



## Arquitecturas SOA y cloud

Laboratorio 3.  
Servicio Mis rutas

# Contenido

1. Introducción.....	3
2. Node.js.....	3
3. Express.js.....	4
4. El servicio RESTful.....	5
5. Probar el servicio RESTful.....	7
6. Consumir el servicio desde una webapp.....	10
6.1 Webapp de prueba.....	10
6.2 CORS.....	13
6.3 Backbone.js.....	14
7. Heroku.....	17
7.1 Instalar cliente Heroku.....	17
7.2 Preparar la aplicación.....	18
7.3 Crear una aplicación.....	20
8. Extensiones.....	21
8.1 Bases de datos.....	21
8.2 Objetos georreferenciados.....	21
8.3 Multiusuario.....	21
8.4 Open your mind .....	21

# 1. Introducción

Esta práctica persigue los siguientes objetivos:

- 1.- Comprender la plataforma Node.js.
- 2.- Diseñar servicios RESTful con Node.js.
- 3.- Consumir servicios RESTful con Backbone.js.
- 4.- Desplegar aplicaciones en el cloud con Heroku.

En esta práctica vamos a diseñar un servicio RESTful en Node.js, capaz de gestionar una colección de rutas remotas. Este servicio será consumido por la webapp Mis Rutas que se implementó en el laboratorio anterior. Para ello, primero instalaremos la plataforma Node.js, así como todos los paquetes que son necesarios para proceder a continuación a implementar una primera versión básica del servicio. Posteriormente modificaremos la webapp Mis Rutas para que sincronice sus datos con el servicio, utilizando para ello las utilidades de Backbone.js. Después desplegaremos nuestro servicio en la plataforma PaaS Heroku. Finalmente, se plantearán algunas extensiones que podrá efectuar el alumno aventajado.

## 2. Node.js

Ya hemos visto en el curso que Node.js es una plataforma muy eficiente específicamente diseñada para desarrollar aplicaciones de servidor escalables. En esta práctica utilizaremos Node.js para desarrollar un servicio RESTful que nos permita gestionar una colección de rutas remotas.

El primer paso consiste en instalar la plataforma. Para ello, accederemos a la página web del proyecto:

<http://nodejs.org>

Descargaremos el paquete adecuado para nuestra plataforma y lo instalaremos.

Una vez instalado comprobaremos que la plataforma está disponible a través de la línea de comandos, abriendo una consola del sistema y ejecutando el comando:

```
> node
```

Si la plataforma está disponible, el intérprete interactivo de Node.js comenzará su ejecución. En el intérprete interactivo podemos ejecutar cualquier sentencia JavaScript. Si introducimos `.help` obtendremos todos los comandos disponibles, que pueden ser invocados poniendo el prefijo `.`

```
> .help
break  Sometimes you get stuck, this gets you out
clear  Alias for .break
exit   Exit the repl
help   Show repl options
load   Load JS from a file into the REPL session
save   Save all evaluated commands in this REPL session to a file
```

Uno de estos comandos, `.exit`, permite salir del entorno.

El siguiente paso consiste en escoger un IDE para crear nuestro proyecto. Recomendamos utilizar eclipse, ya que hemos estado trabajando con él en teoría. Para ello, deberemos instalar un plugin que nos permita trabajar con Node.js en uestros proyectos: nodeclipse.

En eclipse, seleccionamos Help > Eclipse Marketplace ... En la caja de búsqueda introducimos "node" y seleccionamos Nodeclipse.

Tras el proceso de instalación deberemos reiniciar eclipse y entonces obtendremos la posibilidad de crear proyectos Node.js. Seleccionamos File > New > Project ... Nodeclipse > Node.js Project y creamos un proyecto Node.js vacío que contendrá nuestro servicio RESTful (e.g. ServicioMisRutas)

Ahora creamos un nuevo fichero JavaScript llamado app.js seleccionando File > New > Other ... Nodeclipse > JavaScript File, y lo guardamos en el directorio principal de la aplicación.

```
ServicioMisRutas
  \_ app.js
```

Abrimos el fichero app.js y crearemos el ejemplo “Hola Mundo”, simplemente para verificar que todo funciona correctamente.

```
console.log('Hola mundo!!');
```

A continuación guardamos el fichero y lo ejecutamos desde el IDE, presionando con el botón derecho del ratón sobre app.js y seleccionando Run As > Node Application. La salida debería de ser visible ahora.

### 3. Express.js

En este punto pondremos en marcha el framework web Express, que usaremos para desarrollar nuestro servicio RESTful. El primer paso consiste en instalar el framework. Para ello, abriremos una consola del sistema, nos moveremos al directorio raíz (usando los clásicos comandos cd) y ejecutaremos el siguiente comando:

```
> npm install express
```

Ahora el framework está instalado en el proyecto actual y podemos empezar a usarlo.

El siguiente paso consiste en crear un servidor Express sencillo, que por un lado sirva todo el contenido estático que hemos generado en el laboratorio pasado (la webapp Mis Rutas), y que por otro lado devuelva el clásico Hola Mundo de manera dinámica, esto es, registrando una ruta y generando dinámicamente dicho contenido. Para ello, crearemos una nueva carpeta en nuestro proyecto, denominada /static, y aquí insertaremos todo el contenido del laboratorio 2. Tras esto, la estructura de nuestro proyecto se parecerá a la siguiente:

```
ServicioMisRutas
  \_ app.js
  \_ static
      \_ index.html
      \_ css
      \_ js
          \_ app.js
          \_ externo
              \_ jquery.js
              \_ ...
          \_ modelos
              \_ Ruta.js
              \_ Rutas.js
          \_ vistas
              \_ EditarRuta.js
              \_ ListarRutas.js
              \_ Mapa.js
              \_ NuevaRuta.js
```

A continuación crearemos el servidor Express en el fichero app.js (el que reside en el directorio raíz):

```
var express = require('express');
var app = express();
app.use(express.static('static'));
app.listen(8080);
```

A continuación abriremos el navegador y accederemos a nuestra webapp Mis Rutas, usando URLs del tipo:

`http://localhost:8080/xxx`

En concreto, la siguiente URL debería de cargar nuestra webapp, y debería de estar plenamente funcional:

`http://localhost:8080/index.html`

Como es posible observar, todos los recursos estáticos están accesibles a través de la URL raíz. Si se desea ubicar los recursos estáticos bajo otra URL, lo único que hay que hacer es montar el middleware `express.static()` bajo otra URL, por ejemplo:

```
app.use('/static' express.static('static'));
```

Con este ejemplo, los recursos estáticos están disponibles a través de URLs del tipo:

`http://localhost:8080/static/xxx`

El siguiente paso consiste en crear el ejemplo “Hola Mundo” dinámico. Para ello, añadiremos una nueva ruta a Express, usando el siguiente código:

```
app.get('/hello', function(req, res) {
    res.end('Hola Mundo!!');
});
```

Si introducimos la URL <http://localhost:8080/hello> en el navegador obtendremos el mensaje de texto “Hola Mundo!!”.

Ahora ya disponemos de un servidor base a partir del cual construir nuestro servicio RESTful. En la siguiente sección analizamos cómo construirlo.

## 4. El servicio RESTful

El servicio a implementar debe responder a las siguientes peticiones:

- HTTP POST sobre la URL `/misrutas/rutas`: para crear nuevas rutas. La petición contendrá los datos de la nueva ruta codificados en JSON. Esta llamada debe devolver la ruta creada, con su nuevo identificador asignado, también codificado en JSON.
- HTTP GET sobre la URL `/misrutas/rutas`: para recuperar una colección con todas las rutas registradas en el servicio. Esta colección será codificada en JSON.
- HTTP GET sobre URLs del tipo `/misrutas/rutas/<id>`: para recuperar una ruta concreta, donde `<id>` representa el identificador de dicha ruta. La ruta se codifica en JSON.
- HTTP PUT sobre URLs del tipo `/misrutas/rutas/<id>`: para actualizar una ruta concreta, donde `<id>` representa el identificador de dicha ruta. La petición contendrá los datos de la ruta a modificar, codificados en JSON. Esta llamada debe devolver la ruta modificada, también codificada en JSON.

- HTTP DELETE sobre URLs del tipo `/misrutas/rutas/<id>`: para eliminar una ruta concreta, donde `<id>` representa el identificador de dicha ruta.

Con esta información, nos ponemos manos a la obra. Como se puede observar, nuestra aplicación va a procesar el contenido de las peticiones HTTP, que se espera estén codificados en formato JSON. Para facilitar el análisis de este contenido utilizaremos el middleware `body-parser`, que nos permite acceder a dicho contenido en forma de objeto JavaScript bajo el atributo `req.body`.

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
app.use(bodyParser.json());
app.use(express.static('static'));
```

Por otro lado, nuestro servicio custodiará una colección de rutas, que constituyen su base de datos. En esta primera versión del servicio, modelaremos dicha base de datos como un vector. Además, cuando creamos nuevas rutas, es necesario asignarles un identificador único. Para ello, mantendremos un contador.

```
var rutas = [];
var nextId = 0;
```

Ya sólo falta definir las distintas rutas expuestas al comienzo de esta sección. Iremos una por una, empezando por la ruta que permite crear nuevas rutas:

```
app.post('/misrutas/rutas', function(req, res) {
  console.log('POST /misrutas/rutas');
  var ruta = req.body;
  ruta.id = nextId++;
  rutas.push(ruta);
  res.send(ruta);
});
```

Como se puede observar, accedemos a la nueva ruta a través del atributo `req.body`, donde el middleware `body-parser` ha dejado previamente el contenido de la petición codificado en JSON. Como se puede observar, al enviar con `res.send()` un objeto, éste es transformado automáticamente a JSON.

La siguiente ruta a definir es la de recuperación:

```
app.get('/misrutas/rutas', function(req, res) {
  console.log('GET /misrutas/rutas');
  res.send(rutas);
});
```

Basta con enviar la colección de rutas. Como es un objeto, Express lo transformará automáticamente a JSON.

La siguiente ruta nos permite recuperar elementos por identificador:

```
app.get('/misrutas/rutas/:id', function(req, res) {
  console.log('GET /misrutas/' + req.params.id);
  for (var i = 0; i < rutas.length; i++) {
    if (rutas[i].id == req.params.id) {
      res.send(rutas[i]);
      return;
    }
  }
  res.status(404).send('Not found');
});
```

La siguiente ruta permite actualizar elementos:

```
app.put('/misrutas/rutas/:id', function(req, res) {
```

```

    console.log('PUT /misrutas/' + req.params.id);
    console.log(req.originalUrl);
    for (var i = 0; i < rutas.length; i++) {
        if (rutas[i].id == req.params.id) {
            rutas[i] = req.body;
            res.send(rutas[i]);
            return;
        }
    }
    res.status(404).send('Not found');
});

```

Por último, la ruta de eliminación:

```

app.delete('/misrutas/rutas/:id', function(req, res) {
    console.log('DELETE /misrutas/' + req.params.id);
    console.log(req.originalUrl);
    for (var i = 0; i < rutas.length; i++) {
        if (rutas[i].id == req.params.id) {
            rutas.splice(i, 1);
            res.status(204).send();
            return;
        }
    }
    res.status(404).send('Not found');
});

```

Una vez definidas todas las rutas, el siguiente paso es arrancar el servidor:

```

app.listen(8080);

```

## 5. Probar el servicio RESTful

En este apartado verificaremos que el servicio RESTful implementado en el apartado anterior funciona como debe. Como sabemos, las peticiones HTTP GET pueden ser simuladas fácilmente desde un navegador simplemente introduciendo la URL en la barra de direcciones. Sin embargo, con HTTP GET sólo podemos hacer consultas, por lo que podremos comprobar muy poco si no existen datos previamente.

Por tanto, crearemos un programa Node.js que consuma el servicio. Para ello, creamos un nuevo fichero en el directorio raíz denominado test.js. Este cliente hará uso del paquete “request” que ya hemos presentado en teoría. Además, lo haremos interactivo, de manera que el usuario podrá definir la acción a realizar por línea de comandos, y el cliente reproducirá dicha acción contra el servicio RESTful.

Para ello necesitaremos acceder a la entrada estándar. Como sabemos, la entrada estándar reside en la variable process.stdin. Se trata de un stream de entrada (Readable) y para acceder a él deberíamos de usar la interfaz definida en el API de Node.js y que hemos explorado brevemente en teoría. Este API no es fácil de controlar y requiere cierta experiencia. No lo utilizaremos de manera directa, sino que usaremos otro módulo core de Node.js, que nos permite leer líneas de texto desde la entrada estándar con mayor facilidad, se trata del módulo “readline”. Este módulo nos permite definir un prompt, a través del cual el usuario introducirá información, y nos notifica tras cada nueva línea que proceda de la entrada estándar. Para ello, primero es necesario crear una interfaz que envuelva entrada y salida estándar:

```

var readline = require('readline');
var request = require('request');

var rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

```

Definimos el prompt a utilizar y lo mostramos por pantalla:

```
rl.setPrompt('mis rutas>');
rl.prompt();
```

Lo siguiente que debemos hacer es recuperar los comandos del usuario. Para ello nos suscribimos al evento 'line' de la interfaz creada:

```
rl.on('line', function(line) {
    ...
})
```

En el interior del manejador anterior sucederá el resto del programa. Analizaremos el comando introducido por el usuario y actuaremos de manera conveniente. Soportaremos los siguientes comandos:

- help: permite mostrar el mensaje de ayuda
- exit: para salir del cliente interactivo
- get [<id>]: recupera todas las rutas, o sólo la ruta con el <id> especificado
- new <título>: crea una nueva ruta con el título especificado. Si no se especifica título definiremos uno por defecto.
- update <id> <título>: modifica la ruta con el <id> especificado cambiando su título
- delete <id>: elimina la ruta con el <id> especificado

La estructura básica del manejador será como sigue:

```
rl.on('line', function(line) {
    var args = line.split(' ');
    switch (args[0]) {
        case 'exit': /* salir del cliente */
            ...
            break;
        case 'help': /* mostrar mensaje de ayuda */
            ...
            break;
        case 'get': /* recuperar rutas */
            ...
            break;
        case 'new': /* crear nueva ruta */
            ...
            break;
        case 'update': /* actualizar ruta */
            ...
            break;
        case 'delete': /* eliminar ruta */
            ...
            break;
        default:
            rl.prompt();
    }
});
```

Para salir del cliente, introduciremos un código similar al siguiente:

```
console.log('Bye');
rl.close();
process.exit();
```

Para mostrar el mensaje de ayuda, algo como esto:

```
console.log('Comandos disponibles:');
console.log('  - help: mostrar este mensaje');
```



```

console.log(' - exit: salir del cliente');
console.log(' - get [<id>]: recupera todas las rutas o bien la ruta con el <id> especificado');
console.log(' - new <titulo>: crea una nueva ruta con el titulo especificado');
console.log(' - update <id> <titulo>: modifica la ruta con el <id> especificado cambiando su titulo');
console.log(' - delete <id>: elimina la ruta con el <id> especificado');
rl.prompt();

```

Ahora viene lo realmente interesante, consumir el servicio RESTful. Primero, la recuperación de información:

```

var url = 'http://localhost:8080/misrutas/rutas';
if (args.length > 1) url += '/' + args[1];
var opts = {
  url: url,
  method: 'GET'
};
request(opts, function(error, res, body) {
  if (error || res.statusCode !== 200) {
    console.log('Error: ' + error);
  } else {
    console.log(body);
  }
  rl.prompt();
});

```

Después, la creación de nuevas rutas:

```

var titulo = "Default";
if (args.length > 1) titulo = args[1];
var opts = {
  url: 'http://localhost:8080/misrutas/rutas',
  method: 'POST',
  json: true,
  body: { titulo: titulo }
};
request(opts, function(error, res, body) {
  if (error || res.statusCode !== 200) {
    console.log('Error: ' + error);
  } else {
    console.log(body);
  }
  rl.prompt();
});

```

Actualización:

```

if (args.length < 3) {
  console.log("Es necesario especificar el <id> y el nuevo <titulo>");
  rl.prompt();
  return;
}
var opts = {
  url: 'http://localhost:8080/misrutas/rutas/' + args[1],
  method: 'PUT',
  json: true,
  body: { id: args[1], titulo: args[2] }
};
request(opts, function(error, res, body) {
  if (error || res.statusCode !== 200) {
    console.log('Error: ' + error);
  } else {
    console.log(body);
  }
  rl.prompt();
});

```

Y, finalmente, la eliminación de rutas:

```
if (args.length < 2) {
  console.log("Es necesario especificar el <id>");
  rl.prompt();
  return;
}
var opts = {
  url: 'http://localhost:8080/misrutas/rutas/' + args[1],
  method: 'DELETE'
};
request(opts, function(error, res, body) {
  if (error || res.statusCode !== 204) {
    console.log('Error: ' + error);
  } else {
    console.log(body);
  }
  rl.prompt();
});
```

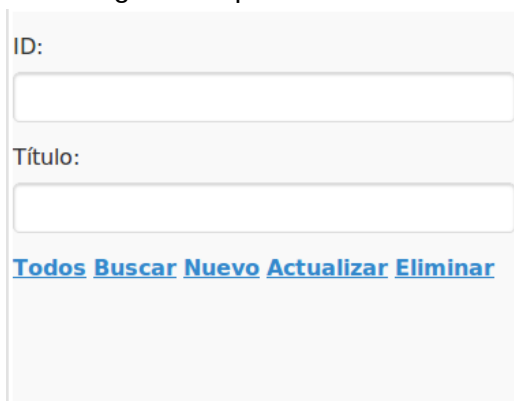
Una vez definido el cliente en test.js, sólo resta ejecutarlo y comprobar que nuestro servicio funciona adecuadamente. Recordar, antes de ejecutar test.js, arrancar el servicio ejecutando para ello app.js.

## 6. Consumir el servicio desde una webapp

Una vez tenemos la certeza de que nuestro servicio funciona de manera adecuada, es el momento de consumirlo desde una webapp. En esta sección, primero crearemos una pequeña webapp de prueba que nos permita practicar con la primitiva básica de jQuery \$.ajax(). Después pasaremos a utilizar Backbone.js y experimentaremos lo sencillo que es acceder a servicios RESTful utilizando este framework; para ello reutilizaremos la webapp del laboratorio 2 y la modificaremos para que el modelo se sincronice con nuestro servicio RESTful.

### 6.1 Webapp de prueba

En esta sección crearemos una pequeña webapp de prueba que nos permita consumir el servicio RESTful utilizando directamente la primitiva \$.ajax() de jQuery. Será una aplicación muy sencilla, con la siguiente apariencia:



Como se puede observar en la imagen, se trata de una webapp que consta de dos campos de texto, un conjunto de comandos (links) y un panel de información bajo estos links, que nos permite mostrar el resultado de los comandos anteriores. La idea es muy parecida al cliente en Node.js, pero ahora desde una interfaz web.

Crearemos el fichero test.html en la carpeta /static/ del proyecto, obteniendo un resultado parecido

a éste:

```
ServicioMisRutas
  \_ app.js
  \_ static
    \_ index.html
    \_ test.html
    \_ css
    \_ js
      \_ app.js
      \_ externo
        \_ jquery.js
        \_ ...
      \_ modelos
        \_ Ruta.js
        \_ Rutas.js
      \_ vistas
        \_ EditarRuta.js
        \_ ListarRutas.js
        \_ Mapa.js
        \_ NuevaRuta.js
```

Este ejemplo podría usar jQuery Mobile o no, es simplemente para practicar con \$.ajax(). A continuación mostramos la plantilla básica:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<link rel="stylesheet" href="css/jquery.mobile-1.4.5.css" />
<script src="js/externo/jquery-1.11.3.js"></script>
<script src="js/externo/jquery.mobile-1.4.5.js"></script>
<script src="js/externo/underscore.js"></script>
<script src="js/externo/backbone.js"></script>
<script>

$(function() {

    $('#todos').click(function() { /* mostrar todas las rutas */ });
    $('#buscar').click(function() { /* mostrar una ruta en concreto */ });
    $('#nuevo').click(function() { /* crear una nueva ruta */ });
    $('#actualizar').click(function() { /* actualizar una ruta */ });
    $('#eliminar').click(function() { /* eliminar una ruta */ });

});
</script>
</head>
<body>

<p>ID: <input type="text" id="txtId" name="txtId" ></p>
<p>Título: <input type="text" id="txtTitulo" name="txtTitulo" ></p>
<a href="#" id="todos">Todos</a>&nbsp;
<a href="#" id="buscar">Buscar</a>&nbsp;
<a href="#" id="nuevo">Nuevo</a>&nbsp;
<a href="#" id="actualizar">Actualizar</a>&nbsp;
<a href="#" id="eliminar">Eliminar</a>

<div id="info"></div>

</body>
</html>
```

El siguiente paso consiste en definir las llamadas al servicio RESTful y volcar los resultados por pantalla, en concreto en el div “info”. Vamos paso por paso:

Para recuperar todas las rutas, bastaría con algo como esto:

```
$.ajax({
  url: 'http://localhost:8080/misrutas/rutas',
  method: 'GET',
  dataType: 'json',
  success: function(data, text, xhr) {
    $('#info').html('Success: ' + text + '<br>');
    for (var i = 0; i < data.length; i++)
      $('#info').append('<p>' + JSON.stringify(data[i]) + '</p>');
  },
  error: function(xhr, text, error) {
    $('#info').html('Error ' + text + '<br>' + error);
  }
});
```

Para recuperar una ruta en concreto (buscar):

```
$.ajax({
  url: 'http://localhost:8080/misrutas/rutas/' + $('#txtId').val(),
  method: 'GET',
  dataType: 'json',
  success: function(data, text, xhr) {
    $('#info').html('Success: ' + text + '<br>');
    $('#info').append('<p>' + JSON.stringify(data) + '</p>');
  },
  error: function(xhr, text, error) {
    $('#info').html('Error ' + text + '<br>' + error);
  }
});
```

Para crear nuevas rutas:

```
$.ajax({
  url: 'http://localhost:8080/misrutas/rutas',
  method: 'POST',
  dataType: 'json',
  data: JSON.stringify({titulo: $('#txtTitulo').val()}),
  processData: false,
  contentType: 'application/json; charset=utf-8',
  success: function(data, text, xhr) {
    $('#info').html('Success: ' + text + '<br>');
    $('#info').append('<p>' + JSON.stringify(data) + '</p>');
  },
  error: function(xhr, text, error) {
    $('#info').html('Error ' + text + '<br>' + error);
  }
});
```

Para actualizar rutas:

```
$.ajax({
  url: 'http://localhost:8080/misrutas/rutas/' + $('#txtId').val(),
  method: 'PUT',
  dataType: 'json',
  data: JSON.stringify({id: $('#txtId').val(), titulo: $('#txtTitulo').val()}),
  processData: false,
  contentType: 'application/json; charset=utf-8',
  success: function(data, text, xhr) {
    $('#info').html('Success: ' + text + '<br>');
    $('#info').append('<p>' + JSON.stringify(data) + '</p>');
  },
  error: function(xhr, text, error) {
    $('#info').html('Error ' + text + '<br>' + error);
  }
});
```

Por último, para eliminar rutas:

```
$.ajax({
  url: 'http://localhost:8080/misrutas/rutas/' + $('#txtId').val(),
  method: 'DELETE',
  success: function(data, text, xhr) {
    $('#info').html('Success: ' + text + '<br>');
  },
  error: function(xhr, text, error) {
    $('#info').html('Error ' + text + '<br>' + error);
  }
});
```

Una vez implementada esta webapp tan sencilla, procedemos a utilizarla. Para ello, recordemos que ha sido implementada en el fichero /static/test.html y por tanto puede ser accedida a través del servidor Express que hemos definido en los apartados anteriores. Por lo tanto, si no lo tenemos arrancado, ejecutamos el fichero app.js (en el directorio raíz) y cargamos desde un navegador la URL:

`http://localhost:8080/test.html`

Debería de aparecer la webapp y deberíamos de poder gestionar nuestras rutas a través de ésta.

