vive dos, o tal vez tres vidas (El pequeño ayudante DHTML de mediados de los 90's, las cosas más serias tales como jQuery y similares, y ahora, el lado del servidor), no es tan fácil encontrar información que te ayude a aprender JavaScript de la "manera correcta", de forma de poder escribir aplicaciones de Node.js en una apariencia que te haga sentir que no sólo estás usando JavaScript, sino que también están desarrollando con él.

Porque ahí está el asunto: Ya eres un desarrollador experimentado, y no quieres aprender una nueva técnica simplemente metiendo código aquí y allá mal-aprovechándolo; Quieres estar seguro que te estás enfocando en un ángulo correcto.

Hay, por supuesto, excelente documentación afuera. Pero la documentación por sí sola no es suficiente. Lo que se necesita es una guía.

Mi objetivo es proveerte esta guía.

Una Advertencia

Hay algunas personas realmente excelente en JavaScript. No soy una de ellas.

Yo soy realmente el tipo del que te he hablado en los párrafos previos. Sé un par de cosas acerca de desarrollar aplicaciones backend, pero aún soy nuevo al JavaScript "real" y aún más nuevo a Node.js. He aprendido solo recientemente alguno de los aspectos avanzados de JavaScript. No soy experimentado.

Por lo que este no es un libro "desde novicio hasta experto". Este es más bien un libro "desde novicio a novicio avanzado".

Si no fallo, entonces este será el tipo de documento que deseo hubiese tenido cuando empecé con Node.js.

JavaScript del Lado del Servidor

Las primeras encarnaciones de JavaScript vivían en los browsers. Pero esto es sólo el contexto. Define lo que puedes hacer con el lenguaje, pero no dice mucho acerca de lo que el lenguaje mismo puede hacer. JavaScript es un lenguaje "completo": Lo puedes usar en muchos contextos y alcanzar con éste, todo lo que puedes alcanzar con cualquier otro lenguaje "completo".

Node.js realmente es sólo otro contexto: te permite correr código JavaScript en el backend, fuera del browser.

Para ejecutar el código JavaScript que tu pretendes correr en el backend, este necesita ser interpretado y, bueno, ejecutado, Esto es lo que Node.js realiza, haciendo uso de la Maquina Virtual V8 de Google, el mismo entorno de ejecución para JavaScript que Google Chrome utiliza.

Además, Node.js viene con muchos módulos útiles, de manera que no tienes que escribir todo de cero, como por ejemplo, algo que ponga un string a la consola.

Entonces, Node.js es en realidad dos cosas: un entorno de ejecución y una librería.

Para hacer uso de éstas (la librería y el entorno), necesitas instalar Node.js. En lugar de repetir el proceso aquí, te ruego visitar las instrucciones oficiales de instalación. Por favor vuelve una vez que tengas tu versión de Node.js corriendo.

"Hola Mundo"

Ok. Saltemos entonces al agua fría y escribamos nuestra primera aplicación Node.js: "Hola Mundo".

Abre tu editor favorito y crea un archivo llamado *holamundo.js*. Nosotros queremos escribir "Hola Mundo" a STDOUT, y aquí está el código necesario para hacer esto:

```
001
```

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

Graba el archivo, y ejecútalo a través de Node.js:

```
node holamundo.js
```

Este debería retornar *Hola Mundo* en tu monitor.

Ok, esto es aburrido, de acuerdo? Así que escribamos alguna cosa real.

Una Aplicación Web Completa con Node.js

Los casos de Uso

Mantengámoslo simple, pero realista:

- El Usuario debería ser capaz de ocupar nuestra aplicación con un browser.
- El Usuario debería ver una página de bienvenida cuando solicita http://*dominio*/inicio, la cual despliega un formulario de subida.
- Eligiendo un archivo de imagen para subir y enviando el formulario, la imagen debería ser subida a http://*dominio*/subir, donde es desplegada una vez que la subida este finalizada.

Muy bien. Ahora, tu puedes ser capaz de alcanzar este objetivo googleando y programando *lo que sea*, pero eso no es lo que queremos hacer aquí.

Más que eso, no queremos escribir simplemente el código más básico posible para alcanzar este objetivo, no importa lo elegante y correcto que pueda ser este código. Nosotros agregaremos intencionalmente más abstracción de la necesaria de manera de poder tener una idea de lo que es construir aplicaciones más complejas de Node.js.

La Pila de Aplicaciones

Hagamos un desglose a nuestra aplicación. ¿Qué partes necesitan ser implementadas para poder satisfacer nuestros casos de uso?

- Queremos servir páginas web, de manera que necesitamos un **Servidor HTTP**.
- Nuestro servidor necesitará responder directamente peticiones (requests), dependiendo de qué URL sea pedida en este requerimiento, es que necesitaremos algún tipo de **enrutador (router)** de manera de mapear los peticiones a los handlers (manejadores) de éstos.
- Para satisfacer a los peticiones que llegaron al servidor y han sido ruteados usando el enrutador, necesitaremos de hecho **handlers (manejadores) de peticiones**
- El Enrutador probablemente debería tratar cualquier información POST que llegue y dársela a los handlers de peticiones en una forma conveniente, luego necesitaremos **manipulación de data de petición**
- Nosotros no solo queremos manejar peticiones de URLs, sino que también queremos desplegar contenido cuando estas URLs sean pedidas, lo que significa que necesitamos

algún tipo de **lógica en las vistas** a ser utilizada por los handlers de peticiones, de manera de poder enviar contenido al browser del Usuario.
- Por último, pero no menos importante, el Usuario será capaz de subir imágenes, así que necesitaremos algún tipo de **manipulación de subidas** quien se hará cargo de los detalles.

Pensemos un momento acerca de como construiríamos esta pila de aplicaciones con PHP. No es exactamente un secreto que la configuración típica sería un Apache HTTP server con mod_php5 instalado.
Lo que, a su vez, significa que el tema "Necesitamos ser capaces de servir páginas web y recibir peticiones HTTP" ni siquiera sucede dentro de PHP mismo.

Bueno, con Node.js, las cosas son un poco distintas. Porque con Node.js, no solo implementamos nuestra aplicación, nosotros también implementamos todo el servidor HTTP completo. De hecho, nuestra aplicación web y su servidor web son básicamente lo mismo.

Esto puede sonar como mucho trabajo, pero veremos en un momento que con Node.js, no lo es.

