



UNIVERSIDAD TECNOLÓGICA NACIONAL
Facultad Regional Córdoba

Diplomatura en Desarrollo WEB
Módulo: Páginas con contenido dinámico

UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Córdoba

Secretaría de Extensión Universitaria

Área Tecnológica de Educación a distancia

Coordinador General de Educación a distancia:

Magíster Leandro D. Torres

Curso:

**DIPLOMATURA EN DESARROLLO WEB EV
15030**

Módulo:

“Páginas con contenido dinámico”

Autor:

Ing. Ricardo Martín Espeche

Tutor Ricardo Martín Espeche



Referencias

Este material se confeccionó en base a las últimas tendencias que hoy se encuentra en Internet. Para eso se buscó el mejor material y se recompiló dicha información para facilitar el aprendizaje del mismo.

El material utilizado para esta primera unidad fue:

- Manual de NodeJS
- Cómo crear un API REST usando Node.js, Express y MongoDB

Índice

Capítulo 7	4
Node.js	4
7.1 Introducción básica	4
7.1.1 Que es Node.js?.....	4
7.2 Instalar NodeJS	6
7.2.1 Probando los primeros comandos NodeJS.....	6
7.3 Módulos y NPM.....	8
7.3.1 Incluir módulos con “require”	8
7.3.2 Comando NPM para gestión de paquetes	9
7.3.3 Instalar paquetes de manera global.....	10
7.4 Crear servidor HTTP	11
7.4.1 Código completo.....	12
7.4.2 Iniciar el servidor	12
7.5 Eventos en NodeJS	13
7.5.1 Modulo de eventos	13
7.5.2 Definir un Evento	13
Capítulo 8	16
Express.js.....	16
8.1 Introducción básica	16
8.1.1 Instalación	16
8.1.2 app.js.....	17
8.2 Configuración.....	19
8.3 Rutas	21
8.4 Otros comandos.....	24



8.4.1 Listen.....	24
8.4.2 Helpers.....	24
8.4.3 Error Handler.....	25
8.4.4 Sesiones.....	25
8.4.5 Respuesta	26
Capítulo 9	28
REST.....	28
9.1 Introducción básica	28
9.2 Desarrollando nuestro API RESTful	29
9.2 Nuestro servidor Node.js.....	30
9.3 Creando los modelos de nuestra API REST.....	32
9.4 Implementando los controladores de nuestras rutas o endpoints	34
9.4.1 Probando nuestro API REST en el navegador.....	37



Capítulo 7

Node.js

7.1 Introducción básica

NodeJS es una tecnología que se apoya en el motor de Javascript V8 para permitir la ejecución de programas hechos en Javascript en un ámbito independiente del navegador. A veces se hace referencia a NodeJS como Javascript del lado del servidor, pero es mucho más.

La característica más importante de NodeJS, y que ahora otra serie de lenguajes están aplicando, es la de no ser bloqueante. Es decir, si durante la ejecución de un programa hay partes que necesitan un tiempo para producirse la respuesta, NodeJS no para la ejecución del programa esperando que esa parte acabe, sino que continúa procesando las siguientes instrucciones. Cuando el proceso lento termina, entonces realiza las instrucciones que fueran definidas para realizar con los resultados recibidos.

Esa característica, y otras que veremos en el Manual de NodeJS hace que el lenguaje sea muy rápido.

Existen muchas aplicaciones de NodeJS, aunque la más famosa es la de atender comunicaciones por medio de sockets, necesarias para las aplicaciones en tiempo real.

7.1.1 Que es Node.js?

NodeJS es básicamente un framework para implementar operaciones de entrada y salida, como decíamos anteriormente. Está basado en eventos, streams y construido encima del motor de Javascript V8, que es con el que funciona el Javascript de Google Chrome. A lo largo de este capítulo daremos más detalles, pero de momento nos interesa abrir la mente a un concepto diferente a lo que podemos conocer, pues NodeJS nos trae una nueva manera de entender Javascript.

Si queremos entender esta plataforma, lo primero que debemos de hacer es desprendernos de varias ideas que los desarrolladores de Javascript hemos cristalizado a lo largo de los años que llevamos usando ese lenguaje. Para empezar, NodeJS se programa del lado del servidor, lo



que indica que los procesos para el desarrollo de software en "Node" se realizan de una manera muy diferente que los de Javascript del lado del cliente.

De entre alguno de los conceptos que cambian al estar Node.JS del lado del servidor, está el asunto del "Cross Browser", que indica la necesidad en el lado del cliente de hacer código que se interprete bien en todos los navegadores. Cuando trabajamos con Node solamente necesitamos preocuparnos de que el código que escribas se ejecute correctamente en tu servidor. El problema mayor que quizás podamos encontrar a la hora de escribir código es hacerlo de calidad, pues con Javascript existe el habitual problema de producir lo que se llama "código espagueti", o código de mala calidad que luego es muy difícil de entender a simple vista y de mantener en el futuro.

Otras de las cosas que deberías tener en cuenta cuando trabajas con NodeJS, que veremos con detalle más adelante, son la programación asíncrona y la programación orientada a eventos, con la particularidad que los eventos en esta plataforma son orientados a cosas que suceden del lado del servidor y no del lado del cliente como los que conocemos anteriormente en Javascript "común".



7.2 Instalar NodeJS

Si no tienes instalado todavía NodeJS el proceso es bastante fácil. Por supuesto, todo comienza por dirigirse a la página de inicio de NodeJS:

<https://nodejs.org/en/>

Allí encontrarás el botón para instalarlo "Install" que pulsas y simplemente sigues las instrucciones.

Los procesos de instalación son sencillos y ahora los describimos. Aunque son ligeramente distintos de instalar dependiendo del sistema operativo, una vez lo tienes instalado, el modo de trabajo con NodeJS es independiente de la plataforma y teóricamente no existe una preferencia dada por uno u otro sistema, Windows, Linux, Mac, etc. Sin embargo, dependiendo de tu sistema operativo sí puede haber unos módulos diferentes que otros, ésto es, unos pueden funcionar en Linux y no así en otros sistemas, y viceversa.

Si estás en Windows, al pulsar sobre Install te descargará el instalador para este sistema, un archivo con extensión "msi" que como ya sabes, te mostrará el típico asistente de instalación de software. Una vez descargado, ejecutas el instalador y listo.