## 6.2 CORS

El ejemplo en el apartado anterior ha funcionado correctamente porque la webapp y el servicio RESTful que ésta consume están incluidos en el mismo dominio. En ocasiones éste no será el caso. Por ejemplo, si instalamos nuestra webapp en un dispositivo móvil, haciendo uso de PhoneGap, o bien cargamos nuestra webapp desde el sistema de ficheros local. En este segundo caso, el navegador activará el mecanismo de seguridad Same-Origin Policy y bloqueará cualquier petición AJAX a un recurso que no pertenece al mismo dominio que el script que efectúa la petición.

Podemos comprobar esta situación fácilmente con nuestra webapp de ejemplo. Para ello, en el navegador cargaremos el fichero test.html desde el sistema de ficheros (generalmente Ctrl + O para abrir el explorador de ficheros). Ahora intentaremos efectuar cualquier petición al servicio y siempre obtendremos un error. Para observar el tipo de error basta con abrir las herramientas para desarrolladores (F12) y observaremos un error similar al siguiente:

`XMLHttpRequest cannot load http://localhost:8080/misrutas/rutas. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.`

Para “relajar” esta restricción, navegador y servicio deben de colaborar, implementando CORS (Cross Origin Resource Sharing). Con CORS el servidor indica qué dominios origen están autorizados para acceder a un determinado recurso, y con qué métodos HTTP. CORS está basado en el intercambio de cabeceras HTTP.

Para habilitar CORS en nuestro servicio deberemos modificar ligeramente el fichero app.js (en la raíz del proyecto), de tal forma que tras cada petición HTTP, enviaremos las cabeceras HTTP necesarias para autorizar acceso a nuestro servicio RESTful desde cualquier origen. Para ello, instalaremos un middleware que se activará siempre, y establecerá dichas cabeceras. Si la petición HTTP es de tipo OPTIONS, la petición finaliza, en otro caso se “pasa la patata caliente” al siguiente middleware/ruta en la cadena.

```
app.use(function(req, res, next) {
  res.header('Access-Control-Allow-Origin', "*");
  res.header('Access-Control-Allow-Methods', 'OPTIONS,GET,PUT,POST,DELETE');
  res.header('Access-Control-Allow-Headers', 'Content-Type');

  if (req.method == 'OPTIONS') {
```

```

    res.status(200).send();
  } else {
    next();
  }
});

```

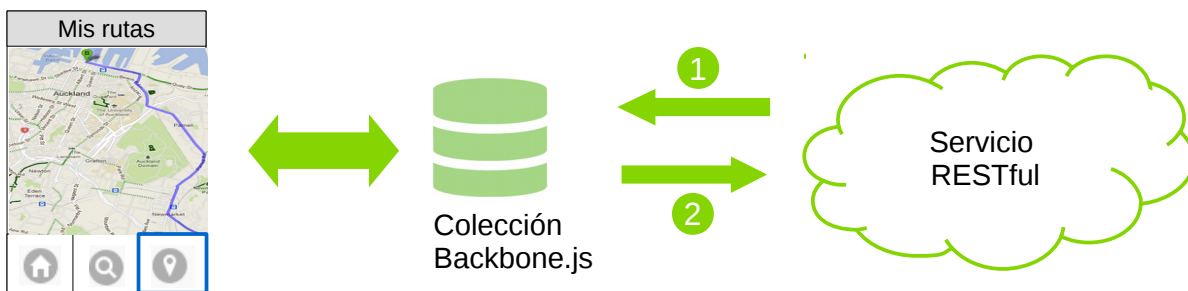
Si reproducimos el mismo ejemplo del comienzo de la sección debería funcionar sin problemas. Recordar parar (si estaba arrancado) y volver a arrancar el servicio app.js previamente.

## 6.3 Backbone.js

Una vez tenemos la garantía de que nuestro servicio funciona bien, e implementa CORS, es el momento de integrarlo con nuestra fantástica webapp Mis Rutas implementada en el laboratorio anterior, que se supone ya está disponible en el proyecto actual, bajo la carpeta /static.

Si recordamos, dicha webapp utilizaba el framework Backbone.js, pero todas las rutas se guardaban en memoria, en el interior de una colección Backbone.js. Lo que deseamos en esta ocasión es que dichas rutas se obtengan de un servicio RESTful, y se mantengan sincronizadas de manera permanente. Afortunadamente, Backbone.js incorpora un completo soporte para implementar esta sincronización de datos.

La estrategia se presenta a continuación de manera gráfica:



Con esta aproximación el modelo de datos de todas las vistas de nuestra aplicación seguirá siendo la colección de rutas. Sin embargo, esta colección de rutas estará en permanente sincronía con el servicio RESTful. Para ello, en el gráfico identificamos dos momentos fundamentales de sincronización:

1. Cuando la webapp se carga por primera vez, es necesario inicializar la colección de rutas obteniéndolas todas desde el servicio RESTful.
2. Tras cada actualización (incluyendo creación y eliminación) de una ruta, es necesario notificar dicho cambio al servicio RESTful.

Como hemos visto en teoría, podemos vincular fácilmente un modelo o colección Backbone.js a un servicio RESTful, estableciendo su atributo 'urlRoot' o 'url' respectivamente. Una vez el modelo/colección queda vinculado, podemos recuperar la información desde el servicio invocando el método `.fetch()`. Además, en el caso de un modelo, podemos crearlo/guardarlo utilizando `.save()` y destruirlo utilizando `.destroy()`. Backbone.js se encarga de traducir todas estas operaciones a las llamadas oportunas sobre el servicio RESTful.

Para implementar la sincronización marcada como (1) en el gráfico anterior, es necesario que cuando se cargue la webapp se recuperen automáticamente todas las rutas desde el servicio. Esta operación puede efectuarse fácilmente en el constructor de la colección Rutas.

