Leanpub

el libro principiante de Node

Un tutorial comprensivo de Node.js

Manuel Kiessling
Obie Fernandez
Pablo y Debi Bontti

El Libro Principiante de Node

Un completo tutorial de Node.js

Obie Fernandez, Manuel Kiessling y Debi & Pablo Bontti

Este libro está a la venta en http://leanpub.com/node-principiante

Esta versión se publicó en 2014-11-15

¡Twitea sobre el libro!

Por favor ayuda a Obie Fernandez, Manuel Kiessling y Debi & Pablo Bontti hablando sobre el libro en Twitter!

El hashtag sugerido para este libro es #principianteNode.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

https://twitter.com/search?q=#principianteNode

Índice general

Aviso especial

Querido lector de *El Libro Principiante de Node*,

El propósito de este libro es nada más y nada menos que ayudarle a comenzar a desarrollar aplicaciones usando Node.js.

Muchos lectores han pedido un segundo libro, un libro que continúe donde *El Libro Principiante de Node* termina, permitiendoles zambullirse a lo profundo del desarrollo con Node.js, aprendiendo acerca de manejo de base de datos, frameworks, pruebas unitarias, y mucho más.

Recientemente comencé a trabajar en este libro. Se llama *El Libro Artesano de Node*, y está disponible vía Leanpub también.

Este nuevo libro es un ´trabajo en curso´, al igual que *El Libro Principiante de Node* lo fue cuando primero salió.

Mi idea es ampliar *El Libro Artesano de Node* constantemente, a la vez que recojo y uso reacciones de la comunidad.

Por lo tanto, decidí poner a la venta una primera versión de este nuevo libro lo antes posible.

Hasta ahora, *El Libro Artesano de Node* contiene los siguientes capítulos:

- Working with NPM and Packages
- Test-Driven Node.js Development
- Object-Oriented JavaScript

Ya tiene 28 páginas, y más capítulos se añadirán pronto.

El precio final del libro completo será $9.99, sin embargo como lector y comprador de *El Libro Principiante de Node*, tiene la oportunidad de comprar el nuevo libro mientras se completa, con un descuento del 50%, por solo $4.99.

De esta manera, tendrá las siguientes ventajas:

- Aprender: Conseguirá capítulos nuevos abarcando Node.js avanzado y temas de JavaScript tan pronto como se completan.
- Sugerir Ideas: ¿qué le gustaría a usted ver en la versión final del libro?
- Ahorrar: Solamente pagará $4.99 una vez, y recibirá todas las futuras actualizaciones *gratis*, incluyendo la versión final y completa del libro.

Si le gustaría aprovechar esta oferta, simplemente vaya a

http://leanpub.com/nodecraftsman[1]

y use el código de cupón *nodereader*.

.

[1] http://leanpub.com/nodecraftsman

Sobre este libro

El propósito de este documento es ayudarle a comenzar a desarrollar aplicaciones para Node.js y a la vez enseñarle todo lo que necesita saber acerca de JavaScript ¨avanzado¨. Va mucho más allá que un simple tutorial ¨Hola Mundo¨.

Status

Usted está leyendo la versión final de este libro, es decir, las únicas actualizaciones que se harán serán para corregir errores o reflejar cambios en nuevas versiones de Node.js. La última actualización se hizo el 1 de julio de 2013.

Los ejemplos de código en este libro se han probado para funcionar con Node.js versión 0.10.12.

A qué público está destinado

Este documento probablemente será más adecuado para lectores que tienen antecedentes parecidos a los míos: experimentados con por lo menos un lenguaje orientado a objetos como Ruby, Python, PHP o Java, poca experiencia con JavaScript, y totalmente nuevos a Node.js.

Al estar destinado a desarrolladores que ya tienen experiencia con otros lenguajes de programación, este documento no abarcará cosas muy básicas como tipos de datos, variables, estructuras de control, y así por el estilo. Usted necesitará tener conocimiento previo de estos para entender este documento.

Sin embargo, ya que las funciones y los objetos en JavaScript son diferentes a sus equivalentes en la mayoría de otros lenguajes, estos se explicarán en más detalle.

Estructura de este documento

Al terminar este documento, habrá creado una aplicación web completa que permite al usuario ver páginas web y subir archivos.

Por supuesto, no es algo que le cambiará la vida, pero iremos más allá al crear, no solamente ¨suficiente¨ código para hacerlo posible, sino un framework simple y a la vez completo, que separará con nitidez los diferentes aspectos de nuestra aplicación. Verá a lo que me refiero en un minuto.

Comenzaremos por ver cómo difiere el desarrollo con JavaScript en Node.js del desarrollo con JavaScript en un navegador.

Después, seguiremos la querida tradición de escribir una aplicación ¨Hola Mundo¨, la cuál es una aplicación Node.js muy básica que ¨hace¨ algo.

Luego, hablaremos de qué clase de aplicación ¨real¨ queremos construir, analizaremos qué diferentes partes necesitan implementarse para armar esta aplicación, y comenzaremos a trabajar en cada una de esas partes paso a paso.

Tal como fue prometido, en el camino aprenderemos algunos de los conceptos más avanzados de JavaScript, cómo usarlos y consideraremos por qué tiene sentido usar estos conceptos en vez de los que ya conocemos de otros lenguajes de programación.

JavaScript y Node.js

JavaScript y Usted

Antes de hablar de lo técnico, tomemos un momento para hablar de usted y su relación con JavaScript. El propósito de este capítulo es permitirle evaluar si la lectura de este documento tiene sentido para usted.

Si usted es como yo, comenzó ¨desarrollando¨ con HTML hace mucho tiempo, escribiendo documentos de HTML. Se encontró con esta cosa curiosa llamada JavaScript, pero solamente la usó de una manera muy básica, de vez en cuando añadiendo interactividad a sus páginas web.

Lo que usted quería era aprender a construir sitios web complejos - aprendió un lenguaje de programación como PHP, Ruby, Java, y comenzó a escribir código de ¨segundo plano¨.

Sin embargo, se mantuvo al tanto de JavaScript, vió que al presentarse jQuery, Prototype, y así por el estilo, las cosas avanzaron en el mundo de JavaScript, y que este lenguaje trataba de algo más que sólo *window.open()*.

Sin embargo seguía siendo asunto de interfaz, y aunque estaba interesante disponer de jQuery cada vez que quería darle un poco de sabor a alguna página web, a final de cuentas seguía siendo usted un *usuario* de JavaScript, pero no un *desarrollador* de JavaScript.

Y luego vino Node.js. JavaScript en el servidor, genial, ¿no?

Usted decidió que ya era hora de investigar el viejo, nuevo JavaScript. Pero, un momento, una cosa es escribir aplicaciones de Node.js; pero entender por qué necesitan escribirse de la manera que se escriben - eso es comprender JavaScript, y esta vez de verdad.

He aquí el problema: Ya que JavaScript en realidad vive dos y tal vez hasta tres vidas (el simpático ayudante para DHTML de los años 90, la parte más seria de interfaz como jQuery y así por el estilo, y ahora en el servidor), no es tán fácil encontrar información que le ayude a aprender JavaScript de la manera ¨correcta¨, aprendiendo a escribir aplicaciones Node.js de manera que no esté simplemente usando JavaScript, sino desarrollándolo.

Y ese es el asunto: usted ya es un desarrollador experimentado, no quiere aprender una técnica nueva solamente intentando a tientas y usándola mal; quiere estar seguro de estar encarándola desde el ángulo apropiado.

Por supuesto que existe documentación excelente. Pero la documentación por sí sola a veces no es suficiente. Lo que se necesita es guía.

Mi meta es proveer una guía.

Una palabra de advertencia

Hay algunas personas que son realmente excelentes con JavaScript. Yo no soy uno de ellos.

Soy simplemente la persona de la que hablé en el párrafo anterior. Conozco bastante del desarrollo de aplicaciones de segundo plano, pero con respecto a JavaScript "real" y Node.js, soy nuevo todavía. Recientemente he aprendido algunos aspectos mas avanzados de JavaScript. No soy experimentado.

Por esa razón este no es un libro "de principiante a experto", sino "de principiante a principiante avanzado".

Si cumplo mi objetivo, este será el tipo de documente que hubiera deseado tener cuando comencé con Node.js.

JavaScript del lado servidor

Las primeras encarnaciones de JavaScript vivían en los navegadores. Pero esto es simplemente el contexto. Define lo que puedes hacer con el lenguaje pero sin hablar de lo que el lenguaje mismo puede hacer. JavaScript es un lenguaje "completo": puedes usarlo en muchos diferentes contextos para lograr lo mismo que lograrías con cualquier otro lenguaje "completo".

Node.js en realidad es simplemente otro contexto más. Te permite ejecutar código JavaScript en el segundo plano, fuera del navegador.

Para ejecutar el JavaScript deseado en el segundo plano, necesita ser interpretado y bueno, ejecutado. Esto es lo que hace Node.js, usando V8 VM de Google, el mismo runtime environment (entorno de ejecución) para JavaScript que el que usa Google Chrome.

Es más, Node.js viene con muchos módulos muy útiles, de modo que no necesitas escribir todo de cero, por ejemplo algo que imprime una cadena en la consola.