A partir de ahora, para ejecutar "Node" tienes que irte a la línea de comandos de Windows e introducir el comando "node". Entonces entrarás en la línea de comandos del propio NodeJS donde puedes ya escribir tus comandos Node, que luego veremos.

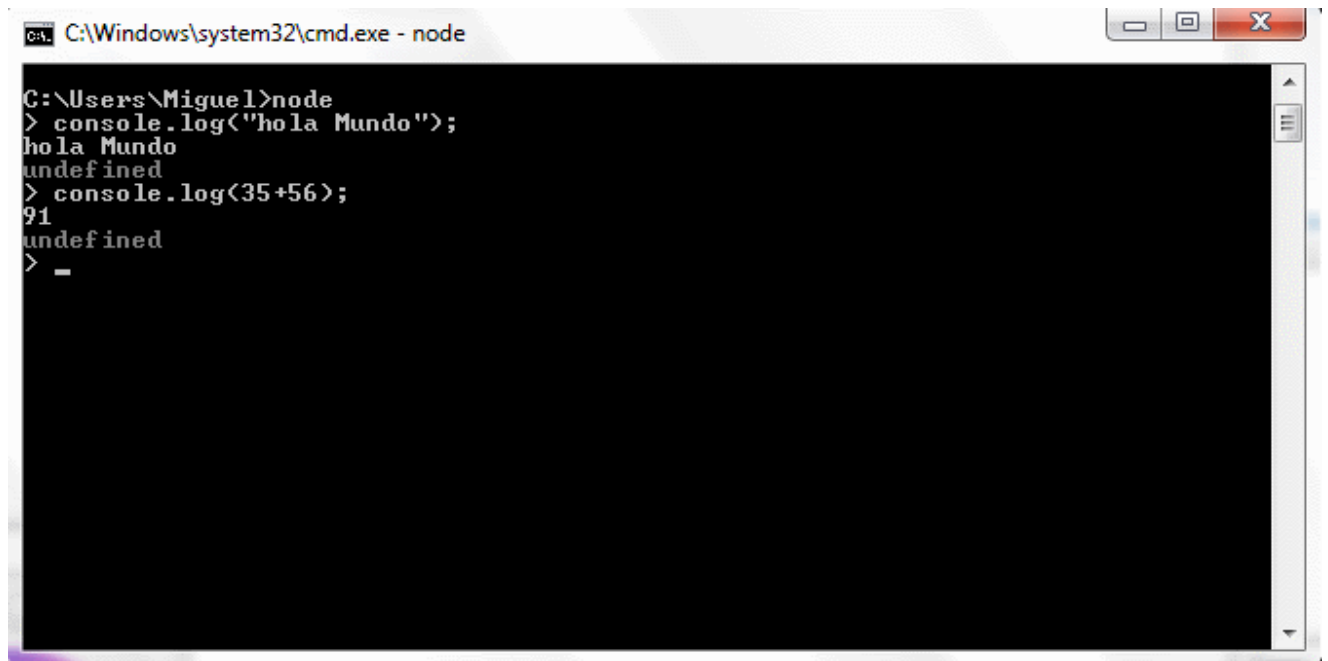
7.2.1 Probando los primeros comandos NodeJS

En NodeJS la consola de Node puedes escribir instrucciones Javascript. Si lo deseas, puedes mandar mensajes a la consola con `console.log()` por lo que ésta podría ser una bonita instrucción para comenzar con node:

```
$ node  
console.log("hola mundo");
```

Te mostrará el mensaje "hola mundo" en la consola.

Podemos ver en la siguiente imagen un par de mensajes a la consola de Node que podemos utilizar para comenzar nuestro camino.



```
C:\Windows\system32\cmd.exe - node

C:\Users\Miguel>node
> console.log("hola Mundo");
hola Mundo
undefined
> console.log(35+56);
91
undefined
> -
```

Observarás que te salen unos mensajes "undefined" en la consola y es porque las instrucciones que estamos ejecutando como "console.log()" no devuelven ningún valor.



7.3 Módulos y NPM

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

El gestor de paquetes npm, no obstante, es un poquito distinto a otros gestores de paquetes que podemos conocer, porque los instala localmente en los proyectos. Es decir, al descargarse un módulo, se agrega a un proyecto local, que es el que lo tendrá disponible para incluir. Aunque cabe decir que también existe la posibilidad de instalar los paquetes de manera global en nuestro sistema.

7.3.1 Incluir módulos con “require”

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

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

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

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

Existen distintos módulos que están disponibles de manera predeterminada en cualquier proyecto NodeJS y que por tanto no necesitamos traernos previamente a local mediante el gestor de paquetes npm. Esos toman el nombre de "Módulos nativos" y ejemplos de ellos tenemos el propio `"http"`, `"fs"` para el acceso al sistema de archivos, `"net"` que es un módulo para conexiones de red todavía de más bajo nivel que `"http"`, `"URL"` que permite realizar operaciones sobre `"url"`, el módulo `"util"` que es un conjunto de utilidades, `"child_process"` que te da herramientas para ejecutar sobre el sistema, `"domain"` que te permite manejar errores, etc.

7.3.1.1 Módulos personalizados

Por supuesto, nosotros también podemos escribir nuestros propios módulos y para ello usamos `module.exports`. Escribimos el código de nuestro módulo, con todas las funciones locales que



queramos, luego hacemos un `module.exports = {}` y entre las llaves colocamos todo aquello que queramos exportar.

```
function suma(a,b){  
    return a + b;  
}  
  
function multiplicar(a,b){  
    return a * b;  
}  
  
module.exports = {  
    suma: suma,  
    multiplicar: multiplicar  
}
```

Asumiendo que el archivo anterior se llame "operaciones.js", nosotros podríamos requerirlo más tarde e otro archivo.js de la siguiente manera:

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

7.3.2 Comando NPM para gestión de paquetes

Por lo que respecta al uso de npm, es un comando que funciona desde la línea de comandos de NodeJS. Por tanto, lo tenemos que invocar con npm seguido de la operación que queramos realizar.

```
npm install async
```

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

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

Otras instrucciones posibles de npm son la de publicar paquetes (con "publish"), instalar globalmente (poniendo el flag -g al hacer el "install"), desinstalar, incluso premiar (puntuar paquetes hechos por otras personas), etc.