Para implementar la sincronización marcada como (2) en el gráfico anterior, debemos capturar cualquier modificación sobre las rutas (eventos 'add', 'remove', 'change') y reenviarla al servicio

RESTful, para que tenga efecto tanto local como remotamente. En el caso del evento 'add', cuando una nueva ruta es añadida a la colección, deberemos crearla en el servicio utilizando .save() sobre la nueva ruta. En el caso de 'remove', cuando una ruta sea eliminada de la colección, deberemos destruirla invocando .destroy() sobre la ruta eliminada. Al método destroy() le pasaremos la opción silent: true, para que no se fuerce un nuevo evento "remove" sobre la colección a la que pertenece. En el caso de 'change', cuando un atributo de la ruta ha cambiado, debemos actualizar su estado en el servicio, invocando para ello .save() sobre la ruta modificada.

A continuación presentamos una primera versión de los ficheros Ruta.js y Rutas.js que implementan esta estrategia:

```
var Ruta = Backbone.Model.extend({
  urlRoot: 'http://localhost:8080/misrutas/rutas',
  initialize: function() {
    if (!this.has("posiciones")) this.set('posiciones', []);
    if (!this.has("fecha")) this.set('fecha', Date());
  },
  defaults: {
    titulo: 'Undefined',
    visible: 'on',
    color: '#000000',
  },
});

var Rutas = Backbone.Collection.extend({
  url: 'http://localhost:8080/misrutas/rutas',
  model: Ruta,
  initialize: function() {
    this.on("add", function(model, col, opt) {
      console.log('Rutas:add ' + model.id);
      model.save();
    });
    this.on("remove", function(model, col, opt) {
      console.log('Rutas:remove ' + model.id);
      model.destroy({silent:true});
    });
    this.on("change", function(model, opt) {
      console.log('Rutas:change ' + model.id);
      model.save();
    });
    this.fetch();
  }
});
```

Se puede observar que ahora los identificadores de las rutas no los genera la webapp, sino que los asignará el servicio RESTful.

Con esta modificación bastaría para conectar nuestra webapp con el servicio RESTful diseñado en esta práctica.

No obstante, observamos cierto comportamiento anómalo cuando la webapp se carga por primera vez y existen rutas previas en el servicio RESTful. Podemos observar que se efectúa una petición HTTP GET al servicio, pero después se efectúa una petición PUT por cada ruta recuperada, modificándola con los mismos datos originales. Este comportamiento anómalo no afecta al resultado final, pero no tiene sentido que al recuperar todas las rutas del servicio RESTful después se efectúe una actualización de cada una de ellas con los mismos datos. Esto sucede porque cuando recuperamos todas las rutas desde el servicio invocando .fetch(), por cada ruta recuperada, Backbone.js emite el evento 'add' sobre la colección, y nosotros hemos indicado que cuando suceda este evento se guarde la ruta en el servicio. Nuestra intención era crear nuevas rutas en el servicio, pero ahora observamos que este evento también sucede cuando se cargan todas las rutas por primera vez.

Para que la aplicación sea correcta debemos de evitar estas comunicaciones innecesarias. Una manera de hacerlo es evitando que se dispare el evento 'add' en esta primera sincronización. Esto se puede conseguir indicando en el método .fetch() la opción { reset: true }. Con esta opción indicamos a Backbone.js que la recuperación de rutas no se haga de manera inteligente, utilizando el método .set() de la colección, y por tanto disparando eventos 'add', sino que se haga utilizando el método .reset() de la colección, que no dispara eventos 'add', sino únicamente el evento 'reset'. Con este “truco” evitaremos el comportamiento anómalo descrito anteriormente.

Con esta mejora, sin embargo, la webapp no es todavía del todo correcta. Existe otro error, que tiene que ver con la generación de identificadores. Es necesario recordar que cuando se crea una ruta en la webapp, ésta no dispone de identificador único .id. Backbone.js le asigna un identificador temporal en el atributo .cid. La ruta recibe su identificador cuando se guarda en el servicio. Existe por tanto un intervalo de tiempo en el que la tarea creada en la webapp no posee identificador .id. Si en ese intervalo de tiempo alguna de nuestras vistas intentar acceder a dicho atributo .id el comportamiento obtenido podría no ser el esperado. Y efectivamente, esto sucede en la vista ListaRutas. Consideremos la siguiente secuencia de eventos:

1. El usuario crea una nueva ruta y se añade a la colección de rutas.
2. Se genera el evento 'add' sobre la colección.
3. La colección recibe el evento 'add' y efectúa una llamada AJAX HTTP POST para crear la nueva ruta en el servicio, pero la llamada es asíncrona, y el resultado con el id de la nueva ruta tarda en llegar.
4. La vista ListaRutas recibe el evento, ya que se registró para recibir este tipo de eventos.
5. La vista ListaRutas se redibuja, reconstruyendo el listview, creando un nuevo item por cada ruta de la colección, incluyendo la nueva ruta. Cuando se crea el nuevo item se genera un link <a> con atributo 'id' establecido al atributo .id de la ruta, que para la nueva ruta es undefined.
6. La colección recibe el resultado de la llamada AJAX efectuada en (3).

Con esta secuencia el listview generado en (5) sería erróneo, y el usuario no podría editar la ruta recién creada.

Existen varias maneras de solucionar este problema. Una de ellas es hacer cambios en las vistas únicamente cuando estos han tenido lugar en el servicio y ya se ha obtenido la respuesta. Para implementar esta estrategia deberíamos de suscribir las vistas ListaRutas y Mapa únicamente al evento 'sync' que genera la colección Backbone.js cuando el proceso de sincronización con el servicio RESTful haya finalizado con éxito. Desgraciadamente, esta aproximación serviría únicamente para creaciones y modificaciones, pero no para eliminaciones. De este modo, en el fichero ListaRutas.js, deberíamos reemplazar este código:

```
this.collection.on('add', function() { self.render(); });
this.collection.on('remove', function() { self.render(); });
this.collection.on('change:titulo', function() { self.render(); });
```

Por este otro:

```
this.collection.on('remove', function() { self.render(); });
this.collection.on('sync', function() { self.render(); });
```

De igual modo, en el fichero Mapa.js, deberíamos reemplazar este código:

```
this.collection.on('change:color', function() { self.render(); });
this.collection.on('change:visible', function() { self.render(); });
this.collection.on('add', function() { self.render(); });
this.collection.on('remove', function() { self.render(); });
```