Por lo tanto, Node.js en realidad es dos cosas: un runtime environment (entorno de ejecución) y una biblioteca.

Para utilizar estos, necesitas instalar Node.js. En vez de repetir el proceso aquí, le pido que visite el sitio oficial de instrucciones de instalación[2]. Por favor vuelva una vez que lo tenga funcionando.

"Hola Mundo"

Listo, saltemos al agua fría escribiendo nuestra primera aplicación Node.js: "Hola Mundo".

Inicie su editor preferido y crea un archivo llamado *holamundo.js*. Queremos que imprima "Hola Mundo" a STDOUT, y aquí está el código para lograr eso:

[2] https://github.com/joyent/node/wiki/Installation

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

Guarde el archivo y ejecútelo a través de Node.js:

```
1 node holamundo.js
```

Esto tendría que imprimir *Hola Mundo* en su terminal.

Bueno, esto es aburrido, ¿no? Escribamos algo real.

Una aplicación web a gran escala con Node.js

Los casos de uso

Mantengamoslo simple, pero realista:

- El usuario debe poder usar nuestra aplicación con un navegador.
- Cuando el usuario solicite http://*domain*/start debe encontrar una página de bienvenida que despliegue un formulario para subir un archivo.
- Después de elegir un archivo de imagen para subir y enviar el formulario, la imagen es subida a http://*domain*/upload, y mostrada una vez que termina de subir.

Muy bien. Podría lograr eso buscando en Google y ensamblando *algo*. Pero eso no es lo que queremos hacer aquí.

Es más, no queremos escribir solamente el código mas básico posible para lograr el objetivo, por mas elegante y correcto que sea ese código. A propósito vamos a añadir más abstracción de la necesaria para lograr sentirnos cómodos con la construcción de aplicaciones Node.js mas complejas.

The application stack

Analicemos nuestra aplicación. ¿Que partes se necesitan implementar para lograr los casos de uso?

- Queremos servir páginas web, por lo tanto necesitamos un **servidor HTTP**
- Nuestro servidor necesitará responder de maneras diferentes a solicitudes diferentes, dependiendo de cuál URL se solicitó, por eso necesitaremos algún tipo de **enrutador** para asignar solicitudes al "controlador de solicitud" (request handler) correspondiente.
- Para cumplir con las solicitudes que el servidor ha recibido y asignado usando el **enrutador**, necesitamos **controladores de solicitudes** (request handlers)
- El enrutador probablemente tendría que también manejar cualquier entrada POST de datos y pasársela a los controladores de solicitudes de una forma conveniente, por lo tanto necesitamos **request data handling** (controladores de datos de solicitud)
- No queremos solamente manejar solicitudes para URL, sino también mostrar contenido cuando se solicitan estos URL. Por lo tanto necesitamos algún tipo de **view logic** que puedan utilizar los request handlers para enviar contenido al navegador del usuario

- Por último, pero no por ello menos importante, el usuario podrá subir imagenes, por lo que necesitaremos algún tipo de **upload handling** (manejo de archivos enviados) que se encargue de los detalles

Pensemos un momento en cómo construiríamos esta pila con PHP. No es un gran secreto que generalmente usaríamos un servidor Apache HTTP con mod_php5 instalado. Esto da a entender que el asunto de "necesitamos poder servir páginas web y recibir solicitudes HTTP" no sucede adentro del mismo PHP.

Bueno, con node, las cosas son un poco diferentes. Porque con Node.js, no estamos solamente implementando nuestra aplicación, sino que estamos implementando el servidor HTTP entero. De hecho, nuestra aplicación web y su servidor web son básicamente lo mismo.

Esto puede dar la impresión de ser mucho trabajo, pero verá en un momento que con Node.js, no lo es.

Vamos a comenzar por el principio implementando la primer parte de nuestra pila, el servidor HTTP.

Construyendo el “application stack”

Un servidor HTTP básico

Llegado el momento cuando quise comenzar mi primer aplicación “real” de Node.js, me preguntaba no solamente cómo escribir el código sino también cómo organizar el código. ¿Necesíto tener todo en un sólo archivo? La mayoría de los tutoriales que te enseñan a escribir un servidor HTTP básico en Node.js tienen toda la lógica en un solo lugar. Pero, ¿y si quiero que mi código siga siendo entendible mientras implemento más cosas?

Resulta que es relativamente fácil mantener separadas las diferentes secciones de tu código al colocarlas en módulos.

Esto permite tener un archivo principal limpio, el cual ejecutas con Node.js, y módulos limpios que son usados por el archivo principal y también entre ellos mismos.

Así que vamos a crear un archivo principal que usaremos para iniciar nuestra aplicación, y un archivo de módulos donde vive nuestro código de servidor HTTP.

Estoy bajo la impresión de que la norma es nombrar el archivo principal *index.js*. Tiene sentido poner nuestro módulo de servidor en un archivo llamado *server.js*.

Comencemos con el módulo de servidor. Cree el archivo *server.js* en el directorio raíz de tu proyecto, y llénalo con el siguiente código:

```
1  var http = require("http");
2
3  http.createServer(function(request, response) {
4    response.writeHead(200, {"Content-Type": "text/plain"});
5    response.write("Hello World");
6    response.end();
7  }).listen(8888);
```

¡Eso es todo! Acabas de escribir un servidor HTTP que ya está funcionando. Hagamos una demostración ejecutando y probandolo. Primero, ejecuta tu script con Node.js:

```
1  node server.js
```

Ahora, inicia el navegador y apuntalo a http://localhost:8888/[3]. Esto debe mostrar una página web que dice “Hello World”.

[3] http://localhost:8888/

Es interesante, ¿no? Hablemos de lo que está pasando aquí y dejemos la cuestión de cómo organizar nuestro proyecto para más tarde. Prometo que volveremos a ese tema.

Analizando nuestro servidor HTTP

Analicemos qué está sucediendo aquí realmente.

La primera línea *require*, es decir: “requiere” el módulo *http* que viene incluído con Node.js. La misma se hace accesible a través del variable *http*.

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

Por favor ignore por un momento la definición de función entre corchetes que sigue a *http.createServer*.

Podríamos haber escrito el código que inicia nuestro servidor y lo hace escuchar en puerto 8888 así:

```
1  var http = require("http");
2
3  var server = http.createServer();
4  server.listen(8888);
```

Eso iniciaría un servidor HTTP eschuchando en puerto 8888 sin hacer más nada (ni siquiera respondiendo solicitudes entrantes).

La parte realmente interesante (y si tus antecedentes son de lenguajes más conservadores como PHP también te puede parecer extraño) es la definición de la función allí mismo donde esperarías ver el primer parámetro de la llamada a *createServer()*.

Resulta que esta definición de la función ES el primer (y único) parámetro que le damos a la llamada a *createServer()*. Porque en JavaScript, las funciones se pueden pasar como cualquiér otro valor.

Pasando funciones

Por ejemplo, podría hacer algo así:

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

¡Lea esto con cuidado! Lo que estamos haciendo es pasar la función *say* como el primer parámetro de la función *execute*. ¡No es valor de retorno de *say*, sino *say* mismo!

Por lo tanto, *say* llega a ser el variable local *someFunction* adentro de *execute*, y *execute* puede llamar la función adentro de este variable al emitir *someFunction()* (añadiendo corchetes).

Por supuesto, ya que *say* recibe un parámetro, *execute* puede pasar ese parámetro cuando llama *someFunction*.

Podemos pasar una función como parametro a otra función usando su nombre, como acabamos de hacer. Pero no necesitamos usar esa manera tan indirecta de primero definirla, luego pasarla. Podemos definir y pasar una función como parametro de otra función en un mismo lugar:

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

Definimos la función que queremos pasar a *execute* allí mismo en el lugar donde *execute* espera su primer parámetro.

De esta manera, no necesitamos siquiera darle nombre a la función, por esa razón le decimos *función anónima*.

Esto es un vislumbre de lo que yo llamo JavaScript “avanzado”, pero tomemoslo paso a paso. Por ahora simplemente aceptemos que en JavaScript, podemos pasar una función como parámetro al llamar a otra función. Podemos hacer esto al asignar nuestra función a un variable y pasarlo, o bién definiendo la función para pasar en un mismo lugar.

Pasar funciones hace que nuestro servidor HTTP funcione - ¿cómo?

Con este conocimiento, volvamos a nuestro servidor HTTP minimalista:

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

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

Ya debe estar claro lo que estamos haciendo aquí realmente: pasamos una función anónima a la función *createServer*.

Podríamos lograr lo mismo con el siguiente código:

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

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

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

Tal vez sea el momento indicado para preguntar: ¿por qué lo estamos haciendo así?

Retrollamada asíncrona activada por eventos

Para entender la razón por la que aplicaciones Node.js tienen que escribirse así, necesitamos entender cómo Node.js ejecuta nuestro código. El enfoque de Node no es único, pero el modelo de ejecución subyacente es diferente de runtime environments como Python, Ruby, PHP o Java.

Tomemos una porción de código muy simple como esta:

```
var result = database.query("SELECT * FROM hugetable");
console.log("Hola Mundo");
```

Por favor ignore por un momento el hecho de que no hemos hablado de conectarnos a una base de datos - es solamente un ejemplo. La primer línea consulta una base de datos por muchas filas, y la segunda línea imprime “Hola Mundo” a la consola.