Tutor Ricardo Martín Espeche



7.3.3 Instalar paquetes de manera global

Como se ha dicho antes, npm instala los paquetes para un proyecto en concreto, sin embargo, existen muchos paquetes de Node que te facilitan tareas relacionadas con el sistema operativo. Estos paquetes, una vez instalados, se convierten en comandos disponibles en terminal, con los que se pueden hacer multitud de cosas. Existen cada vez más módulos de Node que nos ofrecen muchas utilidades para los desarrolladores, accesibles por línea de comandos, como Bower, Grunt, etc.

Las instrucciones para la instalación de paquetes de manera global son prácticamente las mismas que ya hemos pasado para la instalación de paquetes en proyectos. En este caso simplemente colocamos la opción "-g" que permite que ese comando se instale de manera global en tu sistema operativo.

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

Ese comando instala el módulo grunt-cli de manera global en tu sistema.



7.4 Crear servidor HTTP

Este es el módulo que nos sirve para trabajar con el protocolo HTTP, que es el que se utiliza en Internet para transferir datos en la Web. Nos servirá para crear un servidor HTTP que acepte solicitudes desde un cliente web.

Como queremos hacer uso de este módulo, en nuestro ejemplo empezaremos por requerirlo mediante la instrucción "require()".

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

A partir de este momento tenemos una variable http que en realidad es un objeto, sobre el que podemos invocar métodos que estaban en el módulo requerido. Por ejemplo, una de las tareas implementadas en el módulo HTTP es la de crear un servidor, que se hace con el método "createServer()". Este método recibirá un callback que se ejecutará cada vez que el servidor reciba una petición.

```
var server = http.createServer(function (peticion, respuesta) {  
    respuesta.end("Hola Mundo");  
});
```

La función callback que enviamos a createServer() recibe dos parámetros que son la petición y la respuesta. La petición por ahora no la usamos, pero contiene datos de la petición realizada. La respuesta la usaremos para enviarle datos al cliente que hizo la petición. De modo que "respuesta.end()" sirve para terminar la petición y enviar los datos al cliente.

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

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

Con esto le decimos al servidor que escuche en el puerto 3000, aunque podríamos haber puesto cualquier otro puerto que nos hubiera gustado. Además "listen()" recibe también una función callback que realmente no sería necesaria, pero que nos sirve para hacer cosas cuando el servidor se haya iniciado y esté listo. Simplemente, en esa función callback indico que estoy listo y escuchando en el puerto configurado.



7.4.1 Código completo

Como puedes ver, en muy pocas líneas de código hemos generado un servidor web que está escuchando en un puerto dado. El código completo es el siguiente:

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

Ahora podemos guardar ese archivo en cualquier lugar de nuestro disco duro, con extensión .js, por ejemplo servidor.js.

7.4.2 Iniciar el servidor

Ahora podemos ejecutar con Node el archivo que hemos creado. Nos vamos desde la línea de comandos a la carpeta donde hemos guardado el archivo servidor.js y ejecutamos el comando "node" seguido del nombre del archivo que pretendemos ejecutar:

```
node servidor.js
```

Entonces en consola de comandos nos debe aparecer el mensaje que informa que nuestro servidor está escuchando en el puerto 3000.

El modo de comprobar si realmente el servidor está escuchando a solicitudes de clientes en dicho puerto es acceder con un navegador. Dejamos activa esa ventana de línea de comandos y abrimos el navegador. Accedemos a:

```
http://localhost:3000
```

Entonces nos tendría que aparecer el mensaje "Hola Mundo".



7.5 Eventos en NodeJS

Lo primero que debemos entender es qué son eventos del lado del servidor, que no tienen nada que ver con los eventos Javascript que conocemos y utilizamos en las aplicaciones web del lado del cliente. Aquí los eventos se producen en el servidor y pueden ser de diversos tipos dependiendo de las librerías o clases que estemos trabajando.

Para hacernos una idea más exacta, pensemos por ejemplo en un servidor HTTP, donde tendríamos el evento de recibir una solicitud. Por poner otro ejemplo, en un stream de datos tendríamos un evento cuando se recibe un dato como una parte del flujo.

7.5.1 Modulo de eventos

Los eventos se encuentran en un módulo independiente que tenemos que requerir en nuestros programas creados con Node JS. Lo hacemos con la sentencia "require" que conocimos en artículos anteriores cuando hablábamos de módulos.

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

Dentro de esta librería o módulo tienes una serie de utilidades para trabajar con eventos.

Veamos primero el emisor de eventos, que encuentras en la propiedad EventEmitter.

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

7.5.2 Definir un Evento

En "Node" existe un bucle de eventos, de modo que cuando tú declaras un evento, el sistema se queda escuchando en el momento que se produce, para ejecutar entonces una función. Esa función se conoce como "callback" o como "manejador de eventos" y contiene el código que quieres que se ejecute en el momento que se produzca el evento al que la hemos asociado.

Primero tendremos que "instanciar" un objeto de la clase EventEmitter, que hemos guardado en la variable EmisorEventos en el punto anterior de este artículo.

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

Luego tendremos que usar el método on() para definir las funciones manejadoras de eventos, o su equivalente addEventListener(). Para emitir un evento mediante código Javascript usamos el método emit().

Tutor Ricardo Martín Espeche



Por ejemplo, voy a emitir un evento llamado "datos", con este código.

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

Ahora voy a hacer una función manejadora de eventos que se asocie al evento definido en "datos".

```
ee.on('datos', function(fecha){  
    console.log(fecha);  
});
```

Si deseamos aprovechar algunas de las características más interesantes de aplicaciones NodeJS quizás nos venga bien usar setInterval() y así podremos estar emitiendo datos cada cierto tiempo:

```
setInterval(function(){  
    ee.emit('datos', Date.now());  
}, 500);
```

Código completo del ejemplo de eventos y ejecución Con esto ya habremos construido un ejemplo NodeJS totalmente funcional. El código completo sería el siguiente:

```
var eventos = require('events');  
  
var EmisorEventos = eventos.EventEmitter;  
var ee = new EmisorEventos();  
ee.on('datos', function(fecha){  
    console.log(fecha);  
});  
setInterval(function(){  
    ee.emit('datos', Date.now());  
}, 500);
```

Esto lo podemos guardar como "eventos.js" o con cualquier otro nombre de archivos que deseemos. Lo guardamos en el lugar que queramos de nuestro disco duro.