Empecemos por el principio e implementemos la primera parte de nuestra pila, el servidor HTTP..

Construyendo la Pila de Aplicaciones

Un Servidor HTTP Básico

Cuando llegué al punto donde quería empezar con mi primera aplicación Node.js "real", me pregunté no solo como la iba a programar, sino que también, como organizar mi código. ¿Necesitaré tenerlo todo en un archivo? Muchos tutoriales en la Web que te enseñan cómo escribir un servidor HTTP básico en Node.js tienen toda la lógica en un solo lugar. ¿Qué pasa si yo quiero asegurarme que mi código se mantenga leíble a medida que le vaya agregando más cosas?

Resulta, que es relativamente fácil de mantener los distintos aspectos de tu código separados, poniéndolos en módulos.

Esto te permite tener un archivo *main* limpio, en el cual ejecutas Node.js, y módulos limpios que pueden ser utilizados por el archivo *main* entre muchos otros.

Así que vamos a crear un archivo *main* el cual usaremos para iniciar nuestra aplicación, y un archivo de módulo dónde residirá el código de nuestro servidor HTTP.

Mi impresión es que es más o menos un estándar nombrar a tu archivo *principal* como *index.js*. Tiene sentido también que pongamos nuestro módulo de servidor en un archivo llamado *server.js*.

Empecemos con el módulo del servidor. Crea el archivo *server.js* en el directorio raíz de tu proyecto, y llénalo con el código siguiente:
002

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

Eso es! Acabas de escribir un servidor HTTP activo. Probémoslo ejecutándolo y testeándolo. Primero ejecuta tu script con Node.js:

```
node server.js
```

Ahora, abre tu browser y apúntalo a http://localhost:8888/. Esto debería desplegar una página web que diga "Hola Mundo".

Interesante, ¿no? ¿Qué tal si hablamos de que está pasando aquí y dejamos la pregunta de 'cómo organizar nuestro proyecto' para después? Prometo que volveremos a esto.

Analizando nuestro servidor HTTP

Bueno, entonces, analicemos que está pasando aquí.

La primera línea *require*, requiere al módulo *http* que viene incluido con Node.js y lo hace accesible a través de la variable *http*.

Luego llamamos a una de las funciones que el módulo http ofrece: *createServer*. Esta función retorna un objeto, y este objeto tiene un método llamado *listen* (escucha), y toma un valor numérico que indica el número de puerto en que nuestro servidor HTTP va a escuchar.

Por favor ignora por un segundo a la definición de función que sigue a la llave de apertura de *http.createServer*.

Nosotros podríamos haber escrito el código que inicia a nuestro servidor y lo hace escuchar al puerto 8888 de la siguiente manera:

```
var http = require("http");
var server = http.createServer();
server.listen(8888);
```

Esto hubiese iniciado al servidor HTTP en el puerto 8888 y no hubiese hecho nada más (ni siquiera respondido alguna petición entrante).
La parte realmente interesante (y rara, si tu trasfondo es en un lenguaje más conservador, como PHP) es que la definición de función está ahí mismo donde uno esperaría el primer parámetro de la llamada a *createServer()*.
Resulta que, este definición de función ES el primer (y único) parámetro que le vamos a dar a la llamada a *createServer()*. Ya que en JavaScript, las funciones pueden ser pasadas de un lado a otro como cualquier otro valor.

Pasando Funciones de un Lado a Otro

Puedes, por ejemplo, hacer algo como esto:

```
function decir(palabra) {
  console.log(palabra);
```

```
}

function ejecutar(algunaFuncion, valor) {
  algunaFuncion(valor);
}

ejecutar(decir, "Hola");
```

Lee esto cuidadosamente! Lo que estamos haciendo aquí es, nosotros pasamos la función *decir()* como el primer parámetro de la función *ejecutar*. No el valor de retorno de *decir*, sino que *decir()* misma!
Entonces, *decir* se convierte en la variable local *algunaFuncion* dentro de *ejecutar*, y ejecutar puede llamar a la función en esta variable usando *algunaFuncion()* (agregando llaves).

Por supuesto, dado que *decir* toma un parámetro, *ejecutar* puede pasar tal parámetro cuando llama a *algunaFuncion*.

Nosotros podemos, tal como lo hicimos, pasar una función por su nombre como parámetro a otra función. Pero no estamos obligados a tener que definir la función primero y luego pasarla. Podemos también definir y pasar la función como un parámetro a otra función todo al mismo tiempo:

```
function ejecutar(algunaFuncion, valor) {
  algunaFuncion(valor);
}

ejecutar(function(palabra){ console.log(palabra) }, "Hola");
```

(N.del T.: *function* es una palabra clave de JavaScript).
Nosotros definimos la función que queremos pasar a *ejecutar* justo ahí en el lugar donde *ejecutar* espera su primer parámetro.

De esta manera, no necesitamos darle a la función un nombre, por lo que esta función es llamada *función anónima*.

Esta es una primera ojeada a lo que me gusta llamar JavaScript "avanzado". Pero tomémoslo paso a paso. Por ahora, aceptemos que en JavaScript, nosotros podemos pasar una función como un parámetro cuando llamamos a otra función. Podemos hacer esto asignando nuestra función a una variable, la cual luego pasamos, o definiendo la función a pasar en el mismo lugar.

De Qué manera el pasar funciones hace que nuestro servidor HTTP funcione

Con este conocimiento, Volvamos a nuestro servidor HTTP minimalista:

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

A estas alturas, debería quedar claro lo que estamos haciendo acá: Estamos pasándole a la función *createServer* una función anónima.
Podemos llegar a lo mismo refactorizando nuestro código así:

003

```
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);
```

Quizás ahora es un buen momento para preguntar: ¿Por Qué estamos haciendo esto de esta manera?

Callbacks Manejadas por Eventos

La respuesta a) No es algo fácil de explicar (al menos para mí), y b) Yace en la naturaleza misma de como Node.js trabaja: Está orientado al evento, esa es la razón de por qué es tan rápido.

Podrías tomarte un tiempo para leer este excelente post (en inglés) de Felix Geisendördfer: Understanding node.js para alguna explicación de trasfondo.

Al final todo se reduce al hecho que Node.js trabaja orientado al evento. Ah, y sí, yo tampoco sé exactamente qué significa eso. Pero voy a hacer un intento de explicar, el por qué esto tiene sentido para nosotros, que queremos escribir aplicaciones web en Node.js.