Supongamos que la consulta a la base de datos es muy lenta, tiene que leer muchas filas, por lo que lleva varios segundos.

La manera que hemos escrito este código requiere que el interpretador de Node.js primero tiene que leer el conjunto entero de resultados de la base de datos, y luego puede ejecutar la función *console.log()*.

Si esta porcion de código en verdad fuera PHP, por ejemplo, funcionaría de la misma manera: leer todos los resultados de una vez, después ejecutar la próxima línea de código. Si este código fuera parte de un script de página web, el usuario tendría que esperar varios segundos a que cargue la página.

Sin embargo, en el modelo de ejecución de PHP, esto no sería un problema “global”: el servidor web inicia su propio proceso PHP por cada solicitud HTTP que recibe. Si una de estas solicitudes resulta en la ejecución de una porción de código lenta, este usuario particular tendría una página que carga lentamente, pero otros usuarios solicitando otras páginas no serían afectados.

El modelo de ejecución de Node.js es diferente - solamente hay un proceso. Si hay una consulta de base de datos lenta en alguna parte del proceso, esto afectará todo el proceso - todo frena hasta que termine la consulta de datos lenta.

Para evitar esto, JavaScript, y por lo tanto Node.js, introduce el concepto de retrollamada asíncrona, activada por eventos, utilizando un bucle de eventos.

Entenderemos este concepto al analizar una versión nueva de nuestro código problemático:

```
1 database.query("SELECT * FROM hugetable", function(rows) {
2     var result = rows;
3 });
4 console.log("Hello World");
```

Aquí, en vez de esperar que *database.query()* nos devuelva un resultado directamente, le pasamos un parámetro segundo, una función anónima.

En su versión previa, nuestro código era sincrónico: *primero* consulta la base de datos, y solamente cuando eso termina, *luego* imprime a la consola.

Ahora, Node.js puede manejar la consulta a la base de datos de manera asíncrona. Con la condición de que *database.query()* sea parte de una biblioteca asíncrona, esto es lo que hace Node.js: al igual que antes, toma la consulta y la manda a la base de datos. Pero en vez de esperar a que termine, toma nota mentalmente de que “Cuando en algún momento futuro el servidor de la base de datos termine y envíe el resultado de la consulta, entonces tendré que ejecutar la función anónima que fue pasada a *database.query()*.”

Luego, inmediatamente ejecuta *console.log()*, y después, entra en el bucle de eventos. Node.js continuamente itera este bucle vez tras vez siempre que no tenga más nada que hacer, esperando eventos. Eventos como por ejemplo, una consulta lenta de base de datos por fin entregando sus resultados.

Esto también explica por qué nuestro servidor HTTP necesita una función a la que pueda llamar al recibir solicitudes - si Node.js comenzara el servidor y luego pausara nada más, esperando la

próxima solicitud, esto sería muy ineficiente. Si un segundo usuario solicitara el servidor mientras está todavía sirviendo la primera solicitud, esa segunda solicitud no sería respondido hasta que terminara el primero - calculando más de un par de solicitudes HTTP por segundo, ya no funcionaría esto en absoluto.

Es importante notar que este modelo de ejecución asíncrono, de vía única, activado por eventos, no es un remedio mágico infinitamente redimensionable. Es simplemente uno de tantos modelos. Tiene sus limitaciones, una de las cuales es que hasta el momento, Node.js es un proceso único y se ejecuta en un solo núcleo de CPU. Personalmente encuentro este modelo bastante abordable, porque permite escribir aplicaciones que tienen que ver con simultaneidad de una manera eficiente y relativamente directa.

Tal vez quiera tomarse el tiempo de leer la excelente explicación de trasfondo de Felix Geisendoerfer *Understanding node.js*[4] para más información.

Juguemos un poco con este nuevo concepto. ¿Podemos comprobar que nuestro código sigue después de haber creado el servidor, aunque no haya habido ninguna solicitud de HTTP y la función de retrollamada que pasamos no se haya llamado? Intentémoslo:

```
1  var http = require("http");
2
3  function onRequest(request, response) {
4    console.log("Request received.");
5    response.writeHead(200, {"Content-Type": "text/plain"});
6    response.write("Hola Mundo");
7    response.end();
8  }
9
10 http.createServer(onRequest).listen(8888);
11
12 console.log("Servidor ha comenzado.");
```

Fíjese que uso *console.log* para imprimir un texto cada vez que es llamada la función *onRequest* (nuestra retrollamada), y otro texto inmediatamente *después* de iniciar el servidor HTTP.

Cuando lo iniciamos (*node server.js*, como siempre) inmediatamente imprimirá “Servidor ha comenzado.” en la línea de comandos. Cada vez que solicitemos nuestro servidor (al abrir http://localhost:8888/[5] en nuestro navegador), el mensaje “Request received.” (“Solicitud recibida”) se imprime en la línea de comandos.

Es JavaScript del lado servidor con retrollamada asíncrona activada por eventos en acción :-)

(Note que nuestro servidor probablemente imprimirá “Request received.” a STDOUT dos veces al abrir la página en un navegador. Eso es porque la mayoría de los navegadores intentarán cargar el favicono solicitando http://localhost:8888/favicon.ico cada vez que abras http://localhost:8888/).

[4] http://debuggable.com/posts/understanding-node-js:4bd98440-45e4-4a9a-8ef7-0f7ecbdd56cb

[5] http://localhost:8888/

Cómo maneja las solicitudes el servidor

Listo, rapidamente analicemos el resto de nuestro código de servidor, es decir, el cuerpo de nuestra función de retrollamada *onRequest()*.

Cuando se dispara la retrollamada y nuestra función *onRequest()* comienza, recibe dos parámetros: *request* and *response* (“solicitud” y “respuesta”).

Esos son objetos, y se pueden usar sus métodos para manejar los detalles de la solicitud HTTP que se hizo, y para responder a la solicitud (i.e., enviar algo de vuelta al navegador que hizo la solicitud a tu servidor.)

Y eso es precisamente lo que hace nuestro código: Cada vez que se recibe una solicitud, usa la función *response.writeHead()* para enviar un HTTP status 200 y content-type en la cabecera de la respuesta HTTP, y la función *response.write()* para enviar el texto “Hola Mundo” en el cuerpo de la respuesta HTTP.

Finalmente llamamos *response.end()* para finalizar nuestra respuesta.

En este momento no nos interesan los detalles de la solicitud, por eso no usamos el objeto *request* para nada.

Encontrando un lugar para nuestro módulo de servidor

Bueno, había prometido volver a la cuestión de cómo oraganizar nuestra aplicación. Tenemos el código para un servidor HTTP muy básico en el archivo *server.js*. Ya mencioné que es común tener un archivo principal llamado *index.js* que se usa para iniciar nuestra aplicación aprovechando los otros módulos de la aplicación (por ejemplo el módulo de servidor que vive en *server.js*).

Hablemos de cómo hacer de server.js un módulo Node.js real que pueda ser usado por el archivo principal *index.js* que pronto escribiremos.

Como habrá observado, ya hemos usado módulos en nuestro código, como este:

```
1  var http = require("http");
2
3  ...
4
5  http.createServer(...);
```

En algún lugar dentro de Node.js vive un módulo llamado “http”. Podemos usarlo en nuestro propio código al exigirlo (“require”) y asignar el resultado de “require” a un variable local.

Esto hace que nuestro variable local sea un objeto que tiene todos los métodos públicos que provee el módulo *http*.

Es una práctica común elegir el nombre del módulo para el nombre del variable local, pero se puede elegir cualquier nombre:

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

...

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

Está claro cómo aprovechar los módulos internos de Node.js. ¿Cómo creamos nuestros proios módulos? y ¿cómo los usamos?

Averigüemoslo al tomar nuestro script *server.js* y convertirlo en un módulo real.

Resulta que no tenemos que cambiar mucho. Formar un módulo con nuestro código significa que necesitamos *exportar* las porciones de su funcionalidad que queremos proveerle a scripts que requieran nuestro módulo.

Por ahora, la funcionalidad que necesita exportar nuestro servidor http es sencilla, los scripts que requieren nuestro módulo servidor simplemente necesitan iniciar el servidor.

Para posibilitar esto, pondremos nuestro código de servidor en una función llamada *start* y exportaremos esta función.

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

function start() {
  function onRequest(request, response) {
    console.log("Request received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

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

exports.start = start;
```

De esta manera, ahora podemos crear nuestro archivo principal *index.js* y comenzar nuestro HTTP allí, aunque el código para el servidor sigue estando en nuestro archivo *server.js*.

Crea un archivo *index.js* con el siguiente contenido:

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

server.start();
```

Como pueden ver, podemos usar nuestro módulo servidor igual que usaríamos cualquier módulo interno: al requerir (“require”) su archivo y asignarlo a un variable, sus funciones exportadas se hacer disponibles a nosotros.

Eso es todo, ahora podemos iniciar nuestra aplicación vía nuestro script principal y todavía hace exactamente lo mismo:

```
node index.js
```

Bien, ahora podemos poner en diferentes archivos las diferentes partes de nuestra aplicación y unirlas al hacerlas módulos.