Para ponerlo en ejecución nos vamos en línea de comandos hasta la carpeta donde hayamos colocado el archivo "eventos.js" o como quiera que lo hayas llamado y escribes el comando:

```
node eventos.js
```

Como resultado, veremos que empiezan a aparecer líneas en la ventana del terminal del sistema operativo, con un número, que es el "timestamp" de la fecha de cada instante inicial. Puedes salir del programa pulsando las teclas CTRL + c.

Tutor Ricardo Martín Espeche



Por si te lía esto de ejecutar archivos por medio de la línea de comando, a continuación, puedes ver una pantalla del terminal donde hemos puesto en marcha este pequeño ejercicio de eventos.

```
C:\Windows\system32\cmd.exe
c:\html\node>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: E212-93BB

Directorio de c:\html\node

09/12/2012  13:41    <DIR>          .
09/12/2012  13:41    <DIR>          ..
09/12/2012  13:44                238 evento.js
02/12/2012  21:35                244 serverhttp.js
                2 archivos             482 bytes
                2 dirs  24.179.249.152 bytes libres

c:\html\node>node evento
1355068691329
1355068691828
1355068692328
1355068692827
1355068693326
1355068693829
1355068694328
1355068694824
1355068695322
1355068695821
1355068696323
^C
c:\html\node>_
```




Capítulo 8

Express.js

8.1 Introducción básica

Express es sin duda el framework más conocido de node.js, es una extensión del poderoso connect y está inspirado en sinatra, además es robusto, rápido, flexible, simple...

Sin duda el éxito de express radica en lo sencillo que es usarlo, y además abarca un sin número de aspectos que muchos desconocen, pero son necesarios.

8.1.1 Instalación

Para comenzar, y suponiendo que tienes ya instalado node y npm, lo único que es necesario hacer es lo siguiente:

```
$ npm install -g express
```

A través de esta línea, estamos instalando globalmente express, el cual expone una interfaz interactiva en la línea de comandos. Por ejemplo, si escribimos `express -h` lo siguiente será mostrado:

```
Usage: express [options] [path]
```

Options:

<code>-s, --sessions</code>	add session support
<code>-t, --template <engine></code>	add template <engine> support (jade ejs). default=jade
<code>-c, --css <engine></code>	add stylesheet <engine> support (stylus). default=plain css
<code>-v, --version</code>	output framework version
<code>-h, --help</code>	output help information

Una vez instalado express puedes utilizar `express NOMBRE-DEL-APP` lo cual te creará una estructura personalizada y los archivos necesarios para comenzar a trabajar con el mismo. Haz



cd NOMBRE-DEL-APP && npm install para instalar automáticamente las dependencias, la estructura generada seria como la siguiente:

```
NOMBRE-DEL-APP
\
|- package.json
|- app.js
|- public
  \
    |- javascripts
    |- stylesheets
    |- images
|- routes
  \
    |- index.js
|- views
  \
    |- layout.jade
    |- index.jade
```

Como podemos ver, la estructura que se genera es muy útil y sencilla, se ha creado un package.json el cual contiene toda las dependencias necesarias de la aplicación.

8.1.2 app.js

Una aplicación escrita con express posee ciertos rasgos y cosas que no se pueden obviar, en app.js existe una estructura interna bien definida y es como sigue:

```
// app.js
var express = require('express')
  , routes = require('./routes')
var app = module.exports = express.createServer();
```

En esta primera parte se requieren los módulos o archivos externos necesarios para ejecutar la aplicación, en este caso el mismo express y routes que es ni más ni menos un archivo donde



se encuentran cada una de las rutas disponibles, routes es algo opcional, ya que tu mismo puedes ir generando las rutas a medida que vas avanzando. De igual forma, en esta parte se crea el servidor, mediante: `express.createServer()`, y es asignado a la variable `app` y de igual forma se “exporta” para que esté disponible en caso de que sea necesario ejecutarla como secundario.



8.2 Configuración

La segunda parte y una muy importante es la configuración:

```
// Configuration

app.configure(function() {
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

Es aquí donde se defines aspectos muy importantes para el funcionamiento correcto de nuestra aplicación, comúnmente conocido como middleware. Como podemos ver se define que el directorio donde se encuentran las vistas (templates) de nuestra aplicación, además se define en que lenguaje o motor está escrito `app.set("view engine", "jade");` en este caso en jade, además se define `bodyParser` y `methodOverride` mediante `app.use()` el `bodyParser` en palabras cortas, se encarga de decodificar la información que recibimos de un socket-cliente y lo expone en cada una de las requests mediante `request.body`. `methodOverride` provee soporte para el método faux HTTP. Como podemos ver se hace uso también del router de express, el cual nos proporciona la habilidad para estructurar nuestro código de una manera más sencilla mediante: `app.get('/path', function)`, y por último se define el middleware de archivos estáticos con la carpeta donde se encuentran los archivos que son "públicos".

En el código anterior se define una configuración global si tú quieres una configuración más específica, asígnale un nombre a tu ambiente mediante:

```
app.configure('personalizada', function() {

});
```

`personalizada` es una variable global que es pasada al momento de ejecutar tu programa mediante `NODE_ENV=personalizada` y es así como se origina la siguiente parte del código:



```
app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});
```

Es aquí donde se definen los ambientes más comunes, el de desarrollo (que es el default) y el de producción, en este ejemplo, se hace uso del middleware errorHandler, el cual te muestra las excepciones que el express encuentra, al habilitar showStack tu aplicación al encontrar un error te muestra un mensaje vistoso en el navegador (lo cual solo es útil para vos, por lo tanto no es recomendable mostrarlo en producción).

En este caso en particular se define la base de datos a utilizar. Por ejemplo:

```
// app.configure('development' ...
...
app.set('db-uri', 'http://localhost:5984/DB')
...
// app.configure('production' ...
...
app.set('db-uri', 'http://USR.iriscouch.com/DB')
...
```

Y así cualquier configuración que tú creas conveniente realizar.



8.3 Rutas

Las rutas son definitivamente la parte más importante de tu aplicación, porque si estas no están definidas, no existiría una interfaz para el cliente. En el ejemplo se generó esta dirección automáticamente, routes fueron definidas en la primera parte del archivo, y este definido como sigue:

```
app.get('/', routes.index);
```

Como podemos ver una ruta esta especificada de la siguiente forma:

```
app.VERBO(PATH, ACCIÓN);
```

app ya la conocemos.

VERBO puede ser:

- GET
- POST
- PUT
- DELETE

y así para cada uno de los verbos HTTP

- PATH: define la dirección de acceso.
- ACCION: que es lo que se tiene que hacer.

Por ejemplo, algo sencillo:

```
app.post('/user/new', function(request,response){  
  var body = request.body; // accede a la información enviada por el socket  
  // Guarda la información, haz lo que tengas que hacer  
});
```

O algo más global:

```
app.get('/blog/:id', function(req,res){ // accede a `/blog/id-largo` pero no  
a `/blog/id-largo/otro-di`  
  var id = req.params.id; // accede al id que se ha requerido  
})
```

Definiendo el formato:

Tutor Ricardo Martín Espeche



```
app.get('/u/:id/mensajes.:format', function(req,res){  
  var id = req.params.id;  
  var format = req.params.format;  
  if (format === 'json'){  
    res.json({status:200, id:id});  
  } else {  
    res.json({status:500});  
  }  
});
```

Otra forma seria:

```
app.get('/u(*)',function(req,res){})
```

En este caso concuerda con todas las direcciones que empiecen con u :

```
/u/alejandromg/  
/uasdlasdadklasd  
/u.json  
/u.json/potraosda
```

Este método tiene sus claras desventajas pero hay escenarios en el que es muy útil. Además de los verbos mencionados también se expone:

```
app.all(*,function)
```

Por la cual pasan todas las request, es muy útil en el caso que quieras detectar si es de un móvil, o cualquier cosa que tú quieras. Un aspecto interesante de las rutas es que si bien es cierto responden a una estructura definida, tu puedes definir ciertas acciones antes de dar una respuesta al usuario que hizo la petición, ¿a qué me refiero?, bueno supongamos que hay cierta área que solo está disponible para un usuario con cuenta, entonces tu podrías definir la función: verificarCuenta y luego continuar con la request, más específicamente de la siguiente manera:

```
function verificarCuenta(req,res,next){
```



```
if (req.session)
  next()
else
  res.redirect('/login')
}
// luego
app.get('/accesorestringido', verificarCuenta, function(req,res){
  // El usuario esta logueado
});
```

Y de manera más global:

```
app.VERB('PATH', ACCION1, ACCION2, ACCION-N, function(req,res){});
```

Como podemos ver, en este caso se define un parámetro extra next, mediante “next”, nos aseguramos que la aplicación siga su orden normal en el caso de que no haya ningún error. O hacer un `res.redirect()` para mandar al usuario a otra dirección.



8.4 Otros comandos

8.4.1 Listen

La aplicación debe estar disponible en algún puerto, y es aquí:

```
app.listen(3000);
```

Por default es el puerto 3000 que queda habilitado para la aplicación, si lo deseas puedes cambiar esto por algo como lo siguiente:

```
app.listen(process.env.PORT || 3000);
```

process.env.PORT es utilizada por procesos globales de los proveedores de hosting, para especificar el puerto a utilizar por las aplicaciones.

8.4.2 Helpers

Los dynamicHelpers te ayudan a definir una puente entre el cliente y el servidor, para crear variables o funciones que estén disponibles en ambos lados. Hay dos formas de habilitar este middleware, uno de ellos es:

1. Dentro de app.configure():

```
app.helpers(require('./path/to/helpers'));
```

En este caso ./path/to/helpers podría ser como este archivo

2. en cualquier otra parte de app.js

```
app.dynamicHelpers({  
  sitename: function(){  
    return 'NOMBRE-DEL-APP'  
  }  
});
```

Los dynamicHelpers es un objeto, pero cada uno de sus miembros son siempre funciones, por eso es necesario el return 'NOMBRE'.

Tutor Ricardo Martín Espeche



8.4.3 Error Handler

Express provee también de un sistema de errores:

```
app.error(function(err, req, res, next){  
  if (err) {  
    // Hacer algo con el error  
  } else {  
    next();  
  }  
});
```

Funciona también mediante: `app.use(function(err, req, res, next){})`

8.4.4 Sesiones

Para habilitar las sesiones es necesario hacer lo siguiente:

```
app.configure(function() {  
  app.use(express.bodyParser())  
  app.use(express.cookieParser('nhispano'))  
  app.use(express.session({secret: 'SECRET', store: store })))  
});
```

Particularmente, he encontrado errores al utilizar este método por lo tanto se hace lo siguiente:

```
var app = module.exports.app = process.app = express.createServer(  
  //small hack for make sessions works just fine  
  express.bodyParser(),  
  express.cookieParser('nhispano'),  
  express.session({secret: 'Node Hispano', store: store })),  
);
```

Luego de activar el middleware session, un nuevo objeto es creado en cada request y se puede acceder a él mediante: `req.session`.

Un módulo muy importante es connect-redis si usas redis, o connect-couchdb si usas couchDB, ambos disponibles mediante `npm install NombreDelPaquete`, los cuales te permiten manejar una cantidad mayor de sesiones al mismo tiempo.



8.4.5 Respuesta

El método `response`, tiene una gran cantidad de funciones y otros métodos disponibles. Por ejemplo:

```
res.sendFile('/path/to/archivo')
```

Envía un archivo directamente al usuario, muy útil para enviar archivos html directamente.

```
res.json(DATA)
```

Envía como respuesta un documento en formato json.

```
res.write()
```

Escribe información al cliente que hizo la petición, se puede utilizar múltiples veces en una misma petición:

```
req.write('<h1>Hola</h1>');  
req.write('<h2>mundo</h2>');  
res.send()
```

Es un método primario y se puede utilizar múltiples veces también.

```
res.send(); // 204  
res.send(new Buffer('wahoo'));  
res.send({ some: 'json' });  
res.send('<p>some html</p>');  
res.send('Sorry, cant find that', 404);  
res.send('text', { 'Content-Type': 'text/plain' }, 201);  
res.send(404);  
res.writeHead(CODE, Content-type)
```

Define el tipo de datos de la respuesta por ejemplo si usas `res.write()` con html como el ejemplo anterior, ocupas definir el tipo de contenido. `{"Content-type","text/html"}`



```
res.contentType(type)
```

Al igual que `res.writeHead()` nada más que en este caso solo pasas el `Content-type`.

