



Introducción al desarrollo de web apps

Laboratorio 2.
Mis rutas

Contenido

1. Introducción.....	3
2. La aplicación Mis Rutas.....	3
2.1 La ventana principal.....	3
2.2 La ventana de mis rutas.....	4
2.3 La ventana de editar ruta.....	4
2.4 La ventana de mapa.....	4
3. Estructura de la aplicación.....	5
4. El modelo de datos.....	6
5. La GUI.....	7
5.1 La ventana principal.....	8
5.2 La ventana de mis rutas.....	9
5.3 La ventana de editar ruta.....	10
5.4 La ventana de mapa.....	11
6 La vista.....	13
6.1 Vista Nueva Ruta.....	13
6.2 Vista ListaRutas.....	16
6.3 Vista EditarRuta.....	18
6.4 Vista Mapa.....	19
7. El inicio.....	21
8. Extensiones.....	23
8.1 Aplicación nativa.....	23
8.2 Almacenamiento persistente.....	23
8.3 Optimización del refresco de la GUI.....	23
8.4 Objetos georreferenciados.....	23
8.5 Open your mind	24

1. Introducción

Esta práctica persigue los siguientes objetivos:

- 1.- Comprender y utilizar el toolkit jQuery Mobile para pintar la GUI de una webapp.
- 2.- Comprender y utilizar el toolkit Backbone.js para organizar el código de una webapp.

En esta práctica vamos a diseñar una webapp móvil utilizando un stack de desarrollo concreto: jQuery Mobile + Backbone.js + PhoneGap. Para ello, primero presentaremos los requisitos de la aplicación. Después nos centraremos en el modelo de datos que se utilizará, y lo implementaremos en Backbone.js. Una vez definido el modelo de datos pasaremos a analizar la interfaz gráfica de la webapp, y la implementaremos usando jQuery Mobile. A continuación conectaremos el modelo con la GUI utilizando las vistas Backbone.js correspondientes. Finalmente, se plantearán algunas extensiones que podrá efectuar el alumno aventajado.

2. La aplicación Mis Rutas

El objetivo de esta práctica consiste en diseñar una webapp capaz de registrar rutas en tiempo real, haciendo uso del GPS, y visualizarlas en un mapa. Como se planteará en las extensiones, sobre esta base es posible añadir contenido (fotografías, notas, ...) georreferenciadas, que además se visualicen en el mapa.

El usuario será capaz de:

1. Registrar nuevas rutas. Para ello, se activará un sistema de grabación de nueva ruta. Periódicamente se guardará la posición GPS actual. Todos los datos se registrarán localmente. Una ruta tendrá las siguientes propiedades:
 - Título de la ruta
 - Fecha de creación de la ruta
 - Color que se utilizará para pintar la ruta
 - Posiciones que componen la ruta
2. Administrar rutas. El usuario podrá seleccionar una ruta y modificar algunos de sus datos, o bien eliminarla.
3. Visualizar las rutas en un mapa.

Para implementar esta webapp se diseñará una aplicación multipágina que contemple un mínimo de cuatro secciones:

- La ventana principal
- La ventana de mis rutas
- La ventana de editar ruta
- La ventana de mapa

A continuación presentamos brevemente cada una de ellas.

2.1 La ventana principal

Ventana muy sencilla que permitirá el registro rápido de nuevas rutas. Además, permitirá la navegación sencilla entre las distintas secciones. Cuando el usuario comienza la grabación de una ruta la ventana mostrará información sobre los segundos de grabación activa, y permitirá finalizar la grabación. A continuación se presenta un boceto de esta ventana, antes, durante y tras grabar una ruta:



2.2 La ventana de mis rutas

En esta ventana el usuario encontrará un listado de todas sus rutas. Podrá filtrarlas por título, por ejemplo. Además, el usuario podrá seleccionar una de ellas para poder actualizar alguno de sus datos, eliminarla, etc. A continuación se presenta un boceto de la ventana:

Mis rutas	
	Buscar ...
Ruta Pirineos	>
Ruta Alcoy	>
Ruta Valencia	>

2.3 La ventana de editar ruta

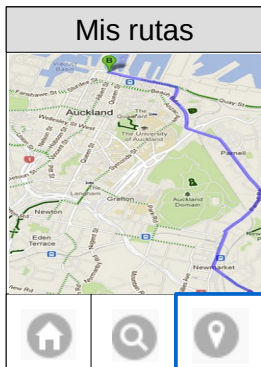
Cuando el usuario selecciona una de las rutas en la ventana de mis rutas, la aplicación abrirá una ventana donde el usuario podrá editar dicha ruta. Las posibilidades de esta ventana dependen del nivel de libertad que se permita efectuar al usuario. Proponemos el siguiente boceto de ventana:

Ruta Pirineos	
El título	
11/03/2015	
<div style="background-color: blue; height: 20px;"></div>	
Visualizar	<input checked="" type="checkbox"/>
Borrar	

Por medio de esta ventana, el usuario podrá actualizar el título de la ruta, el color con el que se pinta en el mapa, así como decidir si se desea visualizar la ruta en el mapa o no.

2.4 La ventana de mapa

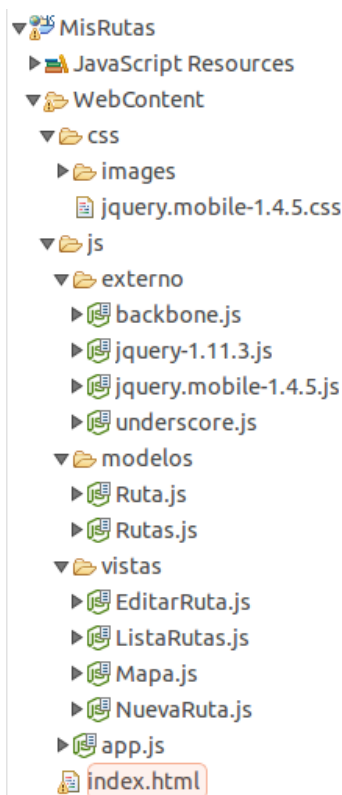
En esta ventana el usuario podrá visualizar todas sus rutas gráficamente.



3. Estructura de la aplicación

Antes de comenzar el diseño es necesario definir la estructura de aplicación que se seguirá. Nosotros seguiremos el patrón sugerido por Backbone.js, que constituye una variante del patrón MV*, como ya hemos visto en las sesiones de teoría. Por tanto, vamos a descomponer nuestra aplicación en una parte estática, compuesta por el fichero principal index.html junto con todos los recursos CSS y JS externos que requiere (jQuery, jQuery Mobile, Backbone.js, Underscore.js). Por otra parte, nuestra webapp poseerá una parte dinámica, encarnada por un conjunto de ficheros JS que implementarán el comportamiento de la aplicación. Esta parte dinámica la descompondremos en un fichero principal app.js, modelos y vistas.

