UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Título del Proyecto:

App para la Gestión de Rutas Senderistas

Autor:

Sanz Monllor, Gorka

Director:

TESINA PARA LA OBTENCIÓN DEL TÍTULO DE:
Máster en Desarrollo de Aplicaciones sobre Dispositivos Móviles

Septiemb

Contenido

Introducción

Descripción del problema

Anteriormente a la existencia de los Smartphones, no había ninguna forma sencilla y eficiente por parte del usuario de poder registrar información de los viajes, de eventos importantes en su vida, de sus aficiones o bien de las rutas que haga un aficionado al senderismo. La única forma de poder hacer este registro es manualmente, escribiendo toda la información de las aficiones del usuario. Cuando aparecieron las primeras máquinas ya se empezó a poder ver ese tipo de información en varios entornos, como los domésticos. La aparición de los primeros Smartphones con las características de conectividad y localización hace que este seguimiento se pueda hacer desde cualquier punto de una forma sencilla y eficiente.

Objetivos

El objetivo principal de la aplicación es, en primer lugar, que el usuario pueda registrar toda la información relativa a las rutas que ha realizado y, en segundo lugar, que se pueda visualizar desde el móvil en cualquier lugar utilizando las características de conectividad y localización del móvil. De esta forma el usuario puede introducir información de las rutas que ha realizado, visualizar información de éstas y aprovechar al máximo las herramientas de conectividad y localización que proporcionan todos los smartphones.

Motivación

La motivación personal para realizar este tipo de aplicación es la necesidad de tener una aplicación que genere de forma simple y concisa información de las rutas que se han hecho en una determinada fecha. De tal forma que aparecerán las rutas realizadas ordenadas por fecha y para cada una de ellas se podrá averiguar la información del usuario asociada a ella. Hasta ahora se había encontrado varias aplicaciones pero desde mi punto de vista eran demasiado complicadas para la usabilidad. Se trataba demasiada información no relevante para el usuario que lo único que podía hacer era confundirlo. Con la aplicación desarrollada se trata únicamente la información específica de cada una de las rutas utilizadas.

Tecnologías utilizadas

La aplicación se ha desarrollado utilizando tecnologías punteras y novedosas, que utilizan muchas empresas en la actualidad. Las principales tecnologías que se han utilizado en el proyecto son las siguientes:

Node.js: es la plataforma sobre la que se va a ejecutar la aplicación Web. Tiene herramientas para instalar librerías y para lanzar aplicaciones Web.

Rest: es la tecnología que permite la obtención de datos de una base de datos a partir de utilizar una serie de peticiones http que en nuestro caso nos van a permitir interactuar con la base de datos y que están implementadas en el backend (DELETE, POST, PUT, GET).

BackBone.js: Es la tecnología que va a permitir la utilización de todos los servicios REST con todas sus llamadas (DELETE, POST, PUT ,GET), así como modelar datos para poder tratar con todas las llamadas del tipo REST , gracias a los métodos de guardado/borrado que utiliza.

JQuery Mobile: Es una tecnología que se utiliza para el desarrollo de la parte de front end en aplicaciones para dispositivos móviles. Esta optimizada para este tipo de dispositivos y los desarrollos son bastante sencillos y eficientes.

SQLite: es la base de datos que se va a utilizar para guardar toda la información. Está muy optimizada para trabajar con node.js y su uso es relativamente sencillo, ya que para usarlo hace falta importar el paquete del driver, crear una conexión y ejecutar la respectiva sentencia SQL y procesar los resultados. Permite utilizar Callbacks de forma sencilla para tratar los resultados y ejecutar sentencias en paralelo de forma asíncrona.

Heroku: es la plataforma de servicio sobre la que se va a desplegar la aplicación con Node.js. Está optimizada para la ejecución de los servicios Rest y para la instalación de la aplicación de tal forma que se pueda acceder a la aplicación desde fuera sin ningún problema.

Aplicación Desarrollada

Prerrequisitos

Todas **las pruebas de la aplicación móvil se han hecho sobre un Samsung Galaxy que tiene el navegador Google Chrome instalado**. Es muy importante que el navegador Google Chrome esté instalado, ya que es el único que soporta la funcionalidad de las tecnologías utilizadas: JQuery Mobile , Node.js y Backbone.

Aplicación Desarrollada

Un apartado importante de cara a la presentación de la aplicación es establecer la aplicación que ha sido desarrollada. La aplicación de rutas es una aplicación que se ha desarrollado en un servidor web y que está alojada en Heroku. Toda la funcionalidad está ahí. La url donde se ha alojado la aplicación sería la siguiente:

https://desolate-hollows-42099.herokuapp.com/

Por otro lado se ha desarrollado una aplicación Nativa en Android que lo que hace es, a partir de una componente WebView, acceder a esa url y mostrarla por pantalla. Al presionar sobre el fichero .apk se abrirá un navegador sobre el que se ejecutará la anterior URL y sobre el que se puede añadir y modificar rutas. Es muy importante que el Smartphone que el usuario esté utilizando tenga el navegador GoogleChrome instalado, ya que el JQueryMobile y la funcionalidad de node que se utiliza solo se puede ejecutar en Google Chrome. En otros navegadores como el Webkit esta aplicación no se podrá ejecutar.