Cuando nosotros llamamos al método *http.createServer*, por supuesto que no sólo queremos que el servidor se quede escuchando en algún puerto, sino que también queremos hacer algo cuando hay una petición HTTP a este servidor.

El problema es, que esto sucede de manera asíncrona: Puede suceder en cualquier momento, pero solo tenemos un único proceso en el cual nuestro servidor corre.

Cuando escribimos aplicaciones PHP, esto no nos molesta en absoluto: cada vez que hay una petición HTTP, el servidor web (por lo general Apache) genera un nuevo proceso solo para esta petición, y empieza el script PHP indicado desde cero, el cual es ejecutado de principio a fin.

Así que respecto al control de flujo, estamos en el medio de nuestro programa en Node.js, cuando una nueva petición llega al puerto 8888: ¿Cómo manipulamos esto sin volvernos locos?

Bueno, esta es la parte donde el diseño orientado al evento de Node.js / JavaScript de verdad ayuda, aunque tengamos que aprender nuevos conceptos para poder dominarlo. Veamos como estos conceptos son aplicados en nuestro código de servidor.

Nosotros creamos el servidor, y pasamos una función al método que lo crea. Cada vez que nuestro servidor recibe una petición, la función que le pasamos será llamada.

No sabemos qué es lo que va a suceder, pero ahora tenemos un lugar donde vamos a poder manipular la petición entrante. Es la función que pasamos, sin importar si la definimos o si la pasamos de manera anónima.

Este concepto es llamado un *callback* (N. del T.: del inglés: call = llamar; y back = de vuelta). Nosotros pasamos una función a algún método, y el método ocupa esta función para llamar (call) de vuelta (back) si un evento relacionado con este método ocurre.

Al menos para mí, esto tomó algún tiempo para ser entendido. Lee el articulo del blog de Felix de nuevo si todavía no te sientes seguro.

Juguemos un poco con este nuevo concepto. ¿Podemos probar que nuestro código continúa después de haber creado el servidor, incluso si no ha sucedido ninguna petición HTTP y la función callback que pasamos no ha sido llamada? Probemos:

004

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

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

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

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

Noten que utilizo *console.log* para entregar un texto cada vez que la función *onRequest* (nuestro callback) es gatillada, y otro texto *después* de iniciar nuestro servidor HTTP.
Cuando iniciamos esta aplicación (con *node server.js*, como siempre). Esta inmediatamente escribirá en pantalla "Servidor Iniciado" en la línea de comandos. Cada vez que hagamos una petición a nuestro servidor (abriendo http://localhost:8888/ en nuestro browser), el mensaje "Peticion Recibida." va a ser impreso en la línea de comandos.

Esto es JavaScript del lado del servidor asíncrono y orientado al evento con callbacks en acción :-)

(Toma en cuenta que nuestro servidor probablemente escribirá "Petición Recibida." a STDOUT dos veces al abrir la página en un browser. 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/).

Como nuestro Servidor manipula las peticiones

OK, analicemos rápidamente el resto del código de nuestro servidor, esto es, el cuerpo de nuestra función de callback *onRequest()*.

Cuando la callback es disparada y nuestra función *onRequest()* es gatillada, dos parámetros son pasados a ella: *request* y *response*.

Estos son objetos, y puedes usar sus métodos para manejar los detalles de la petición HTTP ocurrida y responder a la petición (en otras palabras enviar algo de vuelta al browser que hizo la petición a tu servidor).

Y eso es lo que nuestro código hace: cada vez que una petición es recibida, usa la función *response.writeHead()* para enviar un estatus HTTP 200 y un content-type (parámetro que define que tipo de contenido es) en el encabezado de la respuesta HTTP, y la función *response.write()* para enviar el texto "Hola Mundo" en el cuerpo de la respuesta.

Por último, nosotros llamamos *response.end()* para finalizar nuestra respuesta.

Hasta el momento, no nos hemos interesado por los detalles de la petición, y ese es el por qué no hemos ocupado el objeto *request* completamente.

Encontrando un lugar para nuestro módulo de servidor

OK, prometí que volveríamos a al Cómo organizar nuestra aplicación. Tenemos el código de nuestro servidor HTTP muy básico en el archivo *server.js*, y mencioné que es común tener un archivo principal llamado *index.js*, el cual es usado para arrancar y partir nuestra aplicación haciendo uso de los otros módulos de la aplicación (como el módulo de servidor HTTP que vive en *server.js*).

Hablemos de como podemos hacer que nuestro server.js sea un verdadero módulo Node.js y que pueda ser usado por nuestro pronto-a-ser-escrito archivo principal *index.js*.

Como habrán notado, ya hemos usado módulos en nuestro código, como éste:

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

...

http.createServer(...);
```

En algún lugar dentro de Node.js vive un módulo llamado "http", y podemos hacer uso de éste en nuestro propio código requiriéndolo y asignando el resultado del requerimiento a una variable local. Esto transforma a nuestra variable local en un objeto que acarrea todos los métodos públicos que el módulo *http* provee.

Es práctica común elegir el nombre del módulo como nombre para nuestra variable local, pero somos libres de escoger cualquiera que nos guste:

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

...

foo.createServer(...);
```

Bien. Ya tenemos claro como hacer uso de los módulos internos de Node.js. ¿Cómo hacemos para crear nuestros propios módulos, y Cómo los utilizamos?
Descubrámoslo transformando nuestro script *server.js* en un módulo real.

Sucede que, no tenemos que transformarlo tanto. Hacer que algún código sea un Módulo, significa que necesitamos *exportar* las partes de su funcionalidad que queremos proveer a otros scripts que requieran nuestro módulo.

Por ahora, la funcionalidad que nuestro servidor HTTP necesita exportar es simple: Permitir a los scripts que utilicen este módulo arrancar el servidor.