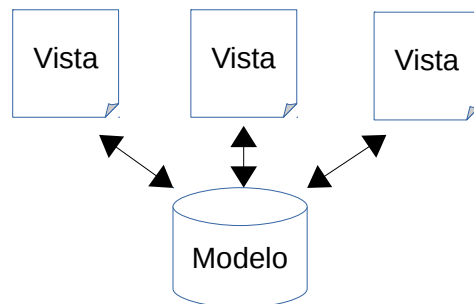
A continuación presentamos una posible estructura de nuestra aplicación:



Como se puede observar todos los recursos CSS se almacenan bajo el directorio /css. Todos los recursos JS se almacenan bajo el directorio /js, y dentro de éste identificamos las dependencias externas, bajo /js/externo, y nuestra aplicación, compuesta por /js/app.js, y los distintos modelos y vistas que iremos implementando poco a poco.

4. El modelo de datos

Comenzaremos el diseño de nuestra aplicación diseñando el modelo de datos. Como ya sabemos, en Backbone.js el modelo de datos representa la base de datos de nuestra aplicación, que es accedida por las distintas vistas.



En nuestro caso, nuestra base de datos está constituida por una colección de rutas. Por tanto, tendremos que definir dos entidades Backbone.js:

- La clase **Ruta**, que extenderá Backbone.Model, y que representará a cada una de las rutas individuales.
- La clase **Rutas**, que extenderá a Backbone.Collection, y que representará nuestra base de datos, es decir, una colección de rutas.

Una buena estrategia consiste en implementar cada modelo en un fichero independiente. Si el número de modelos crece, se favorece el mantenimiento del código. En nuestro caso, con sólo dos clases, podríamos añadir las a un único fichero modelo.js y ubicarlo en el directorio raíz /js. Sin embargo, para implementar un modelo escalable, seguiremos la filosofía de implementar cada clase en un fichero independiente.

A continuación se muestra una posible implementación del fichero /js/modelos/Ruta.js:

```
/**
 * Clase que implementa rutas individuales
 */
var Ruta = Backbone.Model.extend({
  initialize: function() {
    if (!this.id) this.set('id', _.uniqueId());
    if (!this.has("posiciones")) this.set('posiciones', []);
    if (!this.has("fecha")) this.set('fecha', Date());
  },
  defaults: {
    titulo: 'Undefined',
    visible: 'on',
    color: '#000000',
  },
});
```

La única dificultad de esta clase radica en la asignación de identificadores únicos, y si lo deseamos, en la asignación de valores por defecto. Para la generación de identificadores únicos utilizaremos una utilidad de Underscore.js, una dependencia de Backbone.js. Underscore.js está accesible bajo el símbolo '_', y posee un método denominado `_.uniqueId()` que genera identificadores únicos. Esto será válido únicamente si las rutas no se almacenan en almacenamiento persistente. En la primera versión de nuestra webapp así será. Cuando utilicemos almacenamiento persistente será **necesario cambiar esta estrategia**.

Una vez disponemos de la clase principal de nuestro modelo Ruta, es necesario definir una manera de mantener colecciones de rutas. Es el objetivo de la clase Rutas, definida en el fichero /js/modelos/Rutas.js:

```

/**
 * Define una colección de rutas
 */

var Rutas = Backbone.Collection.extend({
  model: Ruta,
  initialize: function() {
    this.on("add", function(model, col, opt) {
      console.log('Rutas:add ' + model.id);
    });
    this.on("remove", function(model, col, opt) {
      console.log('Rutas:remove ' + model.id);
    });
    this.on("change", function(model, opt) {
      console.log('Rutas:change ' + model.id);
    });
  }
});

```

En este caso no hace falta definir nada más que el tipo de los objetos con la opción model. Registramos manejadores para los eventos de manipulación de la colección sólo para depuración, para enterarnos de cómo se va cambiando nuestra base de datos.

Y con esto ya tenemos implementado el modelo de nuestra aplicación, sencillo, ¿no?

5. La GUI

Una vez implementado el modelo llega el momento de definir (1) cómo será la GUI de la aplicación y (2) cómo interactuará el usuario con ella. En esta sección sólo trataremos (1), definiremos la GUI de manera estática utilizando jQuery Mobile, con sus atributos data-xxx. Por su parte, (2) será tratado en la siguiente sección, donde identificaremos las zonas principales de la webapp que constituirán las vistas de la aplicación.

Para definir nuestra webapp crearemos el documento index.html e incluiremos de momento los recursos externos jQuery y jQuery Mobile, que se ha preferido descargar en local por cuestiones de eficiencia. Usaremos un código parecido al siguiente:

```

<!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>
</head>
<body>

</body>
</html>

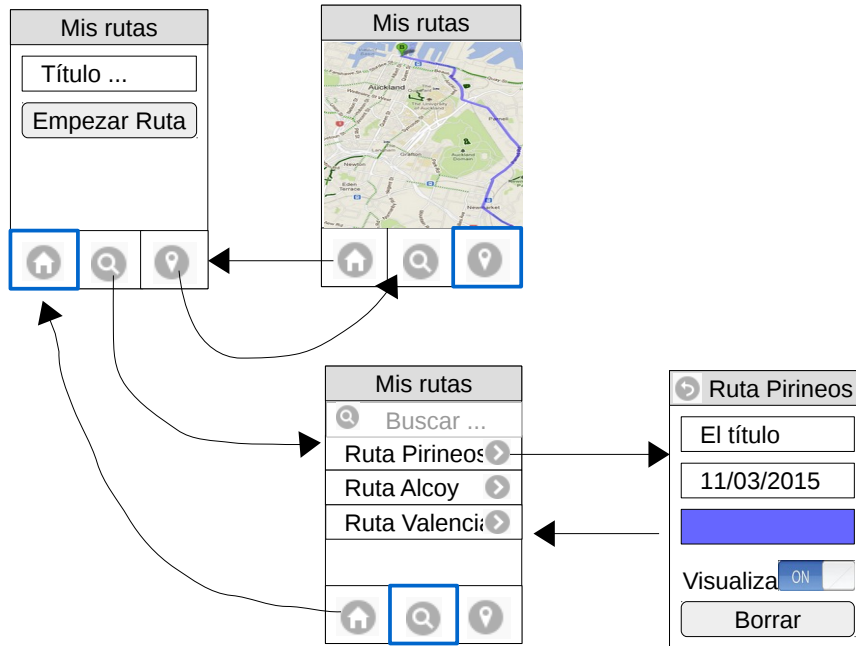
```

Como se ha comentado anteriormente el objetivo consiste en diseñar una webapp multipágina. Definiremos 4 páginas con jQuery Mobile:

- La ventana principal: la denominaremos "pgHome"
- La ventana de mis rutas: la denominaremos "pgMisRutas"
- La ventana de mapa: la denominaremos "pgMapa"
- La ventana de editar ruta: la denominaremos "pgEditarRuta"

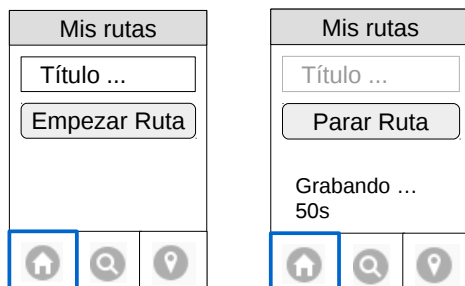
Las tres primeras poseerán una cabecera común, y un pie de página con una barra de navegación que permita navegar entre ellas. La barra de navegación poseerá tres botones, cada uno activado en su correspondiente sección. A continuación se muestran las distintas páginas, así como la

navegación que se debería de habilitar entre ellas:



En las próximas secciones detallamos cómo diseñar las distintas páginas de nuestra aplicación.