Ejecutar la aplicación Web en Local

Básicamente, hay dos formas de ejecutar la aplicación Web:

Una primera es desde la url que está en el servidor de Heroku:

https://desolate-hollows-42099.herokuapp.com/

Una segunda es en local. Para ejecutar la aplicación Web en una máquina en local se tendrían que hacer los siguientes pasos:

- Se descarga del repositorio github el código fuente de la aplicación con el siguiente comando:

  sudo git clone https://github.com/gsanz/FinalProjectMaster

- Ahora una vez descargado se ejecutan los siguientes comandos:

  cd rutaSenderistaWeb

```
cd node_modules
sudo npm  install
```

Las principales modificaciones se deberían hacer en los ficheros Ruta.js y Rutas.js y serían las siguientes:

Ruta.js

```
Var Ruta = Backbone.Model.Extend({

urlRoot: 'http://localhost:8080/misrutas/rutas',

……..

})
```

Rutas.js

```
Var Rutas = Backbone.Collection.extend({

    url: 'http://localhost:8080/misrutas/rutas'

……..

})
```

Con los anteriores comandos se instalan todas las librerías necesarias para la ejecución del proyecto. Ahora se tendrá que iniciar el proyecto con el comando

```
npm start
```

y para ejecutar la url se tendría http://localhost:8080/index.html

Arquitectura de la aplicación

Esquema del diseño

La estructura de la aplicación viene a ser la siguiente:

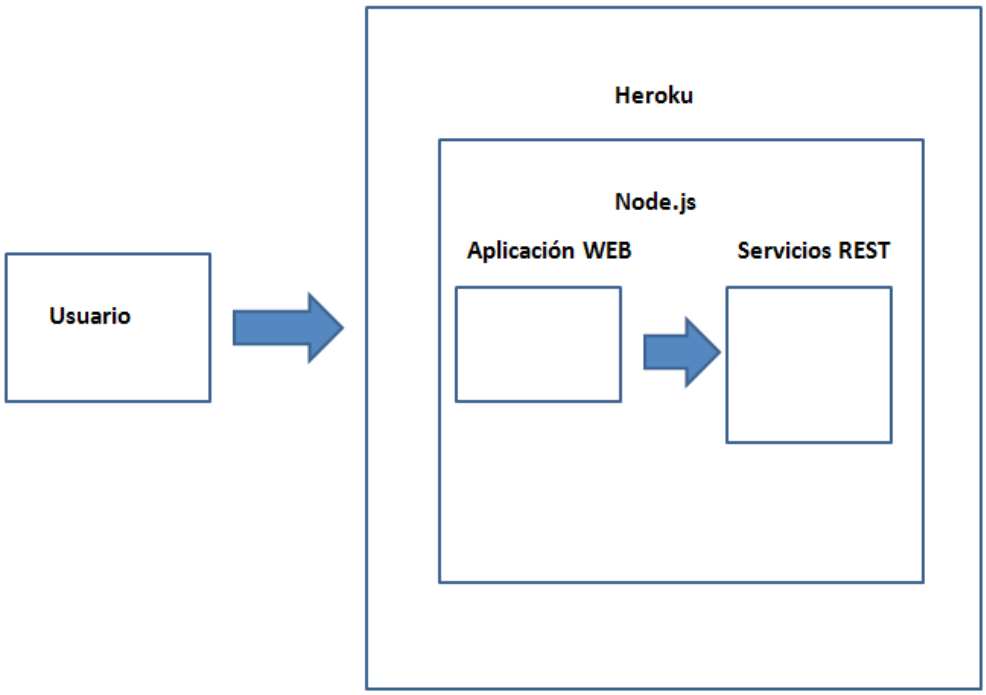

Figura 1 Esquema del Diseño

La aplicación se va a desplegar sobre la plataforma Heroku, que es una plataforma de servicio PaaS que permite lanzar aplicaciones Web realizadas sobre Node.js, así como deployarlas, depurarlas y manejar un sistema de control de las versiones que se van a instalar.

Sobre esta plataforma, que tiene preinstalado Node.js, se instalan las librerías necesarias para que funcione la aplicación. El usuario cuando lanza la aplicación ataca al fichero principal de Node.js, que ejecuta una aplicación Web y mediante servicios REST ataca a una base de Datos de donde obtiene toda la información.

La arquitectura de la aplicación es la siguiente:

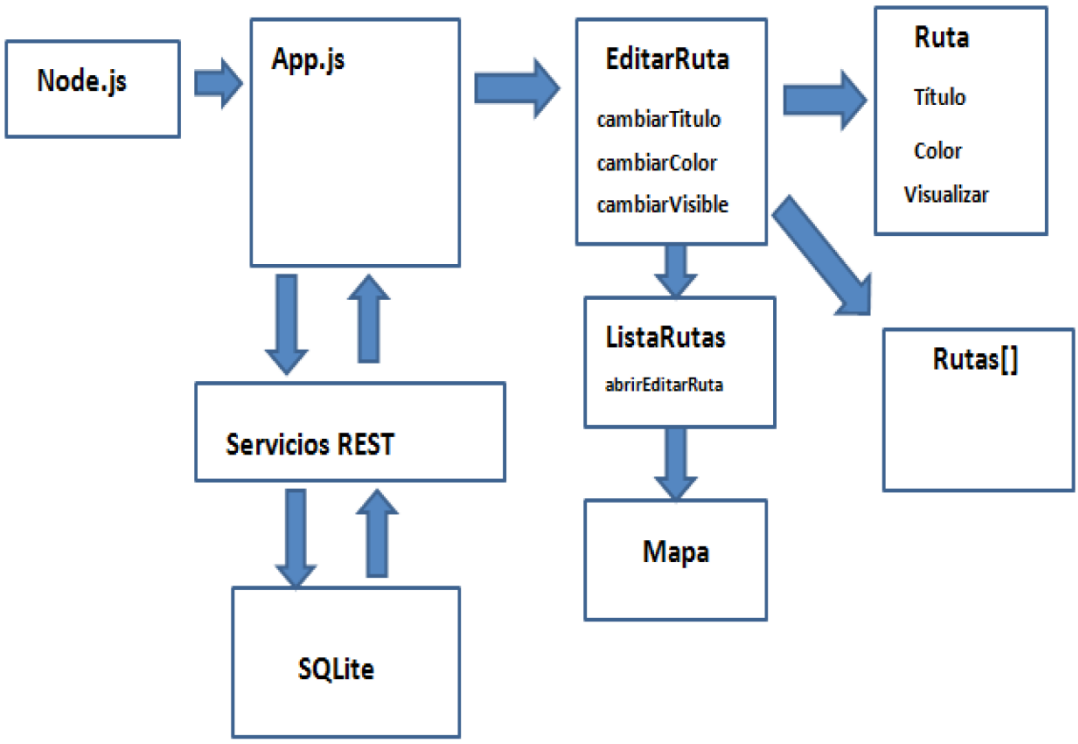

Figura 2 Arquitectura de la Aplicación

La clase principal de Node.js invoca a la clase principal de la aplicación Web y posee referencias REST para llamar a los eventos de las clases que heredan de Backbone para que se hagan las operaciones pertinentes. Desde la clase principal de la aplicación Web se invocan a los servicios REST, para hacer las operaciones de obtener todas las rutas (GET), modificar una ruta (PUT:id), obtener una ruta (GET:id) o borrar una ruta (DELETE). Todos estos Web Services interactúan con una base de datos SQLite que es donde se va a guardar toda la información.

El esquema básico REST de la aplicación es el siguiente:

Ilustración 1 Esquema REST de la Aplicación

Básicamente, se coge todos los datos de un servicio RESTFUL y se guarda en la colección del tipo BackBone. A partir de ahí se representa toda esa información en el Mapa.

Como clases principales para modelar con Backbone el tratamiento de las rutas estarían las clases EditarRutas y ListaRutas. En la clase EditarRutas interactuarían los métodos cambiarTitulo, cambiarColor y cambiarVisible sobre las propiedades de las rutas, cambiando el título, el color asociado a la ruta y determinando si la ruta es visible o no en el mapa. Habría también una clase de ListaRutas para guardar todas las rutas del usuario. Por otro lado estaría la clase Mapa, que se utilizaría para representar todos los puntos de las rutas, sobre un mapa que se obtiene al representar las rutas sobre el GoogleMaps.

Servicios REST

La parte de interacción de datos está hecha en base a servicios REST. Básicamente en la siguiente tabla aparecerá la relación de Servicios REST que se ha utilizado. Sería la siguiente:

URL	REST	SQL	MÉTODO
misrutas/rutas/id	POST	INSERT	Nueva Ruta
misrutas/rutas/	GET	SELECT	ListaRutas:abrirEditarRuta
misrutas/rutas/id	GET	SELECT	ListaRutas:abrirEditarRuta
misrutas/rutas/id	PUT	UPDATE	ListaRutas:abrirEditarRutas
misrutas/rutas/id	DELETE	DROP	ListaRutas:Remove

Figura 3 Servicios REST

Básicamente hay métodos para insertar, obtener, modificar y eliminar rutas. Las llamadas a los servicios REST aparecen estructuradas en los siguientes métodos:

Nueva Ruta: Genera una nueva Ruta introduciendo el nombre de la ruta, el color asociado a la ruta y la fecha de actualización de la ruta.

EditarRuta: permite modificar los parámetros de una ruta determinada, como el nombre o el color de dicha ruta.

Remove: Borra una ruta del vector de rutas.

Todos estos servicios REST interaccionan con la base de datos y todos los cambios que se hacen en el sistema se hacen directamente sobre ésta.

Modelo de datos

Base de datos

La base de datos que se utiliza es una Base de Datos SQLite formada por una única tabla. Sobre esta base de datos interactúan los servicios REST para guardar y obtener toda la información. La base de datos SQLite tiene las siguientes entradas:

Titulo: es el nombre de la ruta que el usuario ha estado haciendo. Es una variable del tipo Text.

Visible: Selecciona si la ruta es visible sobre el Mapa o no. Es una variable del tipo Text.

Color: Determina el color de la ruta que va a ser representada en el Mapa. Es una variable del tipo Text.

Posiciones Text: Es un String resultado de la conversión de json a String donde se representa los puntos de las rutas asociadas al camino que se está haciendo. Es una variable del tipo Text.

Date: representa la fecha en la que se ha terminado de grabar la ruta. Es una variable del tipo DATETIME.

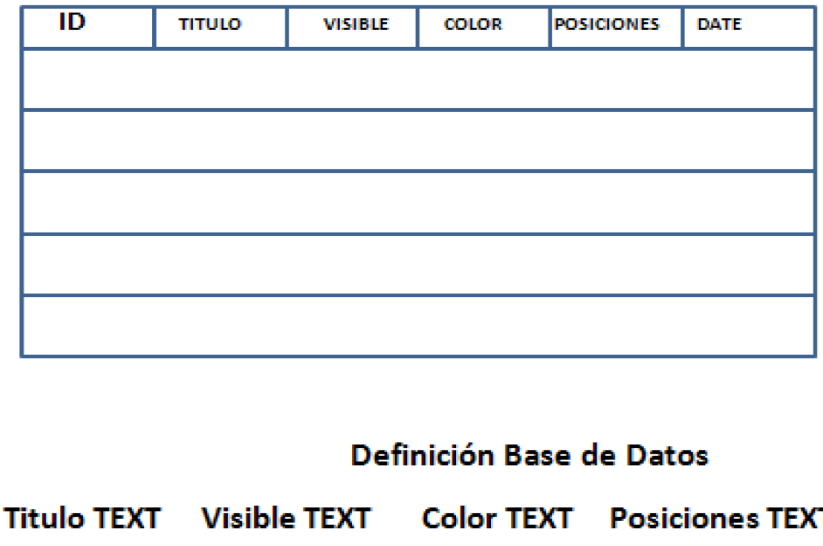

Figura 4 Base de Datos

Diagrama de Clases

Básicamente en Backbone.js el modelo de datos representa a la base de datos, aunque en este caso para guardar los datos se ha utilizado una base de datos Externa del tipo SQLite. Básicamente se tendrá que definir dos entidades Backbone.js:

- Una clase ruta, que extiende de la clase Backbone.model y que va a representar a cada una de las clases individuales.