Para hacer esto posible, dotaremos al código de nuestro servidor de una función llamada *inicio*, y exportaremos esta función:

005

```
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;
```

De este modo, Podemos crear nuestro propio archivo principal *index.js*, y arrancar nuestro servidor HTTP allí, aunque el código para el servidor este en nuestro archivo *server.js*.
Crea un archivo *index.js* con el siguiente contenido:

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

server.iniciar();
```

Como puedes ver, nosotros utilizamos nuestro módulo de servidor tal como cualquier otro módulo interno: requiriendo el archivo donde está contenido y asignándolo a una variable, con las funciones que tenga 'exportadas' disponibles para nosotros.
Eso es. Podemos ahora arrancar nuestra aplicación por medio de nuestro script principal, y va a hacer exactamente lo mismo:

```
node index.js
```

Bien, ahora podemos poner las diferentes partes de nuestra aplicación en archivos diferentes y enlazarlas juntas a través de la creación de estos módulos.

Tenemos sólo la primera parte de nuestra aplicación en su lugar: Podemos recibir peticiones HTTP. Pero necesitamos hacer algo con ellas - necesitamos reaccionar de manera diferente, dependiendo de que URL el browser requiera de nuestro servidor.

Para una aplicación muy simple, podrías hacer esto directamente dentro de una función de callback *OnRequest()*. Pero, como dije, agreguemos un poco más de abstracción, de manera de hacer nuestra aplicación más interesante.

Hacer diferentes peticiones HTTP ir a partes diferentes de nuestro código se llama "ruteo" (N. del T.: routing, en inglés) - bueno, entonces, creemos un módulo llamado *router*.

¿Qué se necesita para "rutear" peticiones?

Necesitamos ser capaces de entregar la URL requerida y los posibles parámetros GET o POST adicionales a nuestro router, y basado en estos, el router debe ser capaz de decidir qué código ejecutar (este "código a ejecutar" es la tercera parte de nuestra aplicación: una colección de manipuladores de peticiones que harán el verdadero trabajo cuando una petición es recibida).

Así que, Necesitamos mirar en las peticiones HTTP y extraer la URL requerida, así como los parámetros GET/POST de ellos. Se puede discutir acerca de si este procedimiento debe ser parte del router o del servidor (o si lo hacemos un módulo por sí mismo), pero hagamos el acuerdo de hacerlo parte de nuestro servidor HTTP por ahora.

Toda la información que necesitamos está disponible en el objeto *request*, el que es pasado como primer parámetro a nuestra función callback *onRequest()*. Pero para interpretar esta información, necesitamos algunos módulos adicionales Node.js, llamados *url* y *querystring*.

El módulo *url* provee métodos que nos permite extraer las diferentes partes de una URL (como por ejemplo la ruta requerida y el string de consulta), y *querystring* puede, en cambio, ser usado para parsear el string de consulta para los parámetros requeridos:

```
url.parse(string).query
                                   |
           url.parse(string).pathname      |
                       |                   |
                       |                   |
                     ------ -------------------
http://localhost:8888/iniciar?foo=bar&hello=world
                                 ---       -----
                                  |          |
                                  |          |
              querystring(string)["foo"]    |
                                            |
                          querystring(string)["hello"]
```

Podemos, por supuesto, también utilizar *querystring* para parsear el cuerpo de una petición POST en busca de parámetros, como veremos más tarde.

Agreguemos ahora a nuestra función *onRequest()* la lógica requerida para encontrar que ruta URL el browser solicitó:

006

```
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;
```

Muy Bien. Nuestra aplicación puede ahora distinguir peticiones basadas en la ruta URL requerida - esto nos permite mapear peticiones hacia nuestro manipuladores de peticiones, basándonos en la ruta URL usando nuestro (pronto a ser escrito) router. Luego, podemos construir nuestra aplicación en una forma REST (N. del T.: RESTful way en Inglés), ya que ahora podemos implementar una interfaz que sigue los principios que guían a la *Identificación de Recursos* (ve por favor el artículo de Wikipedia acerca de la Transferencia del Estado Representacional para información de trasfondo. En el contexto de nuestra aplicación, esto significa simplemente que seremos capaces de tener peticiones para las URLs */iniciar* y */subir* manejadas por partes diferentes de nuestro código. Veremos pronto como todo esto encaja.

OK, es hora de escribir nuestro router. Vamos a crear un nuevo archivo llamado *router.js*, con el siguiente contenido:

007

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

exports.route = route;
```

Por supuesto, este código no está haciendo nada, pero eso está bien por ahora. Empecemos a ver como vamos a encajar este router con nuestro servidor antes de poner más lógica en el router.
Nuestro servidor HTTP necesita saber y hacer uso de nuestro router. Podemos escribir directamente esta dependencia a nuestro servidor, pero como hemos aprendido de la manera difícil en nuestras

experiencias, vamos a acoplar de manera débil (*loose coupling* en Inglés) al router y su servidor vía inyección por dependencia. Para una referencia de fondo, leer el Artículo de Martin Fowler (en Inglés).

Primero extendamos nuestra función *iniciar()* de manera de permitirnos pasar la función de ruteo 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("Peticion 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 extendamos nuestro *index.js* adecuadamente, esto es, inyectando la función de ruteo de nuestro router en el servidor:

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

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

Nuevamente , estamos pasando una función como parámetros, pero esto ya no es una novedad para nosotros.
Si arrancamos nuestra aplicación ahora (*node index.js* como siempre), y hacemos una petición para una URL, puedes ver ahora por las respuestas de la aplicación que nuestro servidor HTTP hace uso de nuestro router y le entrega el nombre de ruta requerido:

```
bash$ node index.js
Petición para /foo recibida.
A punto de rutear una peticion para /foo
```

He omitido la molesta respuesta de la petición para /favicon.ico

Ejecución en el reino de los verbos

¿Puedo divagar un vez más por un momento y hablar acerca de la programación funcional de nuevo?
Pasar funciones no es sólo una consideración técnica. Con respecto al diseño de software, esto es casi filosófico. Tan solo piensa en ello: en nuestro archivo de index, podríamos haber entregado el objeto *router* al servidor, y el servidor hubiese llamado a la función *route* de este objeto.
De esta manera, podríamos haber pasado una *cosa*, y el servidor hubiese usado esa cosa para *hacer* algo. Oye, "Cosa Router", ¿Podrías por favor rutear esto por mí?
Pero el servidor no necesita la cosa. Sólo necesita *hacer algo*, y para que algo se haga, no necesitas cosas para nada, sólo necesitas *acciones*. No necesitas *sustantivos*, sino que necesitas *verbos*.
Entender este cambio de mentalidad fundamental que está en el núcleo de esta idea es lo que realmente me hizo entender la programación funcional.
Y lo entendí mientras leía la obra maestra de Steve Yegge (en Inglés) Ejecución en el Reino de los Sustantivos. Anda, léela, por favor. Es uno de los mejores artículos relacionados con el software que haya tenido el placer de encontrar.