Hasta ahora tenemos solamente la primer parte de nuestra aplicación: Podemos recibir solicitudes HTTP. Pero tenemos que hacer algo con ellas - dependiendo de cuál URL solicitó el navegador a nuestro servidor, tenemos que reaccionar diferentemente.

Para una aplicación muy sencilla, se podría hacer esto directamente adentro de la función de retrollamada *onRequest()*. Pero como ya dije, vamos a añadir un poco de abstracción para hacer nuestra aplicación que estamos usando de ejemplo un poco más interesante.

Hacer que diferentes solicitudes HTTP se dirigan a diferentes partes de nuestro código se llama “enrutar” - así que vamos a crear un módulo llamado *router* (“enrutador”).

¿Qué se necesita para “enrutar” solicitudes?

Necesitamos poder pasar el URL solicitado al enrutador, junto a cualquier posible parámetro GET y POST, y basándose en esto el enrutador necesita poder decidir cuál porción de código ejecutar. (Este “código a ejecutar” es la tercera parte de nuestra aplicación: una colección de controladores de solicitudes que hacen el trabajo real cuando se recibe una solicitud.)

Así que ncesitamos mirar adentro de la solicitud HTTP y extraer los parámetros GET y POST de ellos. Podría debatirse si esto debería ser parte del enrutador o parte del servidor (o ser su propio módulo), pero por ahora hagámoslo parte de nuestro servidor HTTP.

Toda la información que necesitamos está disponible en el objeto *request* que es pasado como el primer parámetro de nuestra función de retrollamada *onRequest()*. Pero para interpretar esta información necesitamos más nódulos Node.js, específicamente *url* y *querystring*.

El nódulo *url* provee métodos que nos permiten extraer diferentes partes de un URL (por ejemplo el camino solicitado y la cadena de consulta), y *querystring* se usa para analizar sintácticamente la cadena de consulta para encontrar parámetros de solicitud:

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

Por supuesto, también podemos usar *querystring* para analizar el cuerpo de una solicitud POST para encontrar parámetros de solicitud, como veremos después.

Ahora vamos a añadir a nuestra función *onRequest()* la lógica necesaria para averiguar cuál fue el camino de URL que solicitó el navegador:

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

function start() {
  function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log("Solicitud para " + pathname + " recibida.");
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("Hello World");
        response.end();
  }

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

exports.start = start;
```

Bien, ahora nuestra aplicaión puede distinguir solicitudes basándose en el URL solicitado. Esto nos permite asignar solicitudes al controlador de solicitudes correspondiente basándonos en el camino de URL, usando el enrutador que estamos por escribir.

En el contexto de nuestra aplicación, esto significa simplemente que solicitudes para los URL */start* y */upload* serán manejados por diferentes partes de nuestro código. Pronto veremos cómo se une todo.

Bueno, llegó el momento de escribir nuestro enrutador. Crea un nuevo archivo llamado *router.js*, con el siguiente contenido:

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

exports.route = route;
```

Por supuesto que este código básicamente no hace nada, pero está bien por ahora. Primero veremos cómo conectar este enrutador con nuestro servidor antes de añadir más lógica al enrutador.

Nuestro servidor HTTP necesita saber del enrutador y usarlo. Podríamos unir permanentemente esta dependencia en el servidor, pero ya hemos aprendido a las malas por experiencia en otros lenguajes de programación, por eso vamos a acoplar ligeramente el servidor y el enrutador al inyectar esta dependencia (tal vez quiera leer el excelente artículo de Martin Fowler acerca de Inyección de Dependencias para más información).

Primero vamos a ampliar la función *start()* de nuestro servidor para poder pasar la función route que será utilizado por el parámetro:

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

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

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

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

exports.start = start;
```

De igual manera ampliemos nuestro *index.js*, es decir, inyectando la función route de nuestro enrutador en el servidor.

```
1 var server = require("./server");
2 var router = require("./router");
3
4 server.start(router.route);
```

Otra vez estamos pasando una función, que ya no es algo nuevo para nosotros.

Si iniciamos nuestra aplicación ahora (*node index.js, como siempre*), y solicitamos un URL, se podrá ver en la salida de la aplicación que nuestro servidor HTTP usa nuestro enrutador y le pasa el camino solicitado:

```
1 bash$ node index.js
2 Solicitud para /foo recibida.
3 A punto de enrutar una solicitud para /foo
```

(Omití la salida de la solicitud de /favicon.ico por ser bastante molesta.)

Ejecución en el reino de los verbos

¿Puedo desviarme un poco nuevamente para hablar de programación funcional de nuevo?

Pasar funciones no es solamente una consideración técnica. Con relación al diseño de software, es casi filosófico. Considere esto: en nuestro archivo index, podríamos haber pasado el objeto *router* al servidor, y el servidor podría haber llamado la función *route* de este objeto.

De esta manera, hubiéramos pasado una *cosa*, y el servidor hubiera usado esta cosa para *hacer* algo. Oye, cosa enrutadora, ¿me haces el favor de enrutar esto?

Pero el servidor no necesita la cosa. Solo necesita lograr *hacer* algo, y para hacer algo, no se necesitan cosas en absoluto, se necesitan *acciones*. No se necesitan *sustantivos*, se necesitan *verbos*.

Comprender el cambio de esquema mental que implica esta idea es lo que me permitió comprender realmente la programación funcional.

Y lo entendí al leer la obra maestra de Steve Yegge Execution in the Kingdom of Nouns[6]. Vaya a leerlo ahora, de verdad. Es uno de las mejores escritos relacionados con el software que haya tenido el placer de encontrar.

[6] http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

Enrutando a controladores de solicitudes reales

Volvemos al tema. Ahora nuestro servidor HTTP y nuestro enrutador de solicitudes son mejores amigos y se hablan como queríamos.

Eso, por supuesto, no es suficiente. “Enrutar” significa que queremos manejar solicitudes a diferentes URL de diferente manera. Nos gustaría tener la “lógica de negocio” para solicitudes a */start* manejados en una función diferente a las solicitudes a */upload*.

Ahora mismo, la ruta “termina” en el enrutador, y el enrutador no es el lugar indicado para “hacer” algo con las solicitudes, porque eso no sería muy adaptable una vez que nuestra aplicación sea más compleja.

Llamemos a estas funciones, adonde se enrutan las solicitudes, *controladores de solicitudes*. Vamos a encarar estos ahora, porque no tiene sentido hacer más nada con el enrutador hasta no tener estos listos.

Nueva parte de la aplicación, nuevo módulo - no nos sorprende eso. Vamos a crear un módulo llamado requestHandlers (controladores de solicitudes), añadir una función para marcar la posición de cada controlador de solicitud, y exportar estos como métodos de los módulos:

```
function start() {
  console.log("Controlador de solicitud 'start' se ha llamado.");
}

function upload() {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
}

exports.start = start;
exports.upload = upload;
```

Esto nos permite conectar los controladores de solicitudes al enrutador, dándole a nuestro enrutador algo con que enrutar.

Llegado este momento, hay que tomar una decisión: ¿unimos permanentemente el módulo requestHandlers al enrutador o inyectamos esta dependencia? Inyección de dependencias, al igual que cualquier otro patrón, no se debe usar por el simple hecho de usarlo, pero en este caso tiene sentido acoplar ligeramente el enrutador y sus controladores de solicitudes, así el enrutador será realmente reutilizable.

Esto significa que necesitamos pasar los controladores de solicitudes de nuestro servidor a nuestro enrutador, pero es no parece ser correcto. Por eso debemos completar el proceso entero, pasarlos al servidor desde nuestro archivo principal, y de allí pasarlo al enrutador.

¿Cómo los vamos a pasar? Ahora mismo tenemos dos controladores, pero en una aplicación real, este número va a crecer y variar, y no queremos enredarnos asignando solicitudes a controladores en el enrutador cada vez que se añade un URL / controlador de solicitud. También meter algún *if request == x then call handler y* sería espantoso.

¿Un número variable de elementos, cada uno asignado a una cadena (el URL solicitado)? Suena como una matriz asociativa.

Bueno, es un poco decepcionante el hecho de que JavaScript no provee matriz asociativa, ¿ o sí? Resulta que ¡en verdad son objetos lo que queremos usar si necesitamos una matriz asociativa!

Hay una buena introducción a esto en http://msdn.microsoft.com/en-us/magazine/cc163419.aspx[7], voy a citar la parte relevante:

> En C++ o C#, cuando hablamos de objetos, nos referimos a instancias de clases o estructuras. Los objetos poseen propiedades y métodos diferentes en función de las plantillas (esto es, clases) desde las que se instancian. Esto no ocurre con los objetos de JavaScript. En JavaScript, los objetos son simplemente colecciones de parejas de nombres/valores; es decir, se podrían considerar como un diccionario con claves de cadena.

Si objetos en JavaScript son simplemente colecciones de parejas de nombres/valores, ¿como pueden tener métodos? Bueno, los valores pueden ser cadenas, números, etc. - ¡o funciones!

Listo, ahora por fin volvemos al código. Decidimos que queremos pasar la lista de requestHandlers como un objeto, y para acoplarlo lijeramente queremos inyectar este objeto al *route()*.

Comencemos por preparar el objeto en nuestro archivo principal *index.js*:

```
1  var server = require("./server");
2  var router = require("./router");
3  var requestHandlers = require("./requestHandlers");
4
5  var handle = {}
6  handle["/"] = requestHandlers.start;
7  handle["/start"] = requestHandlers.start;
8  handle["/upload"] = requestHandlers.upload;
9
10 server.start(router.route, handle);
```

Aunque *handle* (manejar) es una “cosa” (una colección de controladores de solicitudes), propongo que le demos nombre de verbo, porque eso producirá una expresión fluída en nuestro enrutador, como veremos pronto.

[7] http://msdn.microsoft.com/en-us/magazine/cc163419.aspx

Como puede observar, es muy simple asignar diferentes URL al mismo controlador de solicitud: al añadir una pareja de clave/valor de *”/”* y *requestHandlers.start*, podemos expresar de una manera clara y limpia que no solamente solicitudes a */start*, sino tambien solicitudes a / serán manejados por el controlador *start*.

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

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

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

    route(handle, pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

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

exports.start = start;
```

Hemos añadido el parámetro *handle* a nuestra función *start()*, y pasamos el objeto handle a la retrollamada *route()* como primer parámetro.