- Una clase rutas, que extiende de la clase Backbone.Collection y que representará una base de datos; es decir, una colección de rutas.

A continuación un posible modelo de la implementación del fichero /js/modelos/Ruta.js. Para la asignación de identificadores únicos se utilizan técnicas especiales de BackBone.js.

En segundo lugar está la clase Rutas, que extendería de la clase de Backbone.js :**Backbone.Collection.Extend**. Esta clase de por sí ya tendría una estructura para modelar los datos. Lo que se define dentro de ella son eventos para añadir, borrar o modificar el vector.

El diagrama de clases aparecería en la siguiente figura:

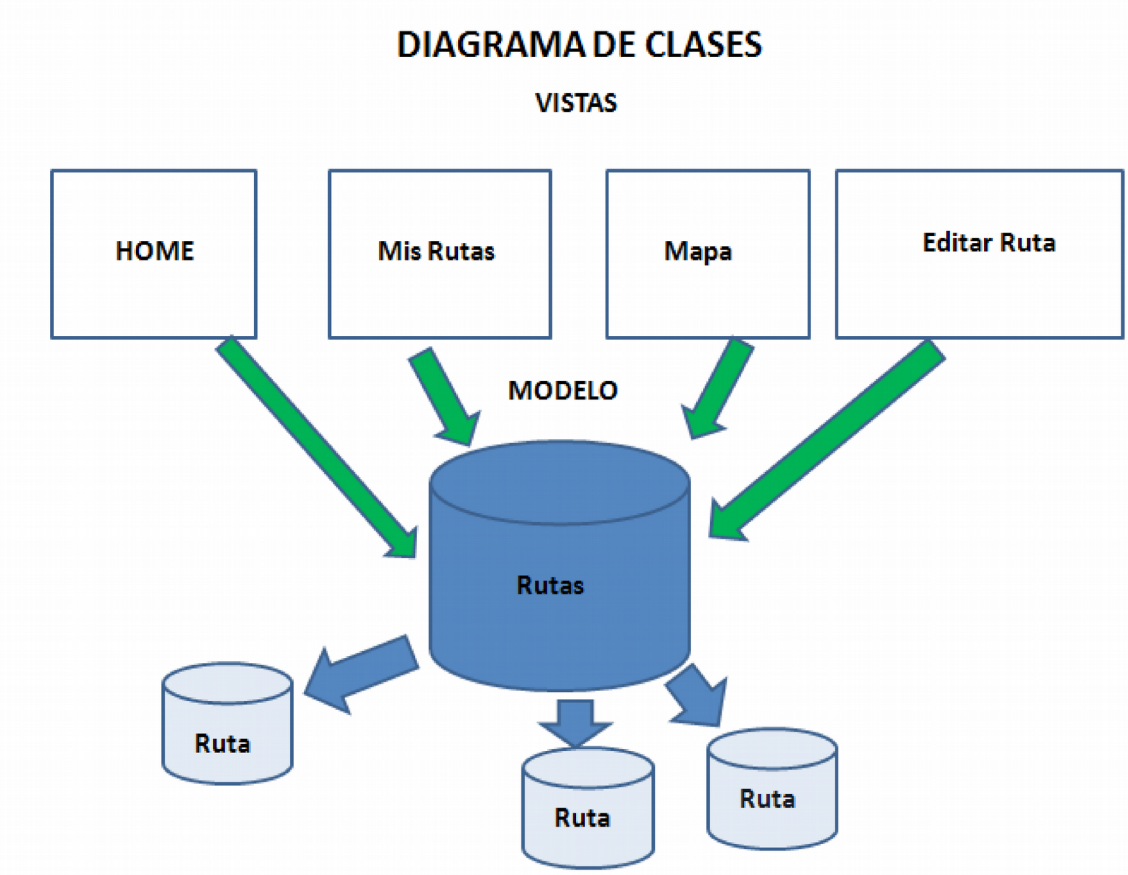

Figura 5 Diagrama de Clases

FRONTEND DE LA APLICACIÓN

El Front End de la aplicación se ha hecho utilizando la tecnología JQuery Mobile, una plataforma de desarrollo frontEnd adaptada para móviles. La estructura frontEnd de la aplicación está toda contenida en un único documento HTML, y es una estructura del tipo SPA (Single Page Application). Para que este modelo funcione bien hay que cerciorarse de que cada página tiene un identificador único. Para ello se crea un documento .html que contendrá a todas las páginas permitiendo una navegación fácil y accesible entre una página y otra, con un mecanismo de navegación entre páginas.

Tal y como se ha comentado anteriormente, la aplicación está estructurada en cuatro vistas:

- Vista Principal
- Ventana de Rutas
- Ventana de mapa

- Ventana de Editar Ruta

Las tres primeras tienen una cabecera común y un pie de página con una barra de navegación que les permitirá navegar entre ellas. La barra de navegación tendrá tres botones, cada uno de ellos activado en su correspondiente sección.

Las principales secciones serían las siguientes:

Ventana Principal

Hay una caja de texto para el título de la ruta, además de un botón y dos etiquetas que permiten escribir la ruta que se está grabando y cuánto tiempo se ha estado grabando.