Ruteando a los verdaderos manipuladores de peticiones

Volviendo al tema. Nuestro servidor HTTP y nuestro router de peticiones son ahora los mejores amigos y conversan entre ellos, tal y como pretendimos.
Por supuesto, esto no es suficiente, "Rutear" significa que nosotros queremos manipular las peticiones a distintas URLs de manera, diferente. Nos gustaría tener la "lógicas de negocios" para peticiones de */inicio* manejadas en otra función, distinta a la que maneja las peticiones para */subir*.
Por ahora, el ruteo "termina" en el router, y el router no es el lugar donde se está "haciendo algo" con las peticiones, ya que esto no escalaría bien una vez que nuestra aplicación se haga más compleja.
Llamemos a estas funciones, donde las peticiones están siendo ruteadas, *manipuladores de peticiones* (ó request handlers). Y procedamos con éstos ahora, porque, a menos que no los tengamos en su lugar, no tiene mucho sentido en hacer nada con el router por ahora.
Nueva parte de la aplicación, significa nuevo módulo - no creo que haya sorpresa acá. Creemos un módulo llamado requestHandlers (N. del T.: por manipuladores de petición), agreguemos una función de ubicación para cada manipulador de petición, y exportemos estos como métodos para el módulo:

008

```
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.
Llegado a este punto, necesitamos tomar una decisión: ¿Ingresaremos las rutas del módulo requestHandlers dentro del código del router (hard-coding), o queremos algo más de dependencia por inyección? Aunque en la dependencia por inyección, como cualquier otro patrón, no debería ser usada simplemente por usarla, en este caso tiene sentido acoplar el router débilmente a sus manipuladores de petición, así, de esta manera hacemos que el router sea reutilizable.

Esto significa que necesitamos pasar los manipuladores de petición desde nuestro server al router, pero esto se siente equivocado, dado que, ¿Por qué tenemos que hacer el camino largo y entregar los manipuladores desde el archivo principal al servidor y de ahí al router?

¿Cómo vamos a pasarlos? Ahora tenemos sólo dos manipuladores, pero en una aplicación real, este número se va a incrementar y variar, y nosotros no queremos estar a cada momento mapeando peticiones a manipuladores cada vez que una nueva URL o manipulador de petición sea agregado. Y si tenemos un código del tipo *if peticion == x then llama manipulador y* en el router, esto se pondría cada vez más feo.

¿Un número variable de items, cada uno de ellos mapeados a un string? (en este caso la URL requerida) Bueno, esto suena como que un array asociativo haría el truco.

Bueno, este descubrimiento es obscurecido por el hecho que JavaScript no provee arrays asociativos - ¿o sí? !Resulta que lo que necesitamos usar son objetos si necesitamos un array asociativo!

Una buena introducción a esto está (en Inglés) en http://msdn.microsoft.com/en-us/magazine/cc163419.aspx, Déjame citarte la parte relevante:

> En C++ o C#, cuando hablamos acerca de objetos, nos estamos refiriendo a instancias de clases de estructuras. Los objetos tienen distintas propiedades y métodos, dependiendo en las plantillas (esto es, las clases) desde donde éstos sean instanciados. Este no es el caso con los objetos de JavaScript. En JavaScript, los objetos son sólo colecciones de pares nombre/valor - piensa en un objeto JavaScript como en un diccionario con llaves de string.

Si los objetos JavaScript son sólo colecciones de pares nombre/valor, ¿Cómo pueden entonces tener métodos? Bueno, los valores pueden ser strings, números, etc... ¡O Funciones!

OK, ahora, volviendo finalmente al código. Hemos decidido que queremos pasar la lista de requestHandlers (manipuladores de petición) como un objeto, y para lograr este acoplamiento débil, necesitamos usar la técnica de inyectar este objeto en la *route()* (ruta).

Empecemos con poner el objeto en nuestro archivo principal *index.js*:

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

(N. del T.: Se Opta por dejar los verbos en Inglés 'route' para rutear y 'handle' para manipular). Aunque *handle* es más una "cosa" (una colección de manipuladores de petición), propongo que lo nombremos como un verbo, ya que esto resultará en una expresión fluida en nuestro router, como veremos a continuación:
Como puedes ver, es realmente simple mapear diferentes URLs al mismo manipulador de peticiones: Mediante la adición de un par llave/valor de *"/"* y *requestHandlers.iniciar*, podemos expresar en una forma agradable y limpia que no sólo peticiones a */start*, sino que también peticiones a */* pueden ser manejadas por el manipulador *inicio*.

Después de definir nuestro objeto, se lo pasamos al servidor como un parámetro adicional. Modifiquemos nuestro *server.js* para hacer uso de este:

```
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("Peticion para " + pathname + " recibida.");

    route(pathname, handle);

    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 chequear si un manipulador de peticiones para una ruta dada existe, y si es así, simplemente llamamos a la función adecuada. Dado que podemos acceder a nuestras funciones manipuladoras de petición desde nuestro objeto de la misma manera que hubiésemos podido acceder a un elemento de un array asociativo, es que tenemos la expresión fluida *handle[pathname] ();* de la que hablé antes, que en otras palabras es: "Por favor, *handle* (maneja) este(a) *pathname* (ruta)".
Bien, ¡Esto es todo lo que necesitamos para atar servidor, router y manipuladores de peticiones juntos! Una vez que arranquemos nuestra aplicación y hagamos una petición en nuestro browser de

http://localhost:8888/iniciar, vamos a probar que el manipulador de petición correcto fue, de hecho, llamado:

```
Servidor Iniciado.
Peticion para /iniciar recibida.

A punto de rutear una petición para /iniciar
Manipulador de peticion 'iniciar' ha sido llamado.
```

Haciendo que los Manipuladores de Peticiones respondan

Muy bien. Ahora, si tan solo los manipuladores de petición pudieran enviar algo de vuelta al browser, esto sería mucho mejor, ¿cierto?

Recuerda, que el "Hola Mundo" que tu browser despliega ante una petición de una página, aún viene desde la función *onRequest* en nuestro archivo *server.js*.

"Manipular Peticiones" no significa otra cosa que "Responder a las Peticiones" después de todo, así que necesitamos empoderar a nuestros manipuladores de peticiones para hablar con el browser de la misma manera que la función *onRequest* lo hace.