```
res.redirect()
```

Redirecciona el cliente a una nueva URL.

```
res.render
```

Define y renderiza una template o layout, el uso más común es el siguiente:

```
res.render('TEMPLATE', {  
  // variables locales de TEMPLATE  
  layout: false, // por default busca dentro de views archivo llamado  
  `layout.jade|ejs`,  
  nombre: 'NOMBRE'  
})  
res.end()
```

Termina la respuesta. Necesaria al utilizar `res.write`, `res.send`.



Capítulo 9

REST

9.1 Introducción básica

REpresentational **S**tate **T**ransfer (REST) es un estilo de arquitectura para sistemas hipermedia distribuidos, tales como la World Wide Web. El centro de la arquitectura RESTful es el concepto de los recursos identificados por los identificadores de recursos universal (universal resource identifiers URIs). Estos recursos pueden ser manipulados usando un interfaz estándar, tales como el HTTP, y la información es intercambiada usando representaciones de estos recursos.



9.2 Desarrollando nuestro API RESTful

El primer paso es crear un directorio en tu entorno local para la aplicación, e iniciar un repositorio para guardar los cambios y que luego podamos desplegarlo por ejemplo en Heroku. Yo personalmente uso Git porque es una maravilla de la creación y porque a Heroku es lo que le mola ;)

```
C:\ md node-api-rest-example  
C:\ cd node-api-rest-example
```

Antes de empezar, necesitas tener Node instalado en tu computadora, para que funcione en tu entorno local de desarrollo.

El primer código que necesitamos escribir en una aplicación basada en Node es el archivo `package.json`. Éste archivo nos indica que dependencias vamos a utilizar en ella. Este archivo va en el directorio raíz de la aplicación:

Por lo tanto `package.json` tendrá:

```
{  
  "name": "node-api-rest-example",  
  "version": "2.0.0",  
  "dependencies": {  
    "mongoose": "~3.6.11",  
    "express": "^4.7.1",  
    "method-override": "^2.1.2",  
    "body-parser": "^1.5.1"  
  }  
}
```

Y ahora para descargar las dependencias escribimos lo siguiente en la consola y NPM (el gestor de paquetes de Node) se encargará de instalarlas.

```
$ npm install
```



9.2 Nuestro servidor Node.js

Con todo listo, podemos comenzar a codear de verdad. Creamos un fichero llamado app.js en el directorio raíz que será el que ejecute nuestra aplicación y arranque nuestro server. Crearemos en primer lugar un sencillo servidor web para comprobar que tenemos todo lo necesario instalado, y a continuación iremos escribiendo más código.

```
var express = require("express"),
    app = express(),
    bodyParser = require("body-parser"),
    methodOverride = require("method-override");
mongoose = require('mongoose');

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(methodOverride());

var router = express.Router();

router.get('/', function(req, res) {
  res.send("Hello World!");
});

app.use(router);

app.listen(3000, function() {
  console.log("Node server running on http://localhost:3000");
});
```

¿Qué hace este código? Muy sencillo, las primeras líneas se encargan de incluir las dependencias que vamos a usar, algo así como los includes en C o PHP, o los import de Python. Importamos Express para facilitarnos crear el servidor y realizar llamadas HTTP. Con http creamos el servidor que posteriormente escuchará en el puerto 3000 de nuestro ordenador (O el que nosotros definamos).

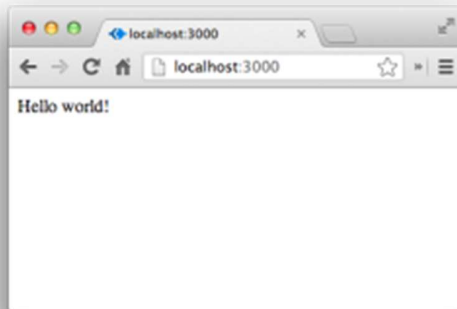
Con bodyParser permitimos que pueda parsear JSON, methodOverride() nos permite implementar y personalizar métodos HTTP.

Podemos declarar las rutas con app.route(nombre_de_la_ruta) seguido de los verbos .get(), .post(), etc... y podemos crear una instancia para ellas con express.Router(). En este primer ejemplo vamos hacer que sólo reciba una petición GET del navegador y muestre en el mismo la frase Hello World

Para ejecutar este pequeño código sólo tienes que escribir en consola lo siguiente y abrir un navegador con la url <http://localhost:3000>

```
C:\node_path\node app.js  
Node server running on http://localhost:3000
```

Si todo va bien, esto es lo que se verá:





9.3 Creando los modelos de nuestra API REST

En esta parte vamos a crear un modelo usando Mongoose para poder guardar la información en la base de datos siguiendo el esquema. Como base de datos vamos a utilizar MongoDB. MongoDB es una base de datos Open Source NoSQL orientada a documentos tipo JSON, lo cual nos viene que ni pintado para entregar los datos en este formato en las llamadas a la API.

Para este ejemplo vamos a crear una base de datos de series de TV, por tanto, vamos a crear un modelo (Archivo: models/tvshow.js) que incluya la información de una serie de TV, como pueden ser su título, el año de inicio, país de producción, una imagen promocional, número de temporadas, género y resumen del argumento:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

var tvshowSchema = new Schema({
  title: { type: String },
  year: { type: Number },
  country: { type: String },
  poster: { type: String },
  seasons: { type: Number },
  genre: { type: String, enum:
    ['Drama', 'Fantasy', 'Sci-Fi', 'Thriller', 'Comedy']
  },
  summary: { type: String }
});

module.exports = mongoose.model('TVShow', tvshowSchema);
```

Con esto ya podemos implementar la conexión a la base de datos en el archivo app.js* añadiendo las siguientes líneas:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/tvshows');
```

Quedando así el código de app.js:

```
var express = require("express"),
    app = express(),
    http = require("http"),
    server = http.createServer(app),
    mongoose = require('mongoose');

app.use(bodyParser.urlencoded({ extended: false }));
```



```
app.use(bodyParser.json());
app.use(methodOverride());

var router = express.Router();

router.get('/', function(req, res) {
  res.send("Hello World!");
});

app.use(router);

mongoose.connect('mongodb://localhost/tvshows', function(err, res) {
  if(err) {
    console.log('ERROR: connecting to Database. ' + err);
  }
  app.listen(3000, function() {
    console.log("Node server running on http://localhost:3000");
  });
});
```

Para que esto funcione en nuestro entorno local, necesitamos tener instalado MongoDB.

Para poder iniciar MongoDB debes ejecutar en otra terminal:

```
C:\mongodb_path\mongod
all output going to: /usr/local/var/log/mongodb/mongo.log
```

Con Mongo arrancado ya podemos ejecutar la aplicación como en la parte anterior con node app.js desde la terminal, si todo va bien tendremos algo en la pantalla como esto:

```
C:\node_path\node app.js
Node server running on http://localhost:3000
Connected to Database
```

Ahora desde otra terminal, podemos entrar al shell de MongoDB y comprobar que la base de datos se ha creado correctamente. Para ello ejecutamos el comando mongo

```
C:\mongodb_path\mongo
MongoDB shell version: 2.4.1
connecting to: test
> use tvshows
switched to db tvshows
> show dbs
local      0.078125GB
tvshows    (empty)
>_
```



Ya tenemos todo configurado y listo para albergar los datos, sólo nos queda crear las rutas que definirán las llamadas a la API para poder guardar y consultar la información.

9.4 Implementando los controladores de nuestras rutas o endpoints

Los controladores de las rutas de nuestro API los vamos a crear en un archivo separad que llamaremos controllers/tvshows.js. Gracias a exports conseguimos modularizarlo y que pueda ser llamado desde el archivo principal de la aplicación. El código de a continuación es el comienzo del archivo con la primera función que será la que devuelva todos los registros almacenados:

```
//File: controllers/tvshows.js
var mongoose = require('mongoose');
var TVShow = mongoose.model('TVShow');

//GET - Return all tvshows in the DB
exports.findAllTVShows = function(req, res) {
  TVShow.find(function(err, tvshows) {
    if(err) res.send(500, err.message);

    console.log('GET /tvshows')
    res.status(200).jsonp(tvshows);
  });
};
```

De esta manera tan sencilla, al llamar a la función findAllTVShows se envía como respuesta toda la colección de tvshows almacenada y en formato JSON. Si queremos que sólo nos devuelva un registro con un identificador único, tenemos que crear una función tal que la siguiente:

```
//GET - Return a TVShow with specified ID
exports.findById = function(req, res) {
  TVShow.findById(req.params.id, function(err, tvshow) {
    if(err) return res.send(500, err.message);

    console.log('GET /tvshow/' + req.params.id);
    res.status(200).jsonp(tvshow);
  });
};
```

Con las funciones find() y findById() podemos buscar en la base de datos a partir de un modelo. Ahora desarrollaré el resto de funciones que permiten insertar, actualizar y borrar registros de



la base de datos. La función de a continuación sería la correspondiente al método POST y lo que hace es añadir un nuevo objeto a la base de datos:

```
//POST - Insert a new TVShow in the DB
exports.addTVShow = function(req, res) {
  console.log('POST');
  console.log(req.body);

  var tvshow = new TVShow({
    title:    req.body.title,
    year:    req.body.year,
    country:  req.body.country,
    poster:   req.body.poster,
    seasons: req.body.seasons,
    genre:    req.body.genre,
    summary:  req.body.summary
  });

  tvshow.save(function(err, tvshow) {
    if(err) return res.status(500).send( err.message);
    res.status(200).jsonp(tvshow);
  });
};
```

Primero creamos un nuevo objeto tvshow siguiendo el patrón del modelo, recogiendo los valores del cuerpo de la petición, lo salvamos en la base de datos con el comando .save() y por último lo enviamos en la respuesta de la función.

La siguiente función nos permitirá actualizar un registro a partir de un ID. Primero buscamos en la base de datos el registro dado el ID, y actualizamos sus campos con los valores que devuelve el cuerpo de la petición:

```
//PUT - Update a register already exists
exports.updateTVShow = function(req, res) {
  TVShow.findById(req.params.id, function(err, tvshow) {
    tvshow.title    = req.body.petId;
    tvshow.year     = req.body.year;
    tvshow.country  = req.body.country;
    tvshow.poster   = req.body.poster;
    tvshow.seasons  = req.body.seasons;
    tvshow.genre    = req.body.genre;
    tvshow.summary  = req.body.summary;

    tvshow.save(function(err) {
```



```
        if(err) return res.status(500).send(err.message);
        res.status(200).jsonp(tvshow);
    });
});
```

Y por último para completar la funcionalidad CRUD de nuestra API, necesitamos la función que nos permita eliminar registros de la base de datos y eso lo podemos hacer con el código de a continuación:

```
//DELETE - Delete a TVShow with specified ID
exports.deleteTVShow = function(req, res) {
    TVShow.findById(req.params.id, function(err, tvshow) {
        tvshow.remove(function(err) {
            if(err) return res.status(500).send(err.message);
            res.status(200).send();
        })
    });
};
```

Como puedes ver, usamos de nuevo el método `.findById()` para buscar en la base de datos y para borrarlo usamos `.remove()` de la misma forma que usamos el `.save()` para salvar.

Ahora tenemos que unir estas funciones a las peticiones que serán nuestras llamadas al API. Volvemos a nuestro archivo principal, `app.js` y declaramos las rutas, siguiendo las pautas de Express v.4

```
var TVShowCtrl = require('./controllers/tvshows');

// API routes
var tvshows = express.Router();

tvshows.route('/tvshows')
    .get(TVShowCtrl.findAllTVShows)
    .post(TVShowCtrl.addTVShow);

tvshows.route('/tvshows/:id')
    .get(TVShowCtrl.findById)
    .put(TVShowCtrl.updateTVShow)
    .delete(TVShowCtrl.deleteTVShow);

app.use('/api', tvshows);
```



9.4.1 Probando nuestro API REST en el navegador

A continuación, voy a probar una herramienta online que nos permite jugar con las llamadas al API y poder consultar y almacenar datos para probarla y ver su funcionamiento un poco más claro.

Para ello nos dirigimos a restconsole.com que es una extensión de Google Chrome, que permite hacer lo que queremos de una manera visual y sencilla.



Antes de probarlo, debemos tener mongo y el servidor node de nuestra app corriendo. Una vez hecho esto introducimos los siguientes datos para hacer una llamada POST que almacene un registro en la base de datos.

- Target: <http://localhost:3000/tvshow> (Dónde está ejecutándose la aplicación y la llamada al método POST que hemos programado)
- Content-Type: application/json (en los dos inputs que nos dan)
- Request-Payload: Aquí va el cuerpo de nuestra petición, con el objeto JSON siguiente (por ejemplo):