Modifiquemos la función *route()* de acuerdo con esto en nuestro archivo *router.js*:

```
function route(handle, pathname) {
  console.log("A punto de enrutar una solicitud para " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname]();
  } else {
    console.log("No se halló ningun controlador de solicitud para " + pathname);
  }
}

exports.route = route;
```

Lo que hacemos aquí es chequear si existe un cotrolador de solicitudes para dado pathname. Si existe, simplemente llamamos la función correspondiente. Ya que podemos acceder a nuestras funciones de controlador de solicitudes desde nuestro objeto, igual que accederíamos a elementos de una matriz asociativa, tenemos esta bonita expresión fluida *handle[pathname]();* que había mencionado antes: “Favor de *handle* este *pathname*”.

Bien, ¡eso es todo lo que necesitamos para conectar servidor, enrutador, y controladores de solicitudes! Al iniciar nuestra aplicación y solicitar http://localhost:8888/start[8] en nuestro navegador, comprobamos que de hecho fue llamado el controlador de solicitudes correcto:

```
Servidor ha comenzado.
Solicitud para /start recibida.
A punto de enrutar una solicitud para /start
Controlador de solicitud 'start' se ha llamado.
```

Y abrir http://localhost:8888/[9] en nuestro navegador comprueba que estas solicitudes también son manejados por el controlador de solicitud *start*:

```
Solicitud para / recibida.
A punto de enrutar una solicitud para /
Controlador de solicitud 'start' se ha llamado.
```

Haciendo que respondan los controladores de solicitudes

Hermoso. Ahora si pudieramos lograr que los controladores de solicitudes enviaran al de vuelta al navegador, eso sería mejor todavía, ¿no es cierto?

Recuerda que el “Hola Mundo” que despliega el navegador al soliciar una página todavía viene de la función *onRequest()* en nuestro archivo *server.js*.

En realidad “manejar” o “controlar” solicitudes significa “responder a solicitudes”, por lo tanto tenemos que hacer que nuestros controladores de solicitudes puedan hablar con el navegador igual que lo hace nuestra función *onRequest*.

La manera en que no se hace

El enfoque directo que nostros - como programadores con experiencia en PHP o Ruby - quizá seguiríamos en realidad es muy engañoso: funciona de maravilla, parece tener sentido, y de repente arruina todo en el momento menos esperado.

[8] http://localhost:8888/start
[9] http://localhost:8888/

A lo que me refiero con “enfoque directo” es esto: hacer que los controladores de solicitudes devuelvan (*return()*) el contenido que desean mostrar al usuario, y enviar esta información de respuesta en la función *onRequest* de vuelta al usuario.

Hagamos eso, y luego veremos por qué no es tan buena idea.

Comenzamos con los controladores de solicitudes, hacemos que devuelvan lo que nos gustaría desplegar en el navegador. Tenemos que cambiar *requestHandlers.js* a esto:

```
function start() {
  console.log("Controlador de solicitud 'start' se ha llamado.");
  return "Hola Start";
}

function upload() {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  return "Hola Upload";
}

exports.start = start;
exports.upload = upload;
```

Bien, de igual manera, el enrutador necesita devolver al servidor lo que los controladores de solicitudes le devuelvan a él. Por lo tanto necesitamos editar *router.js* así:

```
function route(handle, pathname) {
  console.log("A punto de enrutar una solicitud para " + pathname);
  if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
  } else {
    console.log("No se halló ningun controlador de solicitud para " + pathname);
    return "404 Not found";
  }
}

exports.route = route;
```

Como pueden ver, también devolvemos un texto si no se pudo enrutar la solicitud.

Por último, tenemos que modificar nuestro servidor para hacer que responda al navegador con el contenido que haya devuelto el controlador de solicitud vía el enrutador. Transforme *server.js* en:

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

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

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

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

exports.start = start;
```

Al reiniciar nuestra aplicación modificada, todo funciona de maravilla: solicitar http://localhost:8888/start[10] resulta en que se despliegue “Hola Start” en el navegador, solicitar http://localhost:8888/upload[11] nos da “Hola Upload”, y http://localhost:8888/foo[12] produce “404 Not found”.

Bueno, ¿cuál es el problema? La respuesta corta: si uno de los controladores de solicitudes quiere usar una operación asíncrona en el futuro, tendremos problemas.

Tomemos un poco más de tiempo para la respuesta larga.

Sincrónicas y Asíncronas

Como ya dijimos, los problemas van a surgir cuando incluyamos operaciones asíncronas en los controladores de solicitudes. Pero primero hablemos de operaciones sincrónicas, luego de operaciones asíncronas.

En vez de intentar explicar lo que significa “sincrónica” y “asíncrona”, demostremos lo que sucede si añadimos una operación sincrónica a nuestro controlador de solicitudes.

Para lograr esto, vamos a modificar nuestro controlador de solicitudes *start* para hacer que espere 10 segundos antes de devolver la cadena “Hola Start”. Ya que no existe *sleep()* en JavaScript, usaremos un truco ingenioso para hacerlo.

Por favor modifique *requestHandlers.js* de la siguiente manera:

[10] http://localhost:8888/start
[11] http://localhost:8888/upload
[12] http://localhost:8888/foo

```
function start() {
  console.log("Controlador de solicitud 'start' se ha llamado.");

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

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

function upload() {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  return "Hola Upload";
}

exports.start = start;
exports.upload = upload;
```

Para aclarar lo que logra eso: cuando se llama la función *start()*, Node.js espera 10 segundos y recién entonces devuelve inmediatamente, igual que antes.

(Por supuesto estamos imaginando que en vez de estar durmiendo 10 segundos hubiera una operación sincrónica real en *start()*, como algún tipo de proceso largo.)

Veamos que hace este cambio.

Como siempre, tenemos que reiniciar nuestro servidor. Esta vez le voy a pedir que siga un “protocolo” un poco complejo para ver que pasa: Primero, abra dos ventanas o pestañas de navegador. En la primera ventana de navegador ingrese http://localhost:8888/start[13] en la barra de direcciones, pero ¡no abra este URL todavía!

En la barra de direcciones de la segunda ventana de navegador, ingrese http://localhost:8888/upload[14] y de nuevo, por favor no teclee Intro todavía.

Ahora haga lo siguiente: teclee Intro en la primera ventana (“/start”) y rápidamente cambie a la segunda (“/upload”) y teclee Intro también.

Lo que observará es esto: el URL /start se demora diez segundos en abrir, como esperaríamos. Pero el URL /upload *también* se demora diez segundos en abrir, aunque no hay ningún *sleep()* en el controlador de solicitud correspondiente.

¿Por qué? Porque *start()* contiene una operación sincrónica. Ya hemos hablado del modelo de ejecución de Node - operaciones caras estan bien, pero no deben bloquear el proceso Node.js. En

[13] http://localhost:8888/start

[14] http://localhost:8888/upload

vez de eso, siempre que se necesite ejecutar una operación cara, estas deben ser colocadas en el trasfondo, y sus eventos deben ser manejados por el bucle de eventos.

Ahora veremos la razón de que la forma que construimos el “controlador de respuesta en el controlador de solicitud” no permite un uso apropiado de operaciones asíncronas.

De nuevo, intentemos experimentar el problema personalmente al modificar nuestra aplicación.

Vamos a usar nuestro controlador de solicitud *start* para esto otra vez. Favor de modificarlo de la siguiente manera (archivo *requestHandlers.js*):