5.1 La ventana principal



Además de la cabecera y pie de página con el botón de “home” activado por defecto, en el contenido de esta página ubicaremos una caja de texto para el título de la ruta, un botón, y dos etiquetas de texto que nos permitan escribir la ruta que se está grabando y cuánto tiempo se ha estado grabando. Obtendremos un resultado parecido con el siguiente código:

```
<div id="pgHome" data-role="page">
  <div data-role="header"><h1>Mis rutas</h1></div>
  <div class="ui-content"></div>
  <input id="txtTitulo" type="text" placeholder="Título" >
  <input id="btGrabar" type="button" value="Empezar Ruta" >
  <div id="pnInfo" >
    <p id="lblInfo"> Grabando ruta XXX</p>
    <p id="lblReloj"> 00:00:00 </p>
  </div>
  <div data-role="footer" data-position="fixed">
    <div data-role="navbar">
      <ul>
        <li><a href="#pgHome" class="ui-btn-active ui-state-persist" data-
icon="home">Principal</a></li>
        <li><a href="#pgMisRutas" data-icon="search">Mis rutas</a></li>
        <li><a href="#pgMapa" data-icon="navigation">Mapa</a></li>
      </ul>
    </div>
  </div>
</div>
```



```

</div>
</div>
</div>

```

A continuación se muestra el resultado:

El texto que aparece tras el botón deberá de estar oculto cuando no se grabe, y visible cuando se grabe. Este comportamiento deberá de definirse más adelante, en la vista correspondiente.

5.2 La ventana de mis rutas

En este caso, además de la cabecera y pie de página con el botón de búsqueda activado por defecto, debemos añadir un listview al contenido de la página. Dicho listview debe de poseer posibilidades de filtrado. Conseguiremos este resultado con un código similar al siguiente:

```

<div id="pgMisRutas" data-role="page">
  <div data-role="header"><h1>Mis rutas</h1></div>
  <div class="ui-content"></div>
  <div id="pnRutas">
    <ul data-role="listview" data-filter="true">
      <li><a id="1" href="#">Ruta 1</a></li>
      <li><a id="2" href="#">Ruta 2</a></li>
      <li><a id="3" href="#">Ruta 3</a></li>
      <li><a id="4" href="#">Ruta 4</a></li>
    </ul>
  </div>
</div>
<div data-role="footer" data-position="fixed">
<div data-role="navbar">
  <ul>
    <li><a href="#pgHome" data-icon="home">Principal</a></li>
    <li><a href="#pgMisRutas" class="ui-btn-active ui-state-persist" data-
icon="search">Mis rutas</a></li>
    <li><a href="#pgMapa" data-icon="navigation">Mapa</a></li>
  </ul>

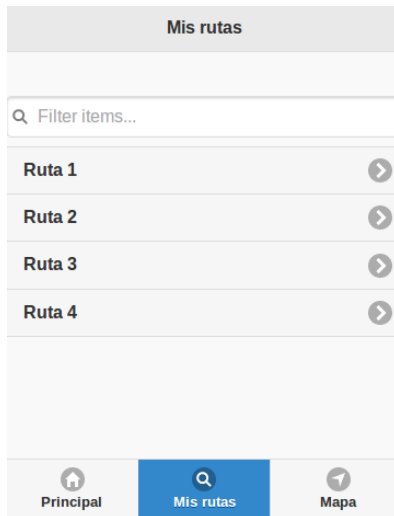
```

```

</div>
</div>
</div>

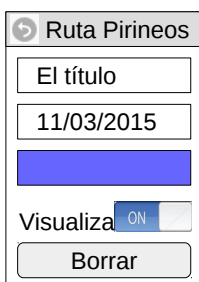
```

Obteniendo el siguiente resultado:



Obviamente, el listado de rutas debe generarse dinámicamente, y además, al presionar sobre una de ellas deberíamos de saltar a la ventana de editar ruta. Este comportamiento será definido posteriormente con la vista correspondiente.

5.3 La ventana de editar ruta



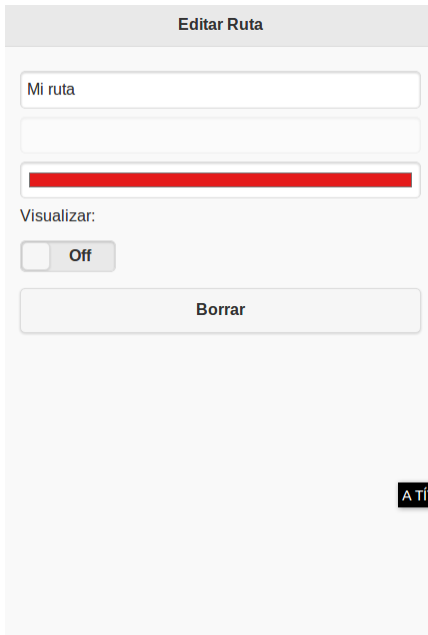
Esta página no tiene acceso directo en la aplicación, sino que será accedida a través de la ventana de mis rutas. Deben aparecer los datos de la ruta seleccionada y permitimos volver atrás. Para ello, sera necesario añadir un botón de atrás en la cabecera. En el contenido de la ventana insertaremos dos cajas de texto, para el título y la fecha de creación de la ruta, un control de tipo "color", para poder seleccionar un color, un interruptor y un botón. A continuación se presenta el código correspondiente:

```

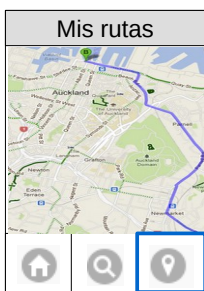
<div id="pgEditorRuta" data-role="page">
  <div data-role="header" data-add-back-btn="true" data-back-btn-
text="Atr&aacute;s"><h1>Editar Ruta</h1></div>
  <div class="ui-content">
    <input id="txtEditorTitulo" type="text" >
    <input id="txtEditorFecha" type="text" disabled >
    <input id="txtEditorColor" type="color" >
    <label for="txtEditorVisualizar">Visualizar:</label>
    <select id="txtEditorVisualizar" data-role="flipswitch">
      <option value="off">Off</option>
      <option value="on">On</option>
    </select>
    <input id="btBorrar" type="button" value="Borrar" >
  </div>
</div>

```

Con el siguiente resultado:



5.4 La ventana de mapa



En este caso, además de la cabecera y pie de página con el botón de mapa activado por defecto, debemos añadir un mapa al contenido de la página. Usaremos para ello Google Maps API v3. Se trata de un API JavaScript muy sencilla de utilizar que puede consultarse a través de la siguiente URL:

<https://developers.google.com/maps/documentation/javascript/?hl=es>

Google Maps API v3 únicamente necesita un contenedor en el que pintar, típicamente un `<div>`. En nuestro caso, definiremos una página muy sencilla con únicamente un contenedor `<div id="pnMapa">`, como se muestra a continuación:

```
<div id="pgMapa" data-role="page">
  <div data-role="header"><h1>Mis rutas</h1></div>
  <div class="ui-content" >
    <div id="pnMapa" ></div>
  </div>
  <div data-role="footer" data-position="fixed">
    <div data-role="navbar">
      <ul>
        <li><a href="#pgHome" data-icon="home">Principal</a></li>
        <li><a href="#pgMisRutas" data-icon="search">Mis rutas</a></li>
        <li><a href="#pgMapa" data-icon="navigation" class="ui-btn-active ui-state-
persist" >Mapa</a></li>
      </ul>
    </div>
  </div>
</div>
```

</div>

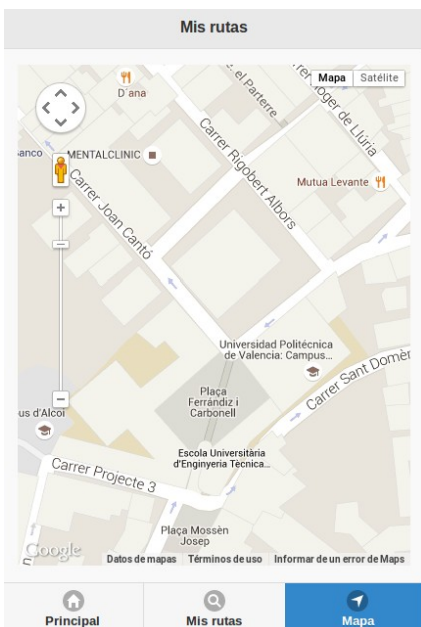
El siguiente paso consiste en añadir el código JS necesario para crear el mapa. Bastaría con el siguiente código, donde fijamos unas coordenadas arbitrarias para centrar el mapa e indicamos dónde (en qué elemento del DOM) debe de pintarse:

```
var myOptions = {
  zoom: 18,
  center: new google.maps.LatLng(38.695015, -0.476049),
  mapTypeId: google.maps.MapTypeId.ROADMAP
};
var map = new google.maps.Map($('#pnMapa')[0], myOptions);
```

Con este código bastaría en cualquier aplicación web normal, pero no en una webapp responsive, ya que Google Maps debe conocer el tamaño del contenedor en el que se pinta, pero en una webapp jQuery Mobile esto no lo sabremos hasta el último momento, y por tanto no se visualizará correctamente el mapa. Es necesario efectuar un pequeño “truco”: antes de mostrar la página, en el evento ‘pageshow’, obtendremos dinámicamente la altura del contenido de la página “pgMapa” y se la fijaremos directamente al panel “pnMapa”. Posteriormente crearemos el mapa en dicho panel. Para más información sobre este problema y cómo solucionarlo se recomienda visitar el siguiente link: <http://www.gajotres.net/using-google-maps-javascript-api-v3-with-jquery-mobile/>

```
$(document).one('pageshow', '#pgMapa', function(e, data) {
  // obtener altura del contenedor
  var header = $("#pgMapa").find("div[data-role='header']:visible");
  var footer = $("#pgMapa").find("div[data-role='footer']:visible");
  var content = $("#pgMapa").find("div.ui-content:visible");
  var viewport_height = $(window).height();
  var content_height = viewport_height - header.outerHeight() - footer.outerHeight();
  if((content.outerHeight() - header.outerHeight() - footer.outerHeight()) <=
viewport_height) {
    content_height -= (content.outerHeight() - content.height());
  }
  $('#pnMapa').height(content_height);
  // crear mapa
  var myOptions = {
    zoom: 18,
    center: new google.maps.LatLng(38.695015, -0.476049),
    mapTypeId: google.maps.MapTypeId.ROADMAP
  };
  var map = new google.maps.Map($('#pnMapa')[0], myOptions);
});
```

Con este pequeño truco el mapa se visualizará correctamente, obteniendo un resultado parecido al siguiente:



Faltaría pintar las rutas de la base de datos. Estudiaremos cómo hacerlo en la vista correspondiente.

6 La vista

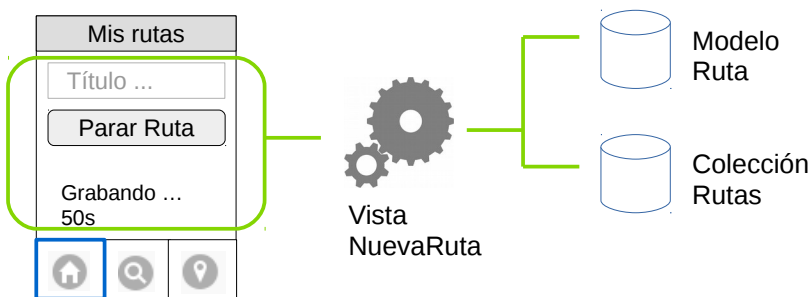
Una vez definida la GUI, debemos darle vida, creando vistas que la conecten con el modelo. Para ello, definiremos 4 vistas, cada una controlando los aspectos fundamentales de cada una de las 4 páginas de nuestra aplicación:

- Vista NuevaRuta: controlará la creación de nuevas rutas, que tiene lugar en la ventana principal.
- Vista ListaRutas: controlará el listado de las rutas de nuestra aplicación, que tiene lugar en la ventana de mis rutas.
- Vista EditarRuta: controlará la edición/borrado de las rutas de nuestra aplicación, que tiene lugar en la ventana de editar ruta.
- Vista Mapa: controlará el pintado de las rutas de nuestra aplicación en el mapa, algo que tiene lugar en la ventana de mapa.

Detallamos estas vistas en las siguientes secciones:

6.1 Vista Nueva Ruta

Esta vista implementa el mecanismo de creación de nuevas rutas. Para ello, envolverá el modelo que representa la nueva ruta, así como la colección de rutas, a la que añadirá la nueva ruta cuando esté lista.