Por este otro:

```
this.collection.on('remove', function() { self.render(); });
```



```
this.collection.on('sync', function() { self.render(); });
```

Con estos cambios solucionamos el problema planteado anteriormente. Aún queda otra cuestión por resolver. Se puede comprobar que al crear una ruta, se producen los siguientes sucesos:

1. El usuario crea una nueva ruta y se añade a la colección de rutas.
2. Se genera el evento 'add' sobre la colección.
3. La colección recibe el evento 'add' y efectúa una llamada AJAX HTTP POST para crear la nueva ruta en el servicio.
4. Cuando se recibe el resultado de la llamada asíncrona anterior, se recibe de nuevo la ruta creada, pero con un atributo más, el .id de la ruta. Esto internamente se traduce en un .set() sobre la ruta en cuestión, que a su vez genera los eventos 'change' y 'change:id' sobre el modelo, y sobre la colección en la que se encuentra el modelo.
5. La colección recibe el evento 'change' y efectúa una llamada AJAX HTTP PUT para modificar la ruta, transfiriendo la misma información que se acaba de recibir sobre la ruta.

Con esta secuencia de eventos se generan dos peticiones HTTP (un HTTP POST seguido de un HTTP PUT) tras cada creación de ruta, y esto es algo innecesario. Una manera de evitar esta situación consiste en identificar cualquier modificación del atributo .id y evitar su propagación al servicio. Parece una solución segura, ya que nunca permitiremos que el usuario modifique el id de sus rutas. Para implementar esta solución bastaría con utilizar el siguiente código al registrar el manejador del evento 'change' en la colección:

```
this.on("change", function(model, opt) {  
  console.log('Rutas:change ' + model.id);  
  if (model.changedAttributes().id) return;  
  model.save();  
});
```

Y con todos estos cambios parece que podemos obtener una primera versión bastante correcta de la webapp Mis Rutas conectada con nuestro servicio RESTful. Esta primera versión puede ser optimizada en varios sentidos, ya que se producen refrescos de vistas cuando no son necesarios (siempre refrescamos tras suceder el evento 'sync', independientemente de los atributos modificados).

## 7. Heroku

En este apartado desplegaremos nuestra aplicación Node.js en el PaaS Heroku. Para ello, primero debemos instalar el cliente de Heroku en nuestra plataforma, crear una aplicación y desplegar nuestro código fuente en dicha aplicación. Describimos todos estos pasos en las siguientes secciones.

### 7.1 Instalar cliente Heroku

Antes que nada, debemos darnos de alta en Heroku (es gratis!!):

<https://www.heroku.com/>

Una vez dados de alta, procederemos a instalar la herramienta cliente (Heroku CLI), representada por el comando **heroku**. Además, será necesario instalar GIT (porque Node.js ya está instalado). Los pasos de instalación para cada plataforma están perfectamente documentados en la siguiente URL:

<https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up>

Hay que tener en cuenta que las herramientas instaladas deben de estar disponibles en línea de comandos; esto significa que los directorios que contienen los binarios deben ser añadidos a la variable de entorno PATH. Una vez instalados estos requisitos, abrimos una línea de comandos y

comprobamos que Heroku CLI está disponible y nos autenticamos:

```
$ heroku login
```

## 7.2 Preparar la aplicación

Toda aplicación Node.js en Heroku debe disponer de un fichero principal package.json que la describa. Podemos generar dicho fichero automáticamente ubicándonos en el directorio raíz de nuestra aplicación y ejecutando el comando:

```
$ npm init
```

Si abrimos el fichero generado podremos observar una estructura similar a ésta:

```
{
  "name": "...",
  "version": "1.0.0",
  "description": "...",
  "main": "...",
  ...
  "dependencies": {
    "express": "~3.4.4"
  },
  "devDependencies": {},
  "author": "..."
}
```

Como se puede observar, existen datos de identificación de la aplicación (name, version, author), que queremos modificar convenientemente. Nosotros estamos especialmente interesados en el campo “dependencies”. Lo describimos a continuación.

El campo “dependencies” registra todos los paquetes necesarios para la ejecución de nuestra aplicación Node.js. Si necesitamos otros paquetes además de los automáticamente detectados por npm, será necesario indicarlos convenientemente. Podemos editar el campo manualmente, o bien dejar que npm lo haga por nosotros. Cuando instalamos un nuevo paquete con npm install, es posible utilizar el flag --save. Si hacemos esto, npm mantendrá registro automáticamente de nuestras dependencias en el fichero package.json.

```
$ npm install express --save
```

Si deseamos desinstalar la dependencia bastaría con reproducir el mismo proceso con el comando npm uninstall y definiendo el flag --save de nuevo:

```
$ npm uninstall express --save
```

En nuestro caso, la versión básica de nuestra aplicación únicamente requiere de express y body-parser. Nos ubicamos por tanto en el directorio raíz de la aplicación y emitimos los comandos:

```
$ npm install express --save
$ npm install body-parser --save
```

Si necesitamos otros paquetes, deberemos de proceder de este modo. Por ejemplo, si nuestras rutas se almacenan en SQLite, necesitamos instalar el paquete sqlite3. Lo haremos así:

```
$ npm install sqlite3 --save
```

Veremos que las dependencias se añade automáticamente a nuestro fichero package.json.

```
{
  "name": "...",
  "version": "1.0.0",
  "description": "...",
  "main": "...",
  ...
  "dependencies": {
    "express": "~3.4.4",
    "body-parser": "^1.13.1"
  },
  "devDependencies": {},
  "author": "..."
}
```

El campo “dependencies” es fundamental para Heroku, pues es la manera que tenemos de indicarle cuáles son las dependencias de nuestra aplicación Node.js, ya que antes de desplegar nuestra aplicación en el cloud dichas dependencias deben ser resueltas (los paquetes de los que dependemos deben ser instalados).