```
1  var exec = require("child_process").exec;
2
3  function start() {
4    console.log("Controlador de solicitud 'start' se ha llamado.");
5    var content = "empty";
6
7    exec("ls -lah", function (error, stdout, stderr) {
8      content = stdout;
9    });
10
11   return content;
12 }
13
14 function upload() {
15   console.log("Controlador de solicitud 'upload' se ha llamado.");
16   return "Hola Upload";
17 }
18
19 exports.start = start;
20 exports.upload = upload;
```

Como habrán observado, acabamos de introducir un módulo Node.js nuevo, *child process*. Hicimos así porque nos permite utilizar una operación asíncrona muy simple y a la vez útil: *exec()*.

Lo que hace *exec()* es ejecutar un comando shell desde adentro de Node.js. En este ejemplo, vamos a utilizarlo para conseguir una lista de todos los archivos en el directorio actual (“ls -lah”), permitiéndonos mostrar esta lista en el navegador de un usuario solicitando el URL */start*.

Es bastante directo lo que hace el código: crear un nuevo variable *content* (con un valor inicial de “empty”), ejecutar “ls -lah”, poner el resultado en el variable, y devolverlo.

Como siempre, iniciaremos nuestra aplicación y visitaremos http://localhost:8888/start[15].

Esto carga una página web hermosa que despliega la cadena “empty” (vacía). ¿Cuál es el problema?

[15] http://localhost:8888/start

Tal vez ya se imagina que *exec()* hace lo suyo de una manera asíncrona. Eso es bueno porque de esta manera podemos ejecutar operaciones shell muy caras (por ejemplo, copiar archivos gigantes y así por el estilo) sin frenar totalmente nuestra aplicación como lo hizo la operación *sleep*.

(Si desea comprobar esto, reemplace “ls -lah” con una operación más cara como “find /”).

Pero no estamos satisfechos con nuestra elegante operación asíncrona si nuestro navegador no muestra su resultado, ¿verdad?

Vamos a arreglarlo entonces. Y en el proceso intentemos entender por qué la estructura actual no funciona.

El problema es que *exec()*, para de manera asíncrona usa una función de retrollamada.

En nuestro ejemplo, es una función anónima que es pasada como segundo parámetro a la llamada a la función *exec()*.

```
1 function (error, stdout, stderr) {
2   content = stdout;
3 }
```

Aquí está la raíz de nuestro problema: nuestro propio código se ejecuta de manera sincrónica, significa que inmediatamente después de llamar *exec()*, Node.js continúa ejecutando *return content;*. Llegado este momento, *content* está vacío todavía (“empty”), porque la función de retrollamada pasada a *exec()* no ha sido llamada todavía - porque *exec()* funciona de manera asíncrona.

Ahora, “ls -lah” es una operación rápida y de muy bajo costo (a no ser que haya millones de archivos en el directorio). Por esa razón la retrollamada se hace con relativa rapidez - pero siempre de forma asíncrona.

Pensemos en un comando más caro para aclarar la idea: “find/” toma aproximadamente un minuto en mi máquina, pero si reemplazo “ls -lah” con “find /” en el controlador de solicitud, todavía recibo una respuesta HTTP inmediatamente al abrir el URL /start - está claro que *exec()* hace algo en el segundo plano, mientras Node.js sigue con la aplicación, y podemos suponer que la función de retrollamada que hemos pasado a *exec()* no será llamada hasta que el comando “find /” haya terminado.

Pero, ¿cómo podemos lograr nuesto objetivo? Es decir, ¿cómo podemos mostrarle al usuario una lista de los archivos en el directorio actual?

Bueno, después de haber visto la manera *incorrecta*, hablemos de la manera correcta de hacer que nuestros controladores de solicitudes respondan al navegador.

Controladores de solicitud que responden con operaciones asíncronas

Acabo de usar la frase: “la manera correcta”. Qué peligroso. A menudo no hay solamente una “manera correcta”.

Pero una posible solución para esto es pasar funciones, como es común en Node.js. Examinemos esto.

Ahora mismo nuestra aplicación puede transportar el contenido (lo que el controlador de solicitud quiere mostrarle al usuario) desde el controlador de solicitudes al servidor HTTP al ir devolviendolo a través de los niveles de la aplicación (controlador de solicitud -> enrutador -> servidor).

Nuestro enfoque nuevo será el siguiente: en vez de traer el contenido al servidor, traeremos el servidor al contenido. Más específicamente, vamos a inyectar el objeto *response* (de la función de retrollamada *onRequest()* del servidor) en los controladores de solicitud a través del enrutador. Entonces los controldadores podrán usar las funciones de este objeto para responder ellos mismos a las solicitudes.

Ya hay suficientes explicaciones, veamos la receta paso por paso de cómo cambiar nuestra aplicación.

Comencemos por *server.js*:

```
1  var http = require("http");
2  var url = require("url");
3
4  function start(route, handle) {
5    function onRequest(request, response) {
6      var pathname = url.parse(request.url).pathname;
7      console.log("Solicitud para " + pathname + " recibida.");
8
9      route(handle, pathname, response);
10   }
11
12   http.createServer(onRequest).listen(8888);
13   console.log("Servidor ha comenzado.");
14 }
15
16 exports.start = start;
```

En vez de esperar un valor de retorno de la función *route()*, le pasamos un tercer parámetro, nuestro objeto *response*. Además, hemos removido cualquier llamada a los métodos de *response* del controlador de solicitud *onRequest()*, porque ahora esperamos que *route* se encargue de eso.

Seguimos con *router.js*:

```
function route(handle, pathname, response) {
  console.log("A punto de enrutar una solicitud para " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response);
  } else {
    console.log("No se halló ningun controlador de solicitud para " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

Seguimos el mismo patrón: en vez de esperar recibir un valor de retorno de nuestro controlador de solicitudes, pasamos el objeto *response*.

Si no puede ser utilizado nungún controlador de solicitud, nos encargamos nosotros mismos de responder como se debe con un encabezamiento y cuerpo “404”.

Y por último modificamos *requestHandlers.js*:

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

function start(response) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  exec("ls -lah", function (error, stdout, stderr) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(stdout);
    response.end();
  });
}

function upload(response) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hola Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Nuestras funciones de controlador necesitan aceptar el parámetro de respuesta, y necesitan utilizarlo para responder a la solicitud directamente.

El controlador *start* responderá desde adentro de la retrollamada anónima *exec()*, y el controlador *upload* seguirá respondiendo sencillamente “Hola Upload”, pero ahora usando el objeto *response*.

Si iniciamos nuestra aplicación de nuevo (*node index.js*), esto debería funcionar como esperamos.

Si quiere comprobar que una operación cara detrás de */start* ya no impedirá que solicitudes a */upload* respondan inmediatamente, entonces modifique su *requestHandlers.js* así:

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

function start(response) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);
      response.end();
    });
}

function upload(response) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hola Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Esto hará que solicitudes HTTP a http://localhost:8888/start[16] tomen por lo menos diez segundos, pero solicitudes a http://localhost:8888/upload[17] sean respondidas inmediatamente, aunque /start todavía esté ejecutando.

[16] http://localhost:8888/start
[17] http://localhost:8888/upload

Sirviendo algo útil

Hasta ahora, lo que hemos hecho está todo muy bien, pero no hemos creado ningún valor para los clientes de nuestro premiado sitio web.

Nuestro servidor, enrutador, y controladores de solicitudes están colocados cada uno en su lugar, por lo tanto podemos comenzar a añadir contenido a nuestro sitio que permita a los usuarios interactuar y seguir los pasos para elegir un archivo, subirlo, y luego visualizar en el navegador el archivo que han subido. Para simplificar vamos a suponer que solamente archivos de imágenes se van a subir y visualizar usando la aplicación.

Listo, tomemoslo paso por paso, pero como ya han sido explicadas la mayoría de las técnicas y principios de JavaScript, vamos a acelerar un poco. De todas formas a este autor le gusta demasiado su propia voz.

Aquí, paso por paso significa básicamente dos pasos: Primero veremos cómo manejar solicitudes POST entrantes (pero no envíos de archivos), y en un segundo paso haremos uso de un módulo Node.js externo para manejar el envío de archivo. He elegido este enfoque por dos razones.

Primero, manejar solicitudes POST básicas es relativamente simple con Node.js, pero a la vez vale la pena ejercitarlo porque podemos aprender algo. Segundo, manejar envíos de archivo (por ejemplo solicitudes POST de varias partes) *no* es simple con Node.js, por lo tanto va más allá del alcance de este tutorial, pero usar un módulo externo es una lección que sí tiene sentido incluir en este tutorial para principiantes.

Manejando solicitudes POST

Mantengamos esto extremadamente sencillo: Presentaremos un área de texto que puede ser llenado por el usuario y enviado al servidor en una solicitud POST. Una vez recibida y manejada la solicitud, desplegaremos el contendio del área de texto.