Esta vista debe habilitar el siguiente comportamiento:

1. Si no hay grabación en curso, cuando el usuario haga click en el botón, la vista comenzará la grabación de la ruta, efectuando las siguientes acciones:
 - creará un nuevo modelo Ruta con el título especificado
 - comenzará a registrar el tiempo de grabación
 - comenzará a registrar posiciones GPS de manera periódica
 - refrescará la GUI, cambiando el texto del botón a "Parar", mostrando el texto "Grabando <Nombre Ruta>", y el tiempo que ha pasado desde que comenzó el proceso
2. Si hay una grabación en curso, cuando el usuario haga click en el botón, la vista finalizará la grabación de la ruta, efectuando las siguientes acciones:
 - finalizará el registro del tiempo de grabación
 - finalizará el registro periódico de posiciones GPS
 - añadirá el modelo Ruta a la colección Rutas
 - refrescará la GUI, cambiando el texto del botón a "Empezar ruta" y ocultando toda la información sobre el estado actual de grabación de la ruta

Comenzamos la definición de la vista definiendo en un nuevo fichero `/js/vistas/NuevaRuta.js` la clase `NuevaRuta`, extendiendo para ello `Backbone.View`:

```
var NuevaRuta = Backbone.View.extend({ ... });
```

En el constructor de esta vista inicializaremos todos los atributos que necesitaremos para implementar el comportamiento descrito anteriormente. Por un lado, para saber si estamos en (1) o (2) utilizaremos un atributo `.grabando`, que nos indicará si estamos grabando o no. Cuando estamos grabando, para habilitar refresco de reloj y recuperación de posiciones periódicas necesitaremos sendos timers `.timerReloj` y `.timerGps`. También deberemos recordar el número de segundos que han pasado desde que comenzó el grabado de la ruta. Por último, al crear la vista la pintaremos, invocando su método `.render()`. A continuación presentamos el código de inicialización asociado:

```
initialize: function() {  
  this.grabando = false;  
  this.timerReloj = null;  
  this.timerGps = null;  
  this.contadorReloj = null;  
  this.render();  
}
```

Una vez inicializada, el siguiente paso consiste en implementar el comportamiento descrito en (1) y (2). Lo haremos definiendo los métodos `.empezarRuta()` y `.pararRuta()` de la vista:

```
empezarRuta: function() {  
  this.model = new Ruta();  
  
  console.log('empezarRuta(' + this.model.id + ')');  
  
  // recuperar título de la ruta  
  var titulo = this.$('#txtTitulo').val();  
  if (!titulo || titulo.length == 0) titulo = "Default";  
  this.model.set("titulo", titulo);  
  
  // comenzar grabación de ruta (timers, ...)  
  this.grabando = true;  
  this.contadorReloj = 0;  
  var self = this;  
  this.timerReloj = setInterval(function() {  
    self.contadorReloj++;  
    self.render();  
  }, 1000);  
  this.timerGps = setInterval(function() {  
    self.leerGps();  
  }, 1000);  
  // pintar vista  
  this.render();  
}  
  
pararRuta: function() {  
  console.log('pararRuta(' + this.model.id + ')');  
  
  this.grabando = false;  
  
  // parar timers  
  clearInterval(this.timerReloj);  
  clearInterval(this.timerGps);  
  
  // guardar ruta  
  this.collection.add(this.model);  
  // limpiar modelo  
  this.model = null;  
  // pintar vista  
  this.render();  
}
```

Como se puede observar, para recoger las lecturas del GPS se invoca cada segundo el método `.leerGps()` de la vista. Este período resultaría demasiado corto para una aplicación real, por lo que sería necesario incrementarlo. Suministramos una posible implementación de este método, donde se simula la adquisición de la posición GPS (en caso de no disponer de dispositivo GPS), y finalmente se almacena en el modelo.:

```
leerGps: function() {

    // init
    window.inc = typeof(inc) == 'undefined'? 0.00005: window.inc;
    window.lat = typeof(lat) == 'undefined'? 38.695015: window.lat;
    window.lng = typeof(lng) == 'undefined'? -0.476049: window.lng;
    window.dir = typeof(dir) == 'undefined'? Math.floor((Math.random()*4)): window.dir; //
    numbers 0,1,2,3 (0 up, 1 right, 2, down, 3 left)

    // generate direction (randomly)
    // it is more likely to follow the previous direction
    var nuevaDir = Math.floor((Math.random()*4)); // number 0,1,2,3

    if (nuevaDir != (dir + 2) % 4) dir = nuevaDir;

    switch (dir) {
    case 0: // up
        lat += inc;
        break;
    case 1: // right
        lng += inc;
        break;
    case 2: // down
        lat -= inc;
        break;
    case 3: // left
        lng -= inc;
        break;
    default:
    }

    var pos = {lat: lat, lng: lng};

    // add new position to the route
    var posiciones = this.model.get('posiciones');
    posiciones.push(pos);
    this.model.set('posiciones', posiciones);
}
```

Tanto `.empezarRuta()` como `.pararRuta()` refrescan la GUI de la vista invocando `.render()`. Esta función se encarga de averiguar cuál es el estado actual de la vista (si está grabando o no) y refresca los distintos controles gráficos en consecuencia:

```
render: function() {
    if (this.grabando) {
        this.$('#txtTitulo').val(this.model.get('titulo')).textinput('refresh');
        // show panel - change button text
        this.$('#btGrabar').val('Parar').button('refresh');
        this.$('#lblInfo').text('Guardando ruta ' + this.model.get('titulo') + ' ...');
        var minutos = parseInt(this.contadorReloj/60);
        var segundos = this.contadorReloj%60;
        var horas = parseInt(minutos/60);
        minutos = minutos%60;
        this.$('#lblReloj').text("" + (horas<10? "0": "") + horas + ":" +
            (minutos<10? "0": "") + minutos + ":" + (segundos < 10? "0": "") + segundos);
        this.$('#pnInfo').css('visibility', 'visible');
    } else {
        // hide panel - change button text
        this.$('#btGrabar').val('Empezar ruta').button('refresh');
        this.$('#pnInfo').css('visibility', 'hidden');
    }
}
```

```

    }
  }
}

```

Por último, todo el comportamiento descrito sucede cuando el usuario presiona el botón 'btGrabar' de la vista. Se trata de la única interacción que tiene lugar por parte del usuario. Lo definimos en el mapa de eventos de la vista.

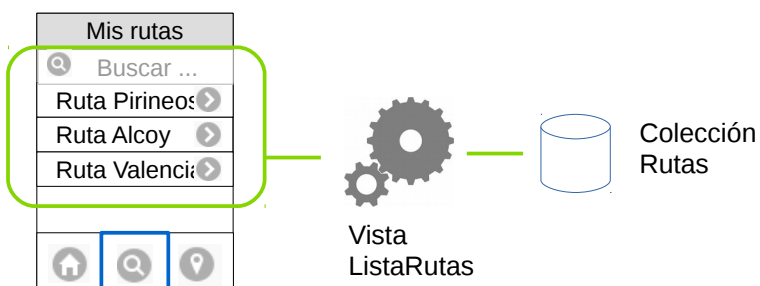
```

events: {
  'click #btGrabar': function() {
    if (this.grabando) this.pararRuta();
    else this.empezarRuta();
  }
}

```

6.2 Vista ListaRutas

Esta vista se encargará de mostrar todas las rutas registradas en la base de datos de la aplicación. Para ello, estará en permanente conexión con la colección de rutas, y refrescará en la GUI cualquier cambio/adición/eliminación que se produzca sobre el conjunto de rutas. Gráficamente:



Para implementar esta vista, primero definiremos la nueva clase ListaRutas en el fichero `/js/vistas/ListaRutas.js`:

```
var ListaRutas = Backbone.View.extend({ ... });
```

Esta vista será la responsable de pintar el listado de rutas de nuestra aplicación. Por tanto, modificaremos la página "pgMisRutas", eliminando el listview que existía de manera estática.

```

<div id="pgMisRutas" data-role="page">
  <div data-role="header"><h1>Mis rutas</h1></div>
  <div class="ui-content"></div>
  <div id="pnRutas">
  </div>
  <div data-role="footer" data-position="fixed">
  <div data-role="navbar">
    <ul>
      <li><a href="#pgHome" data-icon="home">Principal</a></li>
      <li><a href="#pgMisRutas" class="ui-btn-active ui-state-persist" data-
icon="search">Mis rutas</a></li>
      <li><a href="#pgMapa" data-icon="navigation">Mapa</a></li>
    </ul>
  </div>
  </div>
</div>

```

Cuando creamos la vista la vincularemos con el panel "pnRutas" y con la colección de rutas. La vista se encargará de crear y refrescar automáticamente la colección de rutas en dicho panel.