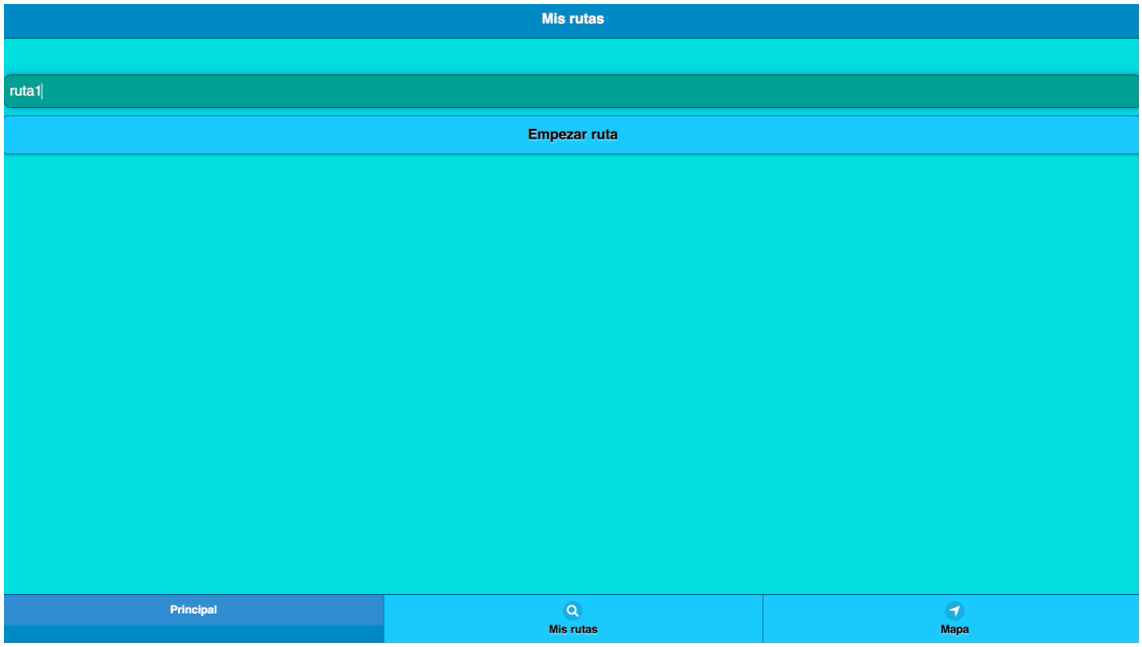

Ilustración 2 Crear una Ruta

Ventana de Mis Rutas

Se añade un ListView al contenido de la página con posibilidades de filtrado

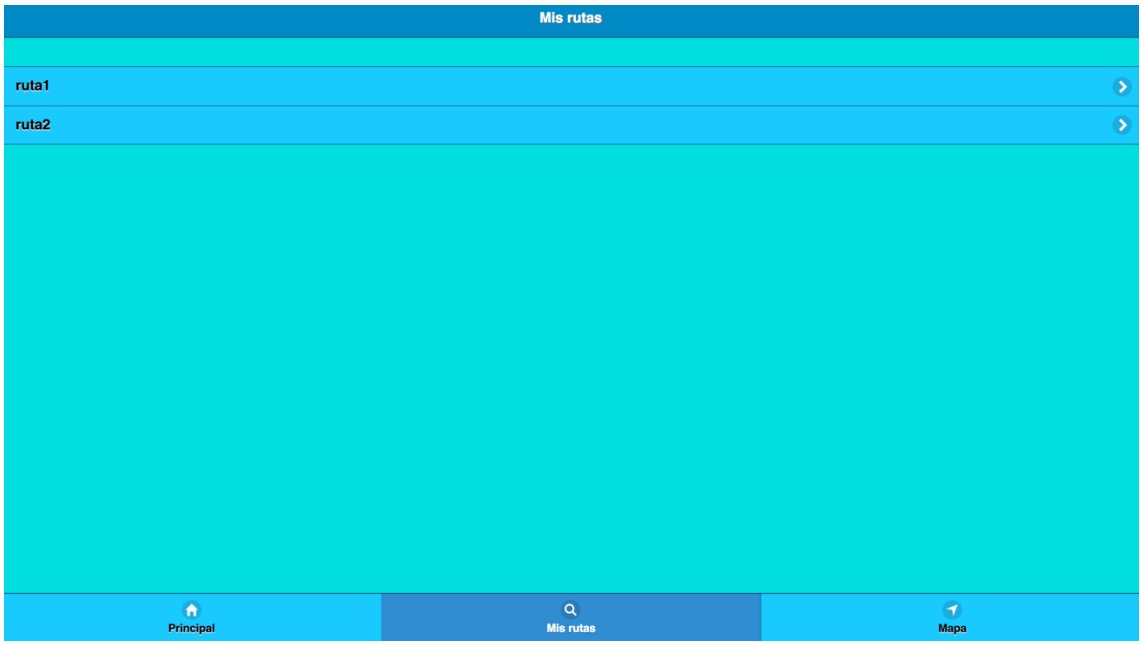

Ilustración 3 Visualizar todas las rutas

Editar Rutas

Aparecen los datos de la Ruta seleccionada y se puede modificar las características de la ruta pudiendo volver atrás.