Nuestro controlador de solicitud */start* tiene que servir el HTML para este formulario con área de texto, así que vamos a añadirlo enseguida en el archivo *requestHandlers.js*:

```
function start(response) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" 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 upload(response) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hola Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Si esto no gana un premio Webby, no sé qué lo ganaría. Debería aparecer este formulario muy sencillo cuando solicite http://localhost:8888/start[18] en su navegador. Si no aparece, probablemente no ha re-iniciado la aplicación.

Ya lo sé: tener contenido de vista allí mismo en el controlador de solicitud es feo. Sin embargo, decidí no incluir ese nivel extra de abstracción (por ejemplo separación de lógica de vista y controlador) en este tutorial, porque me parece que no enseña nada que valga la pena en el contexto de JavaScript o de Node.js.

Más vale que usemos el espacio restante para un problema más interesante, manejar la solicitud POST que recibirá nuestro controlador de solicitudes */upload* cuando el usuario envíe este formulario.

Ahora que somos novatos avanzados, ya no nos sorprende el hecho de que manejar data POST se hace de una manera asíncrona, usando retrollamadas asíncronas.

Esto tiene sentido porque es posible que solicitudes POST sean muy grandes. Nada le impide al usuario ingresar texto de varios megabytes de tamaño. Manejar todo esa data de una vez resultaría en un operación sincrónica.

Para hacer que todo el proceso sea asíncrona, Node.js sirve la data POST en pequeñas porciones, retrollamadas que son disparadas por ciertos eventos. Estos eventos son *data* (una porción nueva de data POST ha sido recibida) y *end* (“fin”, todas las porciones han sido recibidas).

[18] http://localhost:8888/start

Tenemos que decirle a Node.js a cuáles funciones hacer la retrollamada cuando estos eventos ocurran. Esto se hace al añadir *listeners* (“oyentes”) al objeto *request* que es pasado a nuestra retrollamada *onRequest* siempre que se recibe una solicitud HTTP.

Esto básicamente luce así:

```
request.addListener("data", function(chunk) {
  // llamado cuando se recibe una nueva porción de data
});

request.addListener("end", function() {
  // llamado cuando se han recibido todas las porciones de data
});
```

Surge la pregunta de dónde implementar esta lógica. Actualmente podemos acceder al objeto *request* desde nuestro servidor solamente - no lo pasamos ni al enrutador ni a los controladores de solicicutdes como hicimos con el objeto *response*.

En mi opinión, el servidor HTTP debería entregarle a la aplicación toda la información de una solicitud que sea necesaria para cumplir con su trabajo. Por lo tanto, sugiero que manejemos el procesamiento de data POST en el mismo servidor y que pasemos la data final al enrutador y a los controladores de solicitudes, que luego decidirán qué hacer con ella.

Así que la idea es poner las retrollamadas *data* y *end* en el servidor, recolectar todas las porciones de data POST en la retrollamada *data*, y llamar al enrutador al recibir el evento *end*, mientras pasamos las porciones de data recolectadas al enrutador, que por su parte las pasará a los controladores de solicitudes.

Ahí vamos, comenzando por *server.js*:

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

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

    request.setEncoding("utf8");

    request.addListener("data", function(postDataChunk) {
      postData += postDataChunk;
      console.log("Recibida porción de data POST '"+
```

```
15        postDataChunk + "'.");
16      });
17
18      request.addListener("end", function() {
19        route(handle, pathname, response, postData);
20      });
21
22    }
23
24    http.createServer(onRequest).listen(8888);
25    console.log("Servidor ha comenzado.");
26  }
27
28  exports.start = start;
```

Hemos hecho básicamente tres cosas aquí: Primero, hemos definido que esperamos que la codificación de la data recibida sea UTF-8. Añadimos un oyente de eventos (“listener”) para el evento “data”, que paso a paso llena nuestro variable nuevo *postData* cada vez que llega una nueva porción de data POST. Hemos colocado la llamada al enrutador a la retrollamada del evento *end* para asegurarnos de que solamente sea llamada una vez que se haya recolectado toda la data POST. También pasamos la data POST al enrutador, porque vamos a necesitarlo en los controladores de solicitudes.

Registrar cada porción recibida en la consola probablemente sea una mala idea para código de producción (megabytes de data POST, ¿recuerda?), pero tiene sentido para observar qué sucede.

Sugiero jugar con esto un poco. Ingrese cantidades pequeñas de texto en el área de texto así como cantidades grandes. Verá que para los textos más grandes la retrollamada *data* se hace varias veces.

Vamos a añadir más ingenio todavía a nuestra aplicación. En la página /upload, vamos a desplegar el contenido recibido. Para lograr eso, necesitamos pasar *postData* a los controladores de solicitudes, en *router.js*:

```
1   function route(handle, pathname, response, postData) {
2     console.log("A punto de enrutar una solicitud para " + pathname);
3     if (typeof handle[pathname] === 'function') {
4       handle[pathname](response, postData);
5     } else {
6       console.log("No se halló ningun controlador de solicitud para " + pathname);
7       response.writeHead(404, {"Content-Type": "text/plain"});
8       response.write("404 Not found");
9       response.end();
10    }
11  }
12
13  exports.route = route;
```

También en *requestHandlers.js*, incluimos la data en la respuesta del controlador de solicitud *upload*:

```
function start(response, postData) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" 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 upload(response, postData) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Usted ha enviado: " + postData);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Eso es todo, ahora podemos recibir data POST y usarla en nuestros controladores de solicitudes.

Un último asunto con respecto a este tema: lo que pasamos al enrutador y a los controladores de solicitudes es el cuerpo completo de nuestra solicitud POST. Probablemente deseemos consumir los campos individuales que componen la data POST, en este caso, el valor del campo *text*.

Ya hemos leído del módulo *querystring* que nos ayuda con esto:

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

function start(response, postData) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" 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 upload(response, postData) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Usted ha enviado el texto: "+
  querystring.parse(postData).text);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Bueno, para un tutorial de principiantes, eso es todo lo que hay que decir acerca del mamejo de data POST.

Manejando Envíos de Archivos

Encaremos nuestro último caso de uso. Nuestro plan era permitir al usuario subir una imagen y desplegarla en el navegador.

Allá en los años 90, esto habría sido el modelo de negocio de un OPI. Hoy, bastará para enseñarnos dos cosas: cómo instalar bibliotecas externas de Node.js, y cómo utilizarlas en nuestro propio código.

El módulo externo que vamos a usar es *node-formidable* de Felix Geisendoerfer. Permite evitar todos los detalles complicados del análisis sintáctico de data de archivos entrantes. En realidad, el manejo de archivos entrantes implica básicamente manejar data POST “nada más” - pero la clave realmente *está* en los detalles, por eso usar una solución ya preparada en este caso tiene mucho sentido.

Para utilizar el código de Felix hay que instalar el módulo Node.js correspondiente. Node.js viene con su propio administrador de paquetes, llamado *NPM*. Nos permite instalar modulos externos Node.js de una manera muy conveniente. Teniendo una instalación Node.js fucionando, solamente hay que escribir

```
1  npm install formidable
```

en la línea de comandos. Si la salida de eso termina en

```
1  npm info build Success: formidable@1.0.2
2  npm ok
```

entonces está todo bien.

Ahora el módulo *formidable* está disponible para nuestro porpio código - lo único que necesitamos hacer es usar *require* para exigirlo igual que a los módulos incorporados que usamos antes:

```
1  var formidable = require("formidable");
```

La metáfora que usa formidable es la de un formulario enviado vía HTTP POST, logrando así que se pueda analizar sintácticamente en Node.js. Solamente necesitamos crear un nuevo *IncomingForm* (“Formulario Entrante”), que es una abstracción de este formulario enviado, y así puede ser utilizado para analizar el objeto *request* de nuestro servidor HTTP en busca de los campos (“fields”) y archivos que fueron enviados a través de este formulario.

El ejemplo de código en la página del proyecto node-formidable muestra cómo encajan las diferentes partes entre sí:

```
var formidable = require('formidable'),
    http = require('http'),
    sys = require('sys');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();
    form.parse(req, function(error, fields, files) {
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(sys.inspect({fields: fields, files: files}));
    });
    return;
  }

  // show a file upload form
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="text" name="title"><br>'+
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
  );
}).listen(8888);
```

Si colocamos este código en un archivo y lo ejecutamos a través de *node*, podemos enviar un formulario simple, incluyendo un envío de archivo, y ver cómo es la estructura del objeto *files*, que es pasado a la retrollamada definida en la llamada *form.parse*:

```
received upload:

{ fields: { title: 'Hello World' },
  files:
   { upload:
      { size: 1558,
        path: '/tmp/1c747974a27a6292743669e91f29350b',
        name: 'us-flag.png',
        type: 'image/png',
        lastModifiedDate: Tue, 21 Jun 2011 07:02:41 GMT,
```

```
11          _writeStream: [Object],
12          length: [Getter],
13          filename: [Getter],
14          mime: [Getter] } } }
```

Para hacer que suceda nuestro caso de uso, tenemos que incluir la lógica de formidable para analizar formularios en la estructura de nuestro código. También tendremos que averiguar cómo servir el contenido del archivo enviado (que está guardado en la carpeta */tmp*) al navegador que lo solicite.

Encaremos primero lo último: si hay un archivo de imágen en nuestro disco rígido local, ¿cómo lo serviremos a un navegador que lo solicite?

Obviamente vamos a leer el contenido de este archivo a nuestro servidor Node.js, y no nos sorprende que haya un módulo para eso - se llama *fs*.

Vamos a añadir otro controlador de solicitud para el URL */show*, que estará codificado “en duro” para mostrar el contenido del archivo */tmp/test.png*. Por supuesto, tiene sentido primero guardar una imágen png real en esta ubicación.

Modifiquemos *requestHandlers.js* de la siguiente manera:

```
1  var querystring = require("querystring"),
2      fs = require("fs");
3
4  function start(response, postData) {
5    console.log("Controlador de solicitud 'start' se ha llamado.");
6
7    var body = '<html>'+
8      '<head>'+
9      '<meta http-equiv="Content-Type" '+
10     'content="text/html; charset=UTF-8" />'+
11     '</head>'+
12     '<body>'+
13     '<form action="/upload" method="post">'+
14     '<textarea name="text" rows="20" cols="60"></textarea>'+
15     '<input type="submit" value="Submit text" />'+
16     '</form>'+
17     '</body>'+
18     '</html>';
19
20     response.writeHead(200, {"Content-Type": "text/html"});
21     response.write(body);
22     response.end();
23 }
24
```