En Heroku las aplicaciones son ejecutadas por dynos. Cada dino es de un tipo (web, worker, etc.). Nuestra aplicación será ejecutada por un dino web, que es el único capaz de recibir peticiones HTTP del exterior. Nosotros debemos especificarle al dino web cómo ejecutar nuestra aplicación. Para ello disponemos del fichero Procfile ubicado en el directorio raíz de nuestro proyecto. Aquí listamos el comando a ejecutar por cada tipo de dino. En nuestro caso sería algo como:

```
web: node app.js
```

Si desplegamos nuestra aplicación en Heroku tal cual no funcionará. Tiene que ver con cómo se registra nuestro servidor Express para empezar a recibir peticiones HTTP. En Heroku no podemos escuchar por cualquier puerto de comunicaciones. Tenemos que hacerlo por aquél que Heroku nos especifica. Heroku nos comunica información por medio de variables de entorno. Nosotros estamos interesados en la variable de entorno PORT.

En el código de nuestro fichero app.js, el servidor Express se registra de este modo:

```
var express = require('express');
...
app.listen(8080);
```

Debemos cambiar el código de registro, utilizando algo como esto:

```
var express = require('express');
...
var port = process.env.PORT || 8080;
app.listen(port);
```

A continuación debemos efectuar unas modificaciones mínimas en la webapp. En concreto, debemos de acceder a los ficheros del modelo Backbone.js Ruta.js y Rutas.js. Si observamos ambos ficheros, veremos que los objetos de nuestro modelo se sincronizan con un servicio disponible en la URL:

```
http://localhost:8080/misrutas/rutas
```

Esto es correcto si el servicio que consume la webapp está siempre en local, y escuchando por el puerto 8080. Sin embargo, ahora éste no será el caso. Por lo tanto, modificaremos la URL por ésta otra:

```
/misrutas/rutas
```

Quedando el fichero Ruta.js similar a esto:

```
var Ruta = Backbone.Model.extend({
  urlRoot: '/misrutas/rutas',
  ...
})
```

Y el fichero Rutas.js similar a esto otro:

```
var Rutas = Backbone.Collection.extend({
  url: '/misrutas/rutas',
  ...
})
```

Una vez hechas estas modificaciones, crearemos un repositorio GIT local y añadiremos nuestro proyecto. Para ello, nos ubicamos nuevamente en el directorio raíz de la aplicación, y emitimos los siguientes comandos:

```
$ git init
$ git add .
$ git commit -m 'first commit'
```

Ya podemos crear nuestra aplicación y desplegar nuestros fuentes en Heroku. Lo veremos en la próxima sección.

### 7.3 Crear una aplicación

El siguiente paso consiste en crear una aplicación en Heroku. Para ello, nos ubicaremos en directorio raíz de nuestro proyecto y emitiremos el siguiente comando:

```
$ heroku create
Creating app... done, ● nameless-anchorage-76901
https://nameless-anchorage-76901.herokuapp.com/
https://git.heroku.com/nameless-anchorage-76901.git
```

Si no se especifica un nombre para la aplicación, Heroku automáticamente asigna uno disponible generado de manera aleatoria. Una vez creada la aplicación, se lista la URL en la que estará disponible y la URL del repositorio GIT en el que deberemos guardar nuestros fuentes. Además, se crea un remote GIT automáticamente denominado heroku sobre el repositorio almacenado en el directorio actual.

El siguiente paso consiste en subir los fuentes al repositorio GIT en Heroku. Lo podemos hacer con el siguiente comando:

```
$ git push heroku master
```

Para comprobar que la aplicación ha sido correctamente desplegada en el cloud accederemos a la URL listada anteriormente. En nuestro ejemplo sería:

```
https://nameless-anchorage-76901.herokuapp.com/
```

Otra opción es emitir el comando:

```
$ heroku open
```

Que abriría automáticamente el navegador apuntando a la URL anterior.

La aplicación debería estar funcionando sin problemas.

## **8. Extensiones**

A continuación se plantea un conjunto de extensiones, con el objetivo de que el alumno aventajado profundice en algunos conceptos presentados en clase o adquiera soltura con las tecnologías presentadas en el curso. Las extensiones se presentan en los siguientes apartados.

### **8.1 Bases de datos**

El servicio RESTful presentado en los apartados anteriores mantiene la lista de rutas en memoria. Esto significa que si el servicio se para y se vuelve a arrancar toda la información desaparece. Una primera mejora muy evidente consiste en almacenar toda la información sobre rutas en almacenamiento persistente. Para ello, proponemos utilizar SQLite. Sería necesario definir al menos una tabla 'Rutas', y en ella tantas columnas como atributos poseen las rutas de nuestro servicio. El atributo posiciones puede ser fácilmente un string que contenga la representación JSON del vector de posiciones.

### **8.2 Objetos georreferenciados**

En el laboratorio 2 se presentó como extensión la posibilidad de añadir información extra georreferenciada a las rutas (imágenes, sonidos, vídeos, notas, etc.). Si en este laboratorio las rutas se almacenan en un servicio RESTful, es necesario añadir el soporte necesario a la parte del servicio. Obviamente, esta extensión debe trabajar en combinación con la extensión propuesta en el laboratorio 2.

### **8.3 Multiusuario**

Hasta ahora, tanto webapp como servicio han asumido la existencia de un único usuario. Con esta extensión se plantea la posibilidad de añadir soporte multiusuario. Esto tendrá su repercusión tanto en la webapp como en el servicio RESTful.

Por un lado, la webapp debe de implementar gestión de usuarios, incluyendo creación, eliminación, actualización y autenticación. Una vez el usuario ha sido autenticado, webapp y servicio únicamente podrán intercambiar información sobre las rutas que pertenecen al usuario autenticado.

Por otro lado, el servicio deberá ser modificado a varios niveles: por un lado la estructura del almacén persistente debe cambiar, de tal manera que la información se guardará clasificada por usuario; por otro lado, el API del servicio RESTful también debe cambiar, añadiendo nuevas rutas que permitan la autenticación/creación/eliminación/actualización de usuarios y las rutas previas deberán llevar un parámetro nuevo que proporcione información acerca del usuario autenticado.

### **8.4 Open your mind ...**

Cualquier otra mejora será valorada, sólo hay que pararse a pensar un poco ...