La vista debe estar atenta a cualquier cambio del modelo que afecte al listado de rutas. Cualquier adición y eliminación modifica el listado de rutas. Sin embargo, si pensamos en ello, el cambio del título de una ruta también podría afectar al listado de rutas. Registraremos manejadores de todos

estos eventos en el constructor de la vista:

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

En nuestro caso, refrescar la vista significa pintar el listado de rutas, y lo haremos “a lo bruto”, eliminando el listado anterior y creándolo de nuevo. Esta estrategia no es eficiente, pero es simple y efectiva. A continuación mostramos el código:

```
render: function() {  
    // limpiar  
    this.$el.html('<ul data-role="listview" data-filter="true"></ul>');  
    // pintar rutas  
    for(var i = 0; i < this.collection.size(); i++){  
        var m = this.collection.at(i);  
        var str = '<li><a id="' + m.id + '" href="#">' + m.get('titulo') + '</a></li>'  
        this.$el.find('[data-role="listview"]').append(str);  
    }  
    this.$el.find('[data-role="listview"]').listview();  
}
```

Por último, cuando el usuario presiona sobre una ruta deberíamos de dirigirle a la página de editar ruta. Se trata de la única interacción que debemos controlar en nuestro listado. La definimos en el diccionario de eventos:

```
events: {  
    'click a' : 'abrirEditarRuta'  
}
```

El método “abrirEditarRuta” se encargará de:

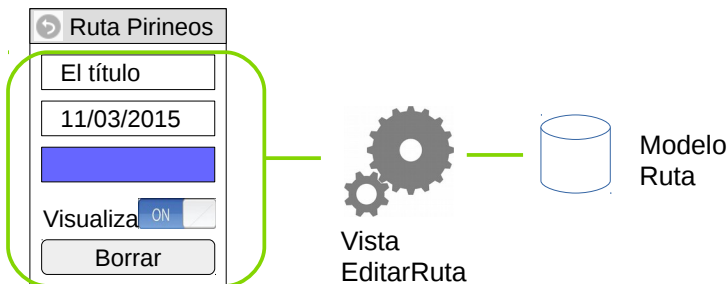
1. Identificar la ruta que se ha seleccionado.
2. Conectar a la vista EditarRuta la ruta que controlará, y refrescar la GUI para que muestre la información de la nueva ruta.
3. Transitar a la página 'pgEditarRuta'.

Sin embargo, para poder comunicarse con la vista EditarRuta, ésta debe de haber sido previamente definida. Publicamos para ello en la vista el método .editarRuta(), que acepta la vista EditarRuta con la que se vinculará.

```
editarRuta: function(vistaEditarRuta) {  
    this.vistaEditarRuta = vistaEditarRuta;  
}  
  
abrirEditarRuta: function(e) {  
    // recuperar id  
    var id = $(e.target).attr('id');  
    console.log('abrirEditarRuta (' + id + ')');  
    this.vistaEditarRuta.model = this.collection.get(id);  
    $(':mobile-pagecontainer').pagecontainer('change', '#pgEditarRuta');  
    this.vistaEditarRuta.render();  
}
```

6.3 Vista EditarRuta

Esta vista se encargará de mostrar todos los datos de una ruta (excepto la lista de posiciones) y permitirá actualizar algunos de ellos, así como eliminar la ruta de la aplicación. Por tanto, debe estar vinculada a una ruta. Gráficamente:



Implementamos esta vista en el fichero `/js/vistas/EditarRuta.js`:

```
var EditarRuta = Backbone.View.extend({ ... })
```

Se trata de una vista muy sencilla que únicamente debe mantener sincronizada la ruta con los datos que aparecen en la GUI. Por tanto, por un lado `.render()` se encargará de pintar los datos de la ruta en los distintos controles de la GUI:

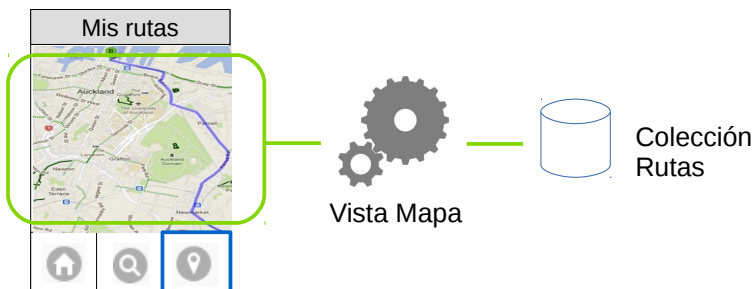
```
render: function() {
  this.$('#txtEditarTitulo').val(this.model.get('titulo')).textinput('refresh');
  this.$('#txtEditarFecha').val(this.model.get('fecha')).textinput('refresh');
  this.$('#txtEditarColor').val(this.model.get('color')).textinput('refresh');
  this.$('#txtEditarVisualizar').val(this.model.get('visible')).flipswitch('refresh');
}
```

Por otro lado, la vista debe detectar cualquier actualización en los controles de la GUI y modificar el modelo convenientemente. Para ello, registraremos manejadores del evento 'change' para los distintos controles actualizables. Por último, también es necesario controlar el 'click' del botón de borrado. Toda esta interacción del usuario con la vista la definimos en el correspondiente mapa `events`:

```
events: {
  'change #txtEditarTitulo': function() {
    this.model.set('titulo', this.$('#txtEditarTitulo').val());
  },
  'change #txtEditarColor': function() {
    this.model.set('color', this.$('#txtEditarColor').val());
  },
  'change #txtEditarVisualizar': function() {
    this.model.set('visible', this.$('#txtEditarVisualizar').val());
  },
  'click #btBorrar': function() {
    this.collection.remove(this.model);
    $(':mobile-pagecontainer').pagecontainer('change', '#pgMisRutas');
  }
}
```

6.4 Vista Mapa

Esta vista se encargará de mostrar el mapa, así como de pintar sobre el mismo todas las rutas de la base de datos que se encuentren en estado visible. Por tanto, deberá estar conectada con la colección de rutas, y detectar cualquier cambio que afecte a su visualización, para refrescar el mapa convenientemente. Gráficamente:



Crearemos la vista Mapa en el fichero `/js/vistas/Mapa.js`:

```
var Mapa = Backbone.View.extend({ ... });
```

La vista mapa se encargará de todo el trabajo de pintado del mapa, interactuando continuamente con Google Maps API v3. Eliminaremos de la página 'pgMapa' todo lo relacionado con el mapa, que incluye todo el código JavaScript y el panel 'pnMapa'. Todo esto lo integraremos en esta vista.