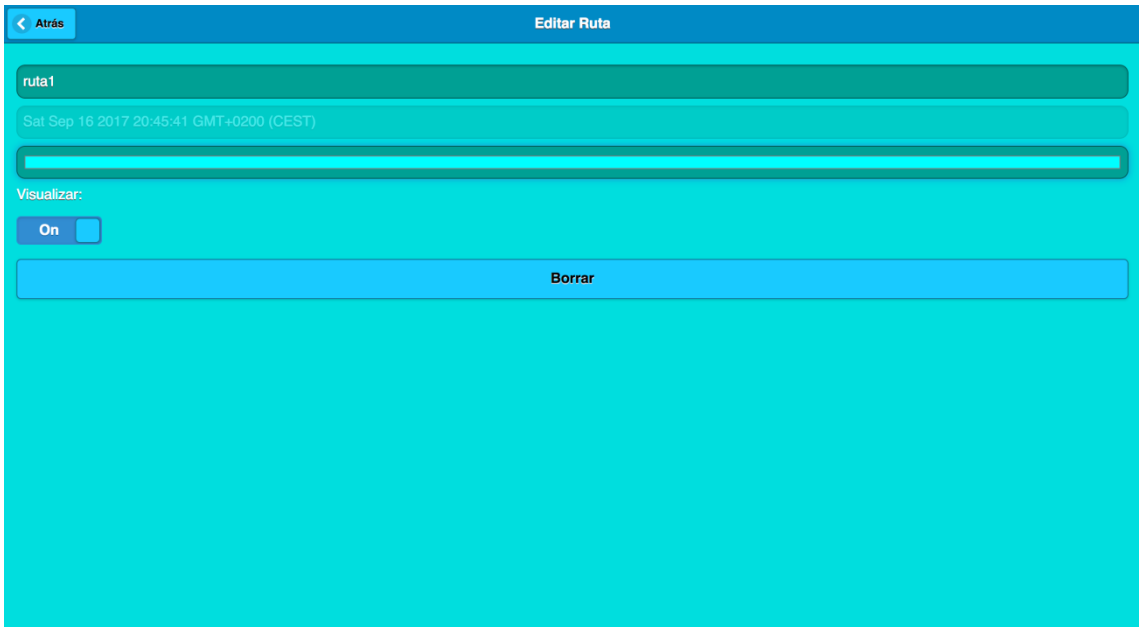

Ilustración 4 Ventana Edición de Rutas

Mostrar Rutas en Mapa

Permite visualizar todas las rutas en el mapa.

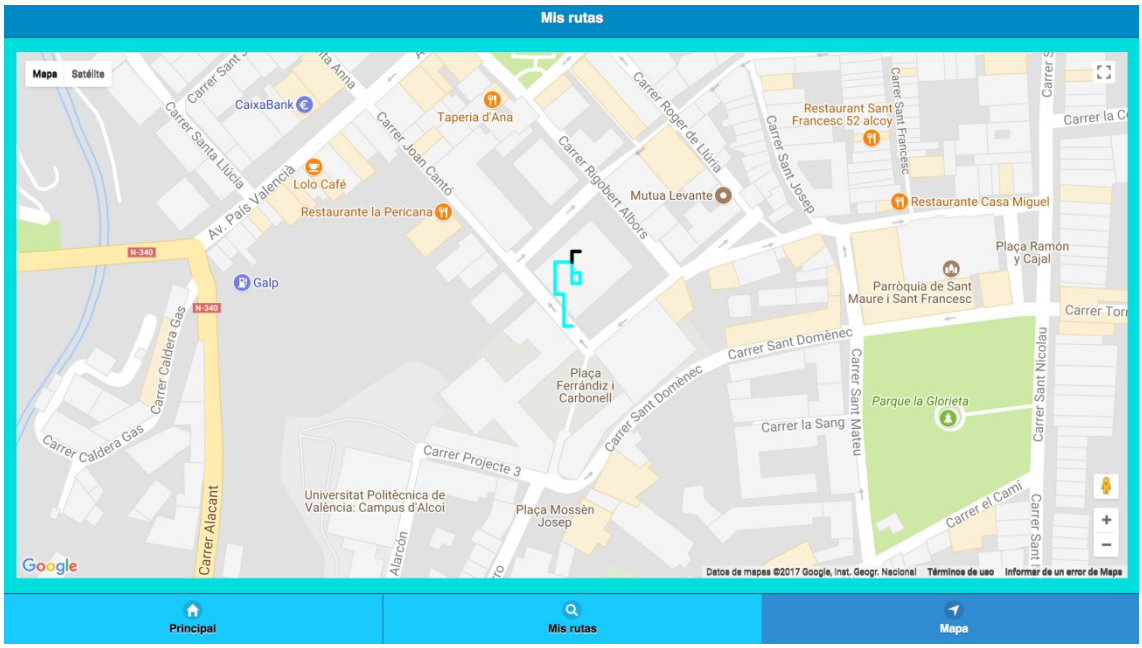

Ilustración 5 Mostrar rutas en el Mapa

DIAGRAMA DE CASOS DE USO DE LA APLICACIÓN

Si se hiciese un diagrama de casos de uso de la aplicación se tendría lo siguiente:

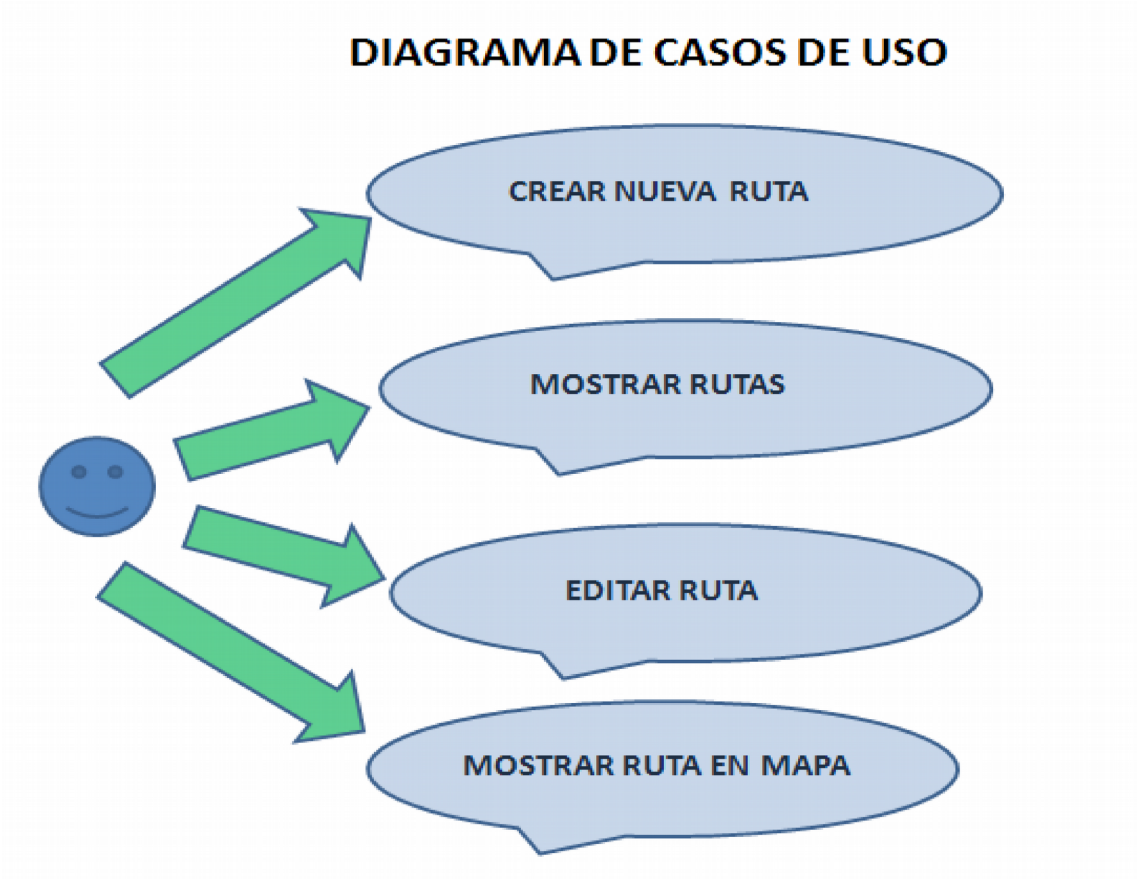

Ilustración 6 Diagrama de Casos de Uso

Básicamente va a haber 4 casos se uso:

Crear Nueva Ruta: se va a crear una nueva ruta.

Mostrar Ruta: se muestran todas las rutas que el usuario ha creado.

Editar ruta: Dada una ruta, se pueden modificar los parámetros característicos de dicha ruta: nombre de ruta, color o si la ruta es visualizable o no.

Mostrar ruta en Mapa: se muestran todas las rutas en el mapa con el respectivo color asignado.

Capítulos adicionales

Funcionamiento de la Aplicación

La aplicación está basada en cuatro vistas, cada una controlando los aspectos fundamentales de cada una de las 4 páginas de la aplicación:

Vista Nueva Ruta: se encargará de la creación de nuevas rutas.

Vista ListaRutas: controlará el listado de rutas de la aplicación

Vista Editar Ruta: controla la edición de todos los parámetros de la ruta.

Vista Mapa: controla la representación gráfica de las rutas en el mapa.

Relaciones entre Vistas

Las relaciones entre todas las vistas de la aplicación serían las siguientes:

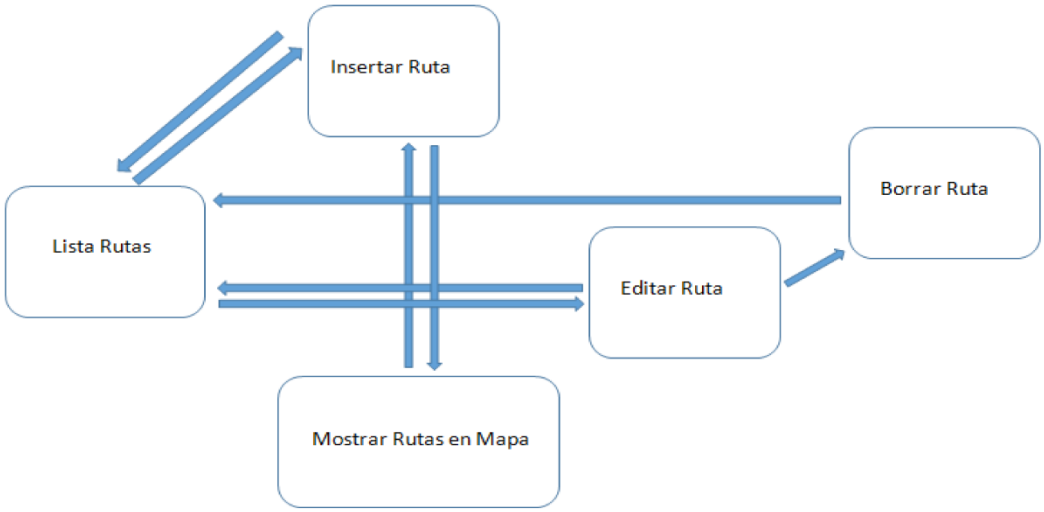

Ilustración 7 Relaciones entre Vistas de la Aplicación

La vista principal de la aplicación es InsertarRuta. Desde esta vista de inserta el nombre de la ruta. A partir de ahí se accede a ListaRutas(visualizar todas las rutas) o a mostrar las rutas en el mapa. Desde la vista de ListaRutas se puede acceder a EditarRuta. Desde EditarRuta se puede acceder a BorrarRuta o a ListaRutas. Desde

InsertarRuta se puede acceder a Mostrar Rutas en Mapa y desde Mostrar Rutas en Mapa se puede acceder a InsertarRuta.