¿Cómo no se debe hacer esto?

La aproximación directa que nosotros - desarrolladores con un trasfondo en PHP o Ruby - quisiéramos seguir es de hecho conducente a errores: trabaja de manera espectacular al principio y parece tener mucho sentido, y de pronto, las cosas se arruinan en el momento menos esperado.

A lo que me refiero con "aproximación directa" es esto: hacer que los manipuladores de petición retornen - *return()* - el contenido que ellos quieran desplegar al usuario, y luego, enviar esta data de respuesta en la función *onRequest* de vuelta al usuario.

Tan sólo hagamos esto, y luego, veamos por qué esto no es tan buena idea.

Empecemos con los manipuladores de petición y hagámoslos retornar, lo que nosotros queremos desplegar en el browser. Necesitamos modificar *requestHandlers.js* a lo siguiente:

009

```
function iniciar() {
  console.log("Manipulador de peticion 'iniciar' fue llamado.");
  return "Hola Iniciar";
}

function subir() {
  console.log("Manipulador de peticion 'subir' fue llamado.");
  return "Hola Subir";
}

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

Bien. De todas maneras, el router necesita retornar al servidor lo que los manipuladores de petición le retornaron a él. Necesitamos entonces editar *router.js* de esta manera:

```
function route(handle, pathname) {
  console.log("A punto de rutear una peticion para " + pathname);
  if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
  } else {
    console.log("No se encontro manipulador para " + pathname);
    return "404 No Encontrado";
  }
}

exports.route = route;
```

Como puedes ver, nosotros también retornaremos algún texto si la petición no es ruteada.
Por último, pero no menos importante, necesitamos refactorizar nuestro servidor para hacerlo responder al browser con el contenido que los manipuladores de petición le retornaron vía el router, transformando de esta manera a *server.js* en:

```
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("Peticion para " + pathname + " recibida.");

    response.writeHead(200, {"Content-Type": "text/html"});
    var content = route(handle, pathname)
    response.write(content);
    response.end();
  }

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

exports.iniciar = iniciar;
```

Si nosotros arrancamos nuestra aplicación re-escrita, todo va a funcionar a las mil maravillas: hacerle una petición a http://localhost:8888/iniciar resulta en "Hola Iniciar" siendo desplegado en el browser, hacerle una petición a http://localhost:8888/subir nos da "Hola Subir", y la petición a http://localhost:8888/foo produce "404 No Encontrado".
OK, entonces ¿Por qué esto es un problema? La respuesta corta es: debido a que si uno de los manipuladores de petición quisiera hacer uso de una operación *no-bloqueante* (non-blocking) en el futuro, entonces esta configuración, como la tenemos, sería problemática.

Tomémosnos algún tiempo para la respuesta larga.

Bloqueante y No-Bloqueante

Como se dijo, los problemas van a surgir cuando nosotros incluyamos operaciones no-bloqueantes en los manipuladores de petición. Pero hablemos acerca de las operaciones bloqueantes primero, luego, acerca de las operaciones no-bloqueantes.

Y, en vez de intentar explicar que es lo que significa "bloqueante" y "no bloqueante", demostremos nosotros mismo que es lo que sucede si agregamos una operación bloqueante a nuestros manipuladores de petición.

Para hacer esto, modificaremos nuestro manipulador de petición *iniciar* para hacer una espera de 10 segundos antes de retornar su string "Hola Iniciar". Ya que no existe tal cosa como *sleep()* en JavaScript, usaremos un hack ingenioso para ello.

Por favor, modifica *requestHandlers.js* como sigue:

010

```
function iniciar() {
  console.log("Manipulador de peticion 'iniciar' fue llamado.");

  function sleep(milliSeconds) {
    // obten la hora actual
    var startTime = new Date().getTime();
    // atasca la cpu
    while (new Date().getTime() < startTime + milliSeconds);
  }

  sleep(10000);
  return "Hola Iniciar";
}

function subir() {
  console.log("Manipulador de peticion 'subir' fue llamado.");
  return "Hola Subir";
}

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

Dejemos claros que es lo que esto hace: cuando la función *iniciar()* es llamada, Node.js espera 10 segundos y sólo ahí retorna "Hola Iniciar". Cuando está llamando a *subir()*, retorna inmediatamente, la misma manera que antes.
(Por supuesto la idea es que te imagines que, en vez de dormir por 10 segundos, exista una operación bloqueante verdadera en *iniciar()*, como algún tipo de calculo de largo aliento.)

Veamos qué es lo que este cambio hace.

Como siempre, necesitamos reiniciar nuestro servidor. Esta vez, te pido sigas un "protocolo" un poco más complejo de manera de ver que sucede: Primero, abre dos ventanas de browser o tablas.

En la primera ventana, por favor ingresa http://localhost:8888/iniciar en la barra de direcciones, pero no abras aún esta url!

En la barra de direcciones de la segunda ventana de browser, ingresa http://localhost:8888/subir y, nuevamente, no presiones enter todavía.

Ahora, haz lo siguiente: presiona enter en la primera ventana ("/iniciar"), luego, rápidamente cambia a la segunda ventana ("/subir") y presiona enter, también.

Lo que veremos será lo siguiente: La URL /inicio toma 10 segundos en cargar, tal cual esperamos. Pero la URL /subir *también* toma 10 segundos para cargar, ¡Aunque no hay definido un *sleep()* en el manipulador de peticiones correspondiente!

¿Por qué? simple, porque *inicio()* contiene una operación bloqueante. En otras palabras "Está bloqueando el trabajo de cualquier otra cosa".

He ahí el problema, porque, el dicho es: *"En Node.js, todo corre en paralelo, excepto tu código"*.

Lo que eso significa es que Node.js puede manejar un montón de temas concurrentes, pero no lo hace dividiendo todo en hilos (threads) - de hecho, Node.js corre en un sólo hilo. En vez de eso, lo hace ejecutando un loop de eventos, y nosotros, los desarrolladores podemos hacer uso de esto - Nosotros debemos evitar operaciones bloqueantes donde sea posible, y utilizar operaciones no-bloqueantes en su lugar.

Lo que *exec()* hace, es que, ejecuta un comando de shell desde dentro de Node.js. En este ejemplo, vamos a usarlo para obtener una lista de todos los archivos del directorio en que nos encontramos ("ls -lah"), permitiéndonos desplegar esta lista en el browser de un usuario que este peticionando la URL */inicio*.