```
{
  "title": "LOST",
  "year": 2004,
  "country": "USA",
  "poster": "http://ia.media-
imdb.com/images/M/MV5BMjA3NzMyMzU1MV5BMl5BanBnXkFtZTcwNjc1ODUwMg@@._V1_
SY317_CR17,0,214,317_.jpg",
  "seasons": 6,
  "genre": "Sci-Fi",
  "summary": "The survivors of a plane crash are forced to live with
each other on a remote island, a dangerous new world that poses unique
threats of its own."
}
```

Pulsamos SEND y si todo va bien, la petición se realizará y abajo de la aplicación REST Console veremos algo como esto:

¿Cómo comprobamos si se ha guardado en nuestra base de datos? Muy sencillo, en nuestro terminal ejecutamos el Shell de mongo con el comando mongod e introducimos los siguientes comandos:

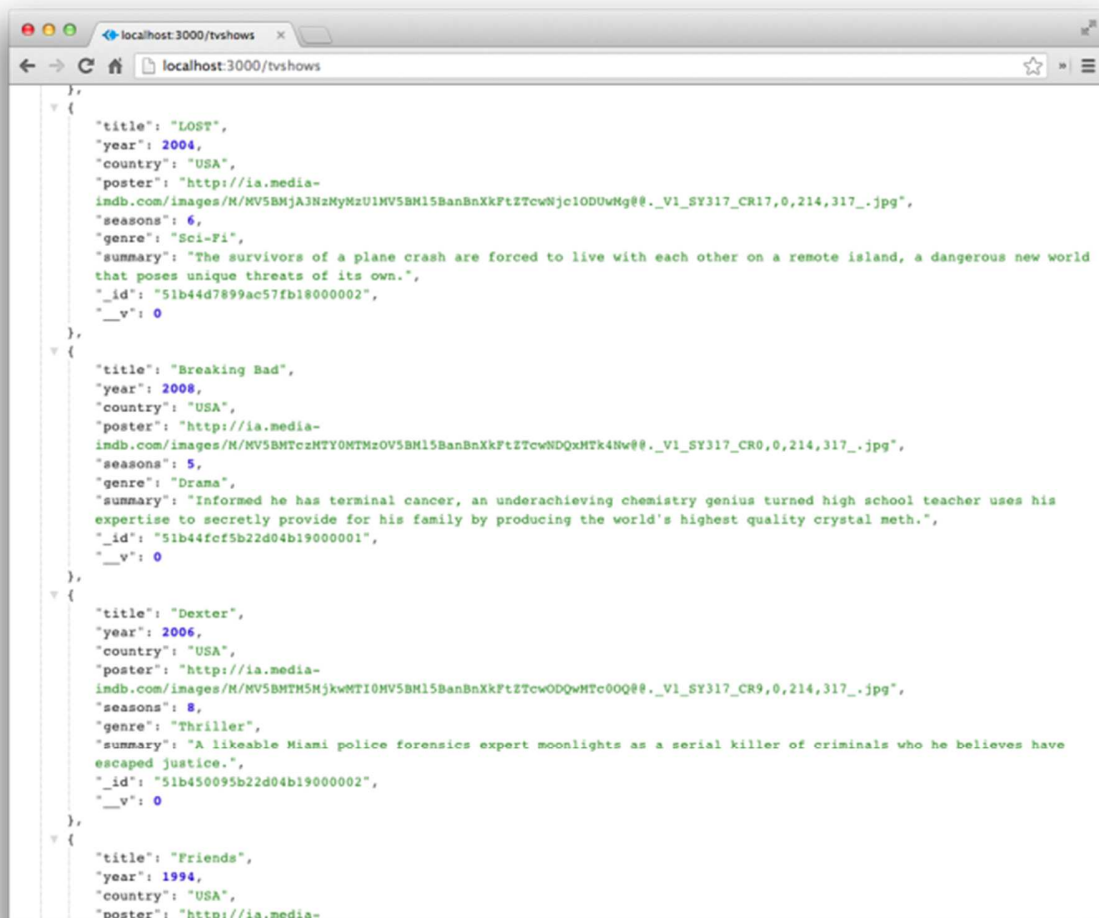
```
C:\mongodb_path\mongo
MongoDB shell version: 2.4.1
connecting to: test
> show databases
tvshows      0.203125GB
> use tvshows
switched to db tvshows
> show collections
system.indexes
tvshows
> db.tvshows.find()
{ "title" : "LOST", "year" : 2004, "country" : "USA", "poster" :
"http://ia.media-
imdb.com/images/M/MV5BMjA3NzMyMzU1MV5BMl5BanBnXkFtZTcwNjc1ODUwMg@@._V1
_SY317_CR17,0,214,317_.jpg", "seasons" : 6, "genre" : "Sci-Fi",
"summary" : "The survivors of a plane crash are forced to live with
```

Tutor Ricardo Martín Espeche



```
each other on a remote island, a dangerous new world that poses unique threats of its own.", "_id" : ObjectId("51b44d7899ac57fb18000002"), "__v" : 0 }
```

Y ahí la tenemos, puedes probar a introducir alguna más, para tener mayor contenido con el que probar. Una vez introduzcas varios registros, puedes probar a llamarlos a través de la petición GET que hemos preparado: <http://localhost:3000/tvshows> la cuál te mostrará algo parecido a esto:



También podemos llamar a un sólo registro gracias a la petición GET `tvshows/:id` que programamos, si ejecutamos por ejemplo <http://localhost:3000/tvshow/51b44d7899ac57fb18000002> nos devolverá un único objeto:



Los métodos restantes PUT y DELETE funcionan de manera parecida al POST sólo que hay que pasarte el valor del ID del objeto que queremos actualizar o borrar. Te invito a que lo pruebes en la Tutorial de NodeJS, creando un API REST - REST Console.

Con esto tendríamos el funcionamiento básico y programación de lo que sería una API REST. Como puedes ver es bastante sencillo y utilizas Javascript en todas partes, como lenguaje de servidor (Node), como formato de datos (JSON) y como base de datos (MongoDB)