La gestión del mapa incluye dos operaciones fundamentales:

1. La inicialización: cuando la página se muestra por primera vez (y por tanto las dimensiones de los distintos elementos de la página están ya definidos), creamos el contenedor del mapa 'pnMapa', e inmediatamente pintamos el mapa en su interior. Una vez el mapa ha sido creado y pintado por primera vez, recorreremos la colección de rutas y pintamos aquellas rutas que sean visibles con el color correspondiente.
2. El refresco: cuando la colección de rutas es alterada, debemos refrescar las rutas que se han pintado en el mapa. En este caso, las actualizaciones que nos interesa detectar serán adiciones, eliminaciones, y el cambio del color o estado de visibilidad de cualquier ruta. Nuevamente, a la hora de refrescar las rutas en el mapa podemos tomar dos acciones: (i) borrar todas las rutas y volver a pintarlas, o (ii) detectar la/s ruta/s que ha/n sido modificada/s, borrarla/s y volver a pintarla/s. (i) es matar moscas a cañonazos pero simplifica el proceso de renderizado, (ii) es más óptimo pero complica la lógica del refresco. En esta primera versión de la aplicación adoptaremos la estrategia (i).

A continuación se presenta el constructor de la vista, que incluirá el código necesario para inicializar el mapa, así como el registro de los manejadores pertinentes para detectar cambios en el modelo:

```
initialize: function() {
    var self = this;

    // crear el mapa la primera vez que se muestra la página
    $(document).one('pageshow', '#' + this.$el.attr('id'), function(e, data) {

        // crear contenedor
        self.$('.ui-content').append('<div id="pnMapa"></div>');

        // obtener altura del contenedor
        var header = $.mobile.activePage.find("div[data-role='header']:visible");
        var footer = $.mobile.activePage.find("div[data-role='footer']:visible");
        var content = $.mobile.activePage.find("div.ui-content:visible");
        var viewport_height = $(window).height();
        var content_height = viewport_height - header.outerHeight() - footer.outerHeight();
```

```

    if((content.outerHeight() - header.outerHeight() - footer.outerHeight()) <=
viewport_height) {
    content_height -= (content.outerHeight() - content.height());
    }
    self.$('#pnMapa').height(content_height);

    // crear mapa
    var myOptions = {
        zoom: 18,
        center: new google.maps.LatLng(38.695015,-0.476049),
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    self.map = new google.maps.Map(self.$('#pnMapa')[0], myOptions);
    self.polylines = new Array();

    // pintar
    self.render();
});

// controlar cambios en el modelo
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 último, el método `.render()` debe pintar las rutas en el mapa. Para ello, haremos uso de los Polyline de Google Maps API v3, que nos permiten pintar una polilínea sobre un mapa. Para más información remitirse a la documentación oficial de Google Maps API v3. Por tanto, cuando se refresque el mapa, limpiaremos todas las polilíneas pintadas hasta el momento, y volveremos a pintarlas. Para poder limpiar las polilíneas pintadas en el último refresco es necesario mantener información sobre ellas. Lo haremos en el vector `polylines[]`. Los distintos puntos que configuran una polilínea deben ser objetos de tipo `google.maps.LatLng`, pero en nuestras rutas, nosotros disponemos de objetos JavaScript con las propiedades `{lat:xxx, lng:xxx}`, por lo tanto es necesario hacer una transformación; usamos `_.map()` de Underscore.js para efectuar esta transformación. Finalmente, la ruta sólo se pintará si está en estado visible, y se pintará del color establecido. A continuación mostramos el código asociado:

```

render: function() {

    // limpiar todas las rutas pintadas
    for (var i = 0; i < this.polylines.length; i++)
        this.polylines[i].setMap(null);
    this.polylines = [];

    // pintar las rutas
    var self = this;
    this.collection.forEach(function(ruta) {

        // sólo si son visibles
        if (ruta.get('visible') == 'on') {
            var polyline = new google.maps.Polyline({
                path: _.map(ruta.get('posiciones'), function(coords) {
                    return new google.maps.LatLng(coords.lat, coords.lng);
                }),
                map: self.map,
                strokeColor: ruta.get('color'),
                strokeOpacity: 1.0,
            });

```

```

        strokeWeight: 4
    });

    // guardar información sobre las rutas pintadas
    self.polylines.push(polyline);
}
});
}

```

7. El inicio

Una vez creado el modelo, la GUI y las vistas, únicamente falta cargar todos los scripts en el documento HTML principal e inicializar la aplicación, uniendo todas las piezas. En index.html nos aseguraremos que todos los scripts se cargan adecuadamente:

```

<link rel="stylesheet" href="css/jquery.mobile-1.4.5.css" />
<script src="https://maps.googleapis.com/maps/api/js?sensor=false"></script>
<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 src="js/modelos/Ruta.js"></script>
<script src="js/modelos/Rutas.js"></script>
<script src="js/vistas/NuevaRuta.js"></script>
<script src="js/vistas/EditarRuta.js"></script>
<script src="js/vistas/ListaRutas.js"></script>
<script src="js/vistas/Mapa.js"></script>
<script src="js/app.js"></script>

```

De la inicialización de la aplicación y de unir todas las piezas del puzle se encargará el fichero principal de nuestra aplicación /js/app.js. Aquí crearemos el modelo, las distintas vistas y conectaremos las vistas con el modelo y con la GUI.

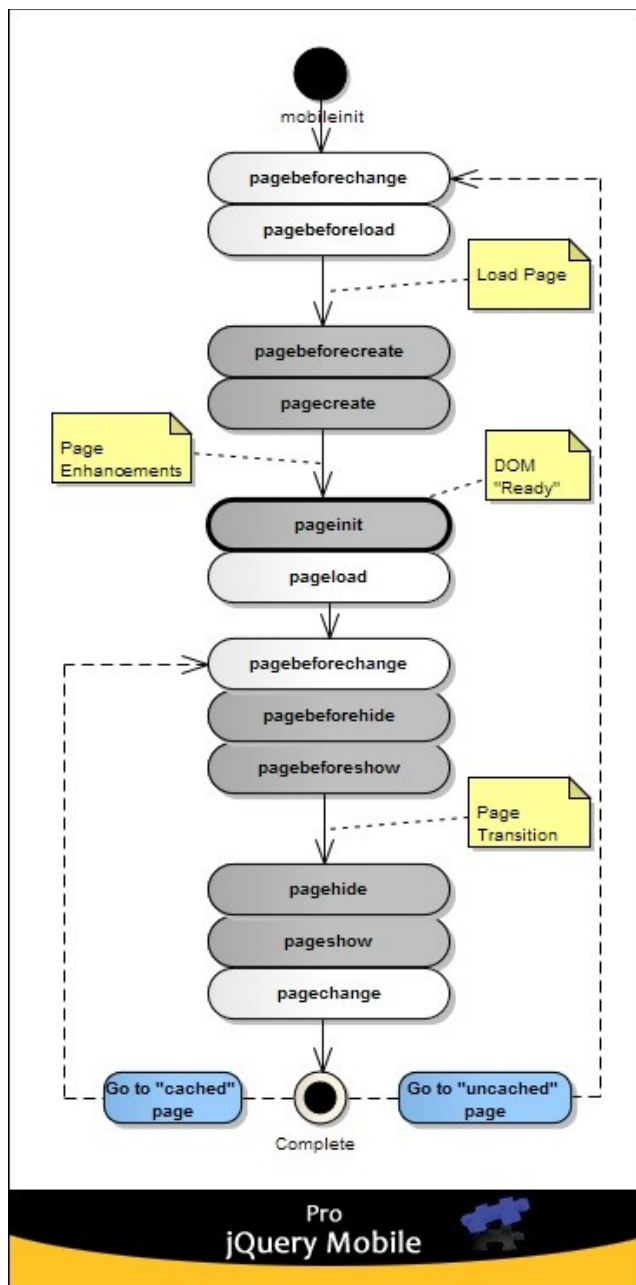
Lo primero consistirá en crear nuestra base de datos; bastará con el siguiente código:

```

// crear bd
var rutas = new Rutas();

```

Después crearemos la vista NuevaRuta. Sin embargo, si lo hacemos en el instante de carga del fichero app.js podríamos encontrarnos un error. Se debe recordar que en el constructor de la vista NuevaRuta se refresca la GUI invocando `.render()`, y el refresco implica acceder a varios widgets de jQuery Mobile y actualizarlos. Resulta que jQuery Mobile efectúa una carga “perezosa” de las páginas, es decir, una página se inicializa (se enriquece) la primera vez que se muestra. Como todavía no se ha mostrado la página principal 'pgHome' entonces los controles no se han enriquecido todavía, y cuando nuestra vista intenta acceder a ellos se encuentra con que no son widgets jQuery Mobile todavía, obteniendo un error. Por ello, la creación de la vista (y primer invocación de `.render()`) debe efectuarse cuando la página 'pgHome' haya sido enriquecida y esto sucede cuando el evento 'pageinit' se dispara sobre la página en cuestión. Para mayor comprensión de los estados por los que pasa una página en jQuery Mobile se adjunta el siguiente diagrama:



Al crear la vista NuevaRuta es necesario conectarla a la colección de rutas y a la GUI, en concreto a la página 'pgHome'. A continuación se muestra el código necesario para hacerlo una vez se ha enriquecido la página 'pgHome':

```
$(document).on('pageinit', '#pgHome', function() {
    var vistaNuevaRuta = new NuevaRuta({collection: rutas, el: '#pgHome'});
});
```

Con una explicación similar a la anterior, la vista ListaRutas debe crearse una vez haya sido enriquecida la página 'pgMisRutas'. Esta vista debe conectarse con la colección de rutas y al panel (div) 'pnRutas' contenido en la página 'pgMisRutas'. Además, es necesario recordar que esta vista depende de otra, de EditarRuta, ya que al seleccionar una ruta de la lista, el control se redirigirá a aquella. Es por tanto necesario crear la vista EditarRuta antes de ListaRutas, y conectarlas con el método ListaRutas.editarRuta() que definimos más arriba.

```
$(document).on('pageinit', '#pgMisRutas', function() {
    var vistaEditarRuta = new EditarRuta({collection: rutas, el: '#pgEditarRuta'});
    var vistaListaRutas = new ListaRutas({collection: rutas, el: '#pnRutas'});
    vistaListaRutas.editarRuta(vistaEditarRuta);
});
```

```
});
```

Por último, debemos crear la vista Mapa en el momento adecuado (evento 'pageinit' de la página 'pgMapa') y conectarla con la colección de rutas y con la página 'pgMapa'.

```
$(document).on('pageinit', '#pgMapa', function() {  
    var vistaMapa = new Mapa({collection: rutas, el: '#pgMapa'});  
});
```

8. Extensiones

Siguiendo los pasos de las secciones anteriores deberíamos de disponer de una webapp funcional, que nos permite registrar rutas tomando lecturas del dispositivo GPS y registrándolas en local. 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 Aplicación nativa

Se plantea transformar la webapp anterior en nativa utilizando para ello el toolkit PhoneGap estudiado en las clases de teoría. Para ello será necesario seguir los pasos mencionados en clase. Se propone transformar la webapp en una aplicación nativa Android.

8.2 Almacenamiento persistente

Como se puede observar fácilmente, la lista de rutas no se recuerda entre ejecuciones diferentes de la webapp. Con esta extensión se pretende que las rutas queden almacenadas en el almacenamiento persistente del navegador. Se propone para ello hacer uso de localStorage. Se recomienda mantener la misma estructura de webapp que se ha implementado, y hacer las siguientes modificaciones simples:

- Crear un método import() en la colección Rutas (/js/modelo/Rutas) que permita importar todas las rutas guardadas en localStorage en la colección de Backbone.js. Debe recordarse que en localStorage únicamente es posible almacenar String, y por tanto será necesario efectuar conversiones con JSON.parse() y JSON.stringify(). Este método se invocará al cargar la webapp.
- Detectar cualquier cambio en la colección Rutas y actualizar dicho cambio en localStorage, para mantener la colección Rutas y el almacenamiento persistente localStorage sincronizados. Para esto, en la colección Rutas será necesario registrar varios manejadores de los eventos "add", "remove" y "change".
- Revisar la estrategia de generación de identificadores únicos de las rutas. Actualmente se utiliza una utilidad de Underscore.js, que no funcionará en el caso de trabajar con almacenamiento persistente. Será necesario idear un sistema de generación de identificadores universales.

8.3 Optimización del refresco de la GUI

En varios puntos de este documento se ha apuntado que el refresco de la GUI no se estaba efectuando de una manera muy eficiente, ya que se limpiaba y se volvía a pintar por completo. Se plantea esta extensión con el objetivo de optimizar este proceso. La idea consiste en detectar qué partes del modelo han cambiado y refrescar en la GUI únicamente las partes afectadas por dichos cambios.

8.4 Objetos georreferenciados

Una extensión muy interesante de la webapp consiste en añadir información extra a las rutas. Durante una grabación de ruta el usuario podrá adjuntar un objeto georreferenciado, pudiendo ser éste una imagen (foto adquirida en el momento), un audio, un vídeo, una nota, etc. Dicho objeto se registrará en la ruta con la posición GPS actual. Posteriormente podrán visualizarse dichos objetos como iconos o globos en la página de mapa, pudiendo incluso seleccionarlo y visualizarlo.

De todas las extensiones planteadas, ésta es la extensión más interesante, pero también la más compleja de llevar a cabo. Requiere extender el modelo de datos, de manera que una ruta además de almacenar posiciones también almacenará objetos georreferenciados. Si deseamos trabajar con fotografías o sonidos requiere trabajar sobre una webapp nativa. Por último, si mostramos iconos personalizados en el mapa requiere un estudio más profundo de Google Maps API v3.

8.5 Open your mind ...

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