Lo que el código hace es claro: crea una nueva variable *content()* (con el valor incial de "vacío"), ejecuta "ls -lah", llena la variable con el resultado, y lo retorna.

Como siempre, arrancaremos nuestra aplicación y visitaremos http://localhost:8888/iniciar.

Lo que carga una bella página que despliega el string "vacío". ¿Qué es lo que está incorrecto acá?

Bueno, como ya habrán adivinado, *exec()* hace su magia de una manera no-bloqueante. Buena cosa esto, porque de esta manera podemos ejecutar operaciones de shell muy caras en ejecución (como, por ejemplo, copiar archivos enormes o cosas similares) sin tener que forzar a nuestra aplicación a detenerse como lo hizo la operación *sleep*.

(Si quieres probar esto, reemplaza "ls -lah" con una operación más cara como "find /").

Pero no estaríamos muy felices si nuestra elegante aplicación no bloqueante no desplegara algún resultado, ¿cierto?.

Bueno, entonces, arreglémosla. Y mientras estamos en eso, tratemos de entender por qué la arquitectura actual no funciona.

El problema es que *exec()*, para poder trabajar de manera no-bloqueante, hace uso de una función de callback.

En nuestro ejemplo, es una función anónima, la cual es pasada como el segundo parámetro de la llamada a la función *exec()*:

011

```
var exec =  require('child_process').exec;

function iniciar() {
  console.log("Manipulador de peticion 'iniciar' fue llamado.");

  function sleep(milliSeconds) {
    var startTime = new Date().getTime();
    while (new Date().getTime() < startTime + milliSeconds);
  }

  sleep(10000);

  exec('ls -lah', function(err, stdout, stderr) {
        console.log(stdout);//content = stdout;
  });
  return "Hola Iniciar";


}

function subir() {
  console.log("Manipulador de peticion 'subir' fue llamado.");
  return "Hola Subir";
}

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

Y aquí yace la raíz de nuestro problema: nuestro código es ejecutado de manera sincrónica, lo que significa que inmediatamente después de llamar a *exec()*, Node.js continúa ejecutando *return content;*. En este punto, *content* todavía está vacío, dado el hecho que la función de callback pasada a *exec()* no ha sido aún llamada - porque *exec()* opera de manera asíncrona.
Ahora, "ls -lah" es una operación sencilla y rápida (a menos, claro, que hayan millones de archivos en el directorio). Por lo que es relativamente necesario llamar al callback - pero de todas maneras esto sucede de manera asíncrona.

Esto se hace más obvio al tratar con un comando más costoso: "find /" se toma un minuto en mi maquina, pero si reemplazo "ls -lah" con "find /" en el manipulador de peticiones, yo recibo inmediatamente una respuesta HTTP cuando abro la URL /inicio - está claro que *exec()* hace algo en el trasfondo, mientras que Node.js mismo continúa con el flujo de la aplicación, y podemos asumir que la función de callback que le entregamos a *exec()* será llamada sólo cuando el comando "find /" haya terminado de correr.

Pero, ¿cómo podemos alcanzar nuestra meta, la de mostrarle al usuario una lista de archivos del directorio actual?

Bueno, después de aprender como *no* hacerlo, discutamos cómo hacer que nuestros manipuladores de petición respondan a los requerimientos del browser de la manera correcta.

Respondiendo a los Manipuladores de Petición con Operaciones No Bloqueantes

Acabo de usar la frase "la manera correcta". Cosa peligrosa. Frecuentemente, no existe una única "manera correcta".

Pero una posible solución para esto, frecuente con Node.js es pasar funciones alrededor. Examinemos esto.

Ahora mismo, nuestra aplicación es capaz de transportar el contenido desde los manipuladores de petición al servidor HTTP retornándolo hacia arriba a través de las capas de la aplicación (manipulador de petición -> router -> servidor).

Nuestro nuevo enfoque es como sigue: en vez de llevar el contenido al servidor, llevaremos el servidor al contenido. Para ser más precisos, inyectaremos el objeto *response* (respuesta) (desde nuestra función de callback de servidor *onRequest()*) 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.

Suficientes explicaciones, aquí hay una receta paso a paso de como cambiar nuestra aplicación.

Empecemos con nuestro servidor, *server.js*:

012

```
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("Peticion para " + pathname + " recibida.");

    route(handle, pathname, response);
  }

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

exports.iniciar = iniciar;
```

En vez de esperar un valor de respuesta desde la función *route()*, pasémosle un tercer parámetro: nuestro objeto *response*. Es más, removamos cualquier llamada a *response* desde el manipulador *onRequest()*, ya que ahora esperamos que *route* se haga cargo de esto.
Ahora viene *router.js*:

```
function route(handle, pathname, response) {
  console.log("A punto de rutear una peticion para " + pathname);
  if (typeof handle[pathname] === 'function') {
```

```
    handle[pathname](response);
  } else {
    console.log("No hay manipulador de peticion para " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 No Encontrado");
    response.end();
  }
}

exports.route = route;
```

Mismo patrón: En vez de esperar que retorne un valor desde nuestros manipuladores de petición, nosotros traspasamos el objeto *response*.
Si no hay manipulador de petición para utilizar, ahora nos hacemos cargo de responder con adecuados encabezado y cuerpo "404".

Y por último, pero no menos importante, modificamos a *requestHandlers.js* (el archivo de manipuladores de petición).

```
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;
```

Nuestras funciones manipuladoras necesitan aceptar el parámetro de respuesta *response*, y de esta manera, hacer uso de él de manera de 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*.

Si arrancamos nuestra aplicación de nuevo (*node index.js*), esto debería funcionar de acuerdo a lo esperado.

Si quieres probar que la operación cara dentro de */iniciar* no bloqueará más las peticiones para */subir* que sean respondidas inmediatamente, entonces modifica tu archivo *requestHandlers.js* como sigue:

013

```
var exec = require("child_process").exec;

function iniciar(response) {
  console.log("Manipulador de petición 'iniciar' fue llamado.");