```
25 function upload(response, postData) {
26   console.log("Controlador de solicitud 'upload' se ha llamado.");
27   response.writeHead(200, {"Content-Type": "text/plain"});
28   response.write("Usted ha enviado el texto: "+
29   querystring.parse(postData).text);
30   response.end();
31 }
32
33 function show(response) {
34   console.log("Controlador de solicitud 'show' se ha llamado.");
35   response.writeHead(200, {"Content-Type": "image/png"});
36   fs.createReadStream("/tmp/test.png").pipe(response);
37 }
38
39 exports.start = start;
40 exports.upload = upload;
41 exports.show = show;
```

También necesitamos asignar este nuevo controlador de solicitudes al URL */show* en el archivo *index.js*:

```
1 var server = require("./server");
2 var router = require("./router");
3 var requestHandlers = require("./requestHandlers");
4
5 var handle = {}
6 handle["/"] = requestHandlers.start;
7 handle["/start"] = requestHandlers.start;
8 handle["/upload"] = requestHandlers.upload;
9 handle["/show"] = requestHandlers.show;
10
11 server.start(router.route, handle);
```

Al reiniciar el servidor y abrir http://localhost:8888/show[19] en el navegador, la imágen guardada en */tmp/test.png* debería ser desplegada.

Bien. Ahora lo único que necesitamos hacer es

- añadir un elemento para subir un archivo al formulario que es servido en */start*,
- integrar node-formidable en el controlador de solicitudes *upload*, para poder guardar el archivo enviado en */tmp/test.png*,

[19] http://localhost:8888/show

- empotrar la imágen enviada en la salida HTML del URL */upload*

El primer paso es simple. Tenemos que añadir el tipo de codificación *multipart/form-data* a nuestro formulario HTML, quitar el área de texto, añadir un campo de envío de archivo, y cambiar el texto del botón a “Subir archivo”. Hagámos eso en el archivo *requestHandlers.js*:

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

function start(response, postData) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload">'+
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

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

function upload(response, postData) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response) {
  console.log("Controlador de solicitud 'show' se ha llamado.");
  response.writeHead(200, {"Content-Type": "image/png"});
```

```
37    fs.createReadStream("/tmp/test.png").pipe(response);
38  }
39
40  exports.start = start;
41  exports.upload = upload;
42  exports.show = show;
```

Excelente. El próximo paso es un poco más complejo, por supuesto. El primer problema es que queremos manejar el envío de archivo en nuestro controlador de solicitud *upload*, y allí tendremos que pasar el objeto *request* a la llamada a *form.parse* de node-formidable.

Pero solamente tenemos el objeto *response* y la matriz *postData*. Que triste. Parece que tendremos que pasar el objeto *request* desde el servidor al enrutador y también hasta el controlador de solicitud. Tal vez haya soluciones más elegantes, pero este enfoque cumplirá nuestro propósito por ahora.

Ya que estamos, quitemos todo ese asunto de *postData* de nuestro servidor y controladores de solicitudes - no lo necesitaremos para manejar nuestro envío de archivo, incluso causa un problema: ya hemos “consumido” los eventos *data* del objeto *request* en el servidor, esto quiere decir que *form.parse*, que también necesita consumir esos eventos, no recibiría más data de ellos (porque node.js no tiene búfer de datos).

Comencemos por *server.js* - quitamos el manejo de postData y la línea *request.setEncoding* (ahora node-formidable mismo manejará eso) y en su lugar pasamos *request* al enrutador:

```
1   var http = require("http");
2   var url = require("url");
3
4   function start(route, handle) {
5     function onRequest(request, response) {
6       var pathname = url.parse(request.url).pathname;
7       console.log("Solicitud para " + pathname + " recibida.");
8       route(handle, pathname, response, request);
9     }
10
11    http.createServer(onRequest).listen(8888);
12    console.log("Servidor ha comenzado.");
13  }
14
15  exports.start = start;
```

Ahora seguimos con *router.js* - ya no necesitamos pasar *postData*, en vez de eso pasamos *request*:

```
function route(handle, pathname, response, request) {
  console.log("A punto de enrutar una solicitud para " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, request);
  } else {
    console.log("No se halló ningun controlador de solicitud para " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

Ahora el objeto *request* puede ser utilizado en la función *upload* de nuestro controlador de solicitud. node-formidable se encargará de los detalles de guardar el archivo subido a una carpeta local dentro de */tmp*, pero nosotros mismos necesitamos asegurarnos de que a este arhivo se le dé el nombre de */tmp/test.png*. Sí, lo mantenemos muy simple al suponer que solamente imagenes tipo PNG serán subidas.

Hay un poco de complejidad extra en la lógica de cambiar el nombre: a la implementación de node para Windows no le gusta cuando intentas cambiar el nombre de un archivo en el lugar de uno ya existente. Por esa razón tenemos que borrar el arvhico en caso de error.

Armemos las piezas de manejar el archivo subido y cambiarle el nombre ahora, en el arhivo *requestHandlers.js*:

```
var querystring = require("querystring"),
    fs = require("fs"),
    formidable = require("formidable");

function start(response) {
  console.log("Controlador de solicitud 'start' se ha llamado.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload" multiple="multiple">'+
```

```
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

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

function upload(response, request) {
  console.log("Controlador de solicitud 'upload' se ha llamado.");

  var form = new formidable.IncomingForm();
  console.log("a punto de analizar 'parse' ");
  form.parse(request, function(error, fields, files) {
    console.log("parsing terminado");

    /* Posible error en sistemas Windows:
       intento de cambiar el nombre a un archivo ya existente */
    fs.rename(files.upload.path, "/tmp/test.png", function(error) {
      if (error) {
        fs.unlink("/tmp/test.png");
        fs.rename(files.upload.path, "/tmp/test.png");
      }
    });
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("received image:<br/>");
    response.write("<img src='/show' />");
    response.end();
  });
}

function show(response) {
  console.log("Controlador de solicitud 'show' se ha llamado.");
  response.writeHead(200, {"Content-Type": "image/png"});
  fs.createReadStream("/tmp/test.png").pipe(response);
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

Eso es todo. Reinicie el servidor, y el caso de uso completo estará disponible. Seleccione una imágen PNG local de su disco rígido, súbalo, y será mostrado en la página web.

Conclusión y perspectiva

¡Felicitaciones! ¡Hemos cumplido nuestro objetivo! Hemos escrito una aplicación web Node.js sencilla y a la vez completa. Hemos hablado de JavaScript del lado servidor, programación funcional, operaciones sincrónicas y asíncronas, retrollamadas, eventos, módulos personalizados, internos y externos, y mucho más.

Por supuesto hay mucho de los que no hemos hablado: cómo comunicarnos con una base de datos, cómo escribir pruebas unitarias, cómo crear módulos externos que se puedan instalar vía NPM, o incluso algo simple como la manera de manejar solicitudes GET.

Pero ese es el destino de todo libro para principiantes - no puede hablar de todos los aspectos con lujo de detalle.

Muchos lectores han pedido un segundo libro, un libro que continúe donde *El Libro Principiante de Node* termina, permitiendoles zambullirse a lo profundo del desarrollo con Node.js, aprendiendo acerca de manejo de base de datos, frameworks, pruebas unitarias, y mucho más.

Recientemente comencé a trabajar en este libro. Se llama *El Libro Artesano de Node*, y está disponible vía Leanpub también.

Este nuevo libro es un ´trabajo en curso´, al igual que *El Libro Principiante de Node* lo fue cuando primero salió.

Mi idea es ampliar *El Libro Artesano de Node* constantemente, a la vez que recojo y uso reacciones de la comunidad.

Por lo tanto, decidí poner a la venta una primera versión de este nuevo libro lo antes posible.

Hasta ahora, *El Libro Artesano de Node* contiene los siguientes capítulos:

- Working with NPM and Packages
- Test-Driven Node.js Development
- Object-Oriented JavaScript

Ya tiene 28 páginas, y más capítulos se añadirán pronto.

El precio final del libro completo será $9.99, sin embargo como lector y comprador de *El Libro Principiante de Node*, tiene la oportunidad de comprar el nuevo libro mientras se completa, con un descuento del 50%, por solo $4.99.

De esta manera, tendrá las siguientes ventajas:

- Aprender: Conseguirá capítulos nuevos abarcando Node.js avanzado y temas de JavaScript tan pronto como se completan.

- Sugerir Ideas: ¿qué le gustaría a usted ver en la versión final del libro?
- Ahorrar: Solamente pagará $4.99 una vez, y recibirá todas las futuras actualizaciones *gratis*, incluyendo la versión final y completa del libro.

Si le gustaría aprovechar esta oferta, simplemente vaya a

http://leanpub.com/nodecraftsman[20]

y use el código de cupón *nodereader*.

Esta oferta es válida hasta el 30 de Noviembre, 2013.

[20] http://leanpub.com/nodecraftsman