  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    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;
```

Esto hará que las peticiones HTTP a http://localhost:8888/iniciar tomen al menos 10 segundos, pero, las peticiones a http://localhost:8888/subir sean respondidas inmediatamente, incluso si */iniciar* todavía está en proceso.

Sirviendo algo útil

Hasta ahora, lo que hemos hecho es todo simpático y bonito, pero no hemos creado aún valor para los clientes de nuestro sitio web ganador de premios.

Nuestro servidor, router y manipuladores de petición están en su lugar, así que ahora 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, subir este archivo, y ver el archivo subido en el browser. Por simplicidad asumiremos que sólo los archivos de imagen van a ser subidos y desplegados a través de la aplicación.

OK, veámoslo paso a paso, pero ahora, con la mayoría de las técnicas y principios de JavaScript explicadas, acelerémoslo un poco al mismo tiempo.

Aquí, paso a paso significa a grandes rasgos dos pasos: vamos a ver primero como manejar peticiones POST entrantes (pero no subidas de archivos), y en un segundo paso, haremos uso de un modulo externo de Node.js para la manipulación de subida de archivos. He escogido este alcance por dos razones:

Primero, manejar peticiones POST básicas es relativamente simple con Node.js, pero aún nos enseña lo suficiente para que valga la pena ejercitarlo.

Segundo, manejar las subidas de archivos (i.e. peticiones POST multiparte) *no es* simple con Node.js, consecuentemente está más allá del alcance de este tutorial, pero el aprender a usar un modulo externo es una lección en sí misma que tiene sentido de ser incluida en un tutorial de principiantes.

Manejando Peticiones POST

Mantengamos esto ridículamente simple: presentaremos un área de texto que pueda ser llenada por el usuario y luego enviada al servidor en una petición POST. Una vez recibida y manipulada esta petición, desplegaremos el contenido del área de texto.

El HTML para el formulario de esta área de texto necesita ser servida por nuestro manipulador de petición */iniciar*, así que agreguémoslo de inmediato. En el archivo *requestHandlers.js*:

014

```
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;
```

Ahora, si esto no va a ganar los Webby Awards, entonces no se que podría. Haciendo la petición http://localhost:8888/iniciar en tu browser, deberías ver un formulario muy simple. Si no, entonces probablemente no has reiniciado la aplicación.
Te estoy escuchando: tener contenido de vista justo en el manipulador de petición es feo. Sin embargo, he decidido no incluir ese nivel extra de abstracción (esto es, separar la lógica de vista y controlador) en este tutorial, ya que pienso que es no nos enseña nada que valga la pena saber en el contexto de JavaScript o Node.js.

Mejor usemos el espacio que queda en pantalla para un problema más interesante, esto es, manipular la petición POST que dará con nuestro manipulador de petición */subir* cuando el usuario envíe este formulario.

Ahora que nos estamos convirtiendo en "novicios expertos", ya no nos sorprende el hecho que manipular información de POST este hecho de una manera no bloqueante, mediante el uso de llamadas asíncronas.

Lo que tiene sentido, ya que las peticiones POST pueden ser potencialmente muy grandes - nada detiene al usuario de introducir texto que tenga muchos megabytes de tamaño. Manipular este gran volumen de información de una vez puede resultar en una operación bloqueante.

Para hacer el proceso completo no bloqueante. Node.js le entrega a nuestro código la información POST en pequeños trozos con callbacks que son llamadas ante determinados eventos. Estos eventos son *data* (un nuevo trozo de información POST ha llegado) y *end* (todos los trozos han sido recibidos).

Necesitamos decirle a Node.js que funciones llamar de vuelta cuando estos eventos ocurran. Esto es hecho agregando *listeners* (N. del T.: del verbo *listen* - escuchar) al objeto de petición (*request*) que es pasado a nuestro callback *onRequest* cada vez que una petición HTTP es recibida.

Esto básicamente luce así:

```
request.addListener("data", function(chunk) {
    // funcion llamada cuando un nuevo trozo (chunk)
    // de informacion (data) es recibido.
});

request.addListener("end", function() {
    // funcion llamada cuando todos los trozos (chunks)
    // de informacion (data) han sido recibidos.
});
```

La pregunta que surge es dónde implementar ésta lógica. Nosotros sólo podemos acceder al objeto *request* en nuestro servidor - no se lo estamos pasando al router o a los manipuladores de petición, como lo hicimos con el objeto *response*.
En mi opinión, es un trabajo del servidor HTTP de darle a la aplicación toda la información de una petición que necesite para hacer su trabajo. Luego, sugiero que manejemos el procesamiento de la petición de POST en el servidor mismo y pasemos la información final al router y a los manipuladores de petición, los que luego decidirán que hacer con ésta.

Entonces, la idea es poner los callbacks *data* y *end* en el servidor, recogiendo todo los trozos de información POST en el callback *data*, y llamando al router una vez recibido el evento *end*, mientras le entregamos los trozos de información recogidos al router, el que a su vez se los pasa a los manipuladores de petición.

Aquí vamos, empezando con *server.js*:

015

```
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("Peticion 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;
```

Básicamente hicimos tres cosas aquí: primero, definimos que esperamos que la codificación de la información recibida sea UTF-8, agregamos un listener de eventos para el evento "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. Además, pasamos la información POST al router, ya que la vamos a necesitar en nuestros manipuladores de eventos.
Agregar un logueo de consola cada vez que un trozo es recibido es una mala idea para código de producción (megabytes de información POST, ¿recuerdan?, pero tiene sentido para que veamos que pasa.

Mejoremos nuestra aplicación. En la página */subir*, desplegaremos el contenido recibido. Para hacer esto posible, necesitamos pasar la *dataPosteada* a los manipuladores de petición, en *router.js*.

```
function route(handle, pathname, response, postData) {
  console.log("A punto de rutear una peticion 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;
```

Eso es, ahora somos capaces de recibir información POST y usarla en nuestros manipuladores de petición.
Una última cosa para este tema: lo que le estamos pasando al router y los manipuladores de petición es el cuerpo (*body*) de nuestra petición POST. Probablemente necesitemos consumir los campos individuales que conforman la información POST, en este caso, el valor del campo *text*.

Nosotros ya hemos leído acerca del módulo *querystring*, el que nos ayuda con esto:

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

function iniciar(response, postData) {
  console.log("Manipulador de peticion 'inicio' 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="Submit text" />'+
    '</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 el texto: : " +
    querystring.parse(dataPosteada)["text"]);
    response.end();
}

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

Bueno, para un tutorial de principiantes, esto es todo lo que diremos acerca de la información POST.

La próxima vez, hablaremos sobre como usar el excelente módulo *node-formidable* para permitirnos lograr nuestro caso de uso final: subir y desplegar imágenes.