Nueva Ruta

La vista debe habilitar este comportamiento:

1. Si no hay grabación en curso, cuando el usuario haga click en 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.

   Si hay una grabación en curso, cuando el usuario haga click en el botón, la vista finalizará la grabación de la ruta, haciendo lo siguiente:

   Finalizará el registro del tiempo de grabación

   Finalizará el registro periódico de posiciones GPS

   Añadirá el modelo Ruta a la colección Rutas

   Se refrescará la GUI cambiando el texto del botón a “Empezar Ruta” y ocultando toda la información sobre el estado actual de la ruta.

Serían las siguientes dos vistas:

Ilustración 8 Crear Nueva Ruta

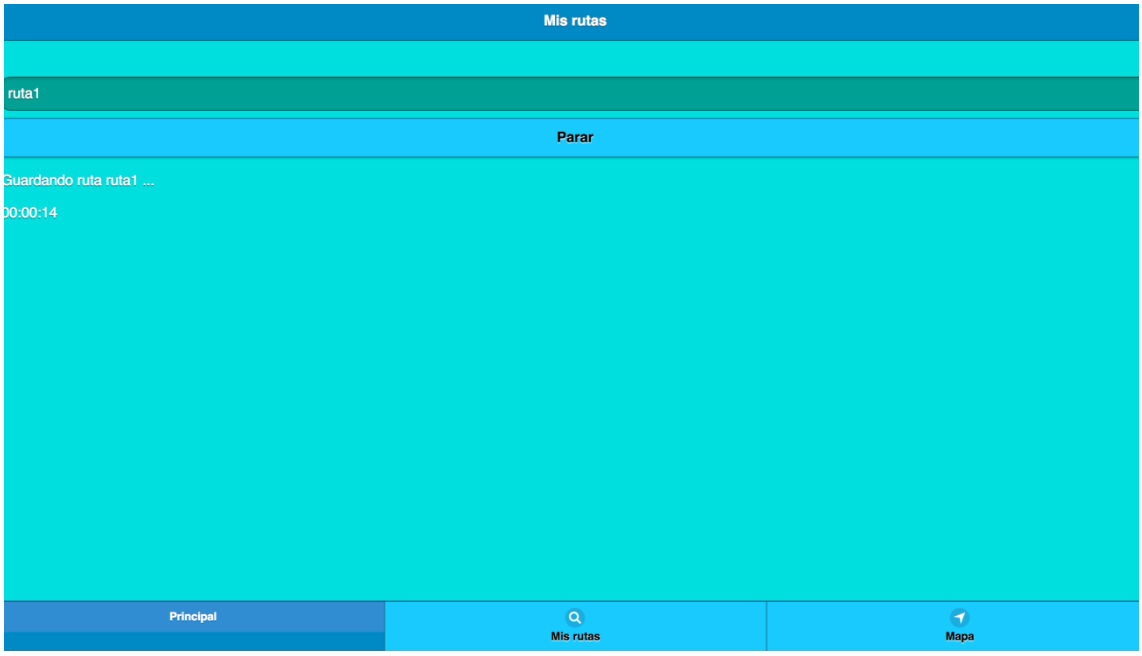

Ilustración 9 Iniciar Nueva Ruta

Para comenzar la definición de la vista anterior se definirá en el fichero NuevaRuta.js la clase NuevaRuta, que extiende de Backbone.View.
En el constructor de esta vista se inicializarán todos los atributos que se necesitará para implementar el comportamiento descrito anteriormente. Por un lado, para saber si estamos en (1) ó (2) se utilizará un atributo .grabando, que indicará si se está grabando o no. Cuando se está grabando, para habilitar refresco de reloj y recuperación de posiciones periódicas se necesitarán sendos timers .timerReloj y .timerGps. También se deberá recordar el número de segundos que han pasado desde que comenzó el grabado de la ruta. Por último, al crear la vista se pintará, invocando su método .render(). A continuación se presenta el código de inicialización asociado:

```
var NuevaRuta = Backbone.View.extend({

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

Ilustración 10 Inicialización de la Ruta

Una vez inicializada, el siguiente paso consiste en implementar el comportamiento descrito en (1) y (2). Se hará 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();
    },
```

Ilustración 11 Código de métodos de la Ruta

Como se puede observar, para recoger las lecturas del GPS se invoca cada segundo el método leerGps() de la vista. El período establecido inicialmente resultaría demasiado corto para una aplicación real, por lo que sería necesario incrementarlo. Se suministra 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);
},
```

Ilustración 12 Código obtener posición GPS

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. Sería el siguiente código:

```
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 + ":" + (
        this.$('#pnInfo').css('visibility', 'visible');
    } else {
        // hide panel  change button text
        this.$('#btGrabar').val('Empezar ruta').button('refresh');
        this.$('#pnInfo').css('visibility', 'hidden');
    }
},
```

Ilustración 13 Código de refresco de la Vista

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. Se definirá en el mapa de eventos de la vista.

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

Ilustración 14 Código de Lista de Eventos Grabar de la Vista

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.
Para implementar esta vista, primero se definirá 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 la aplicación.

```
<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>-->

            <li><a id="1" href="#pgEditarRuta">Ruta 1</a></li>
            <li><a id="2" href="#pgEditarRuta">Ruta 2</a></li>
            <li><a id="3" href="#pgEditarRuta">Ruta 3</a></li>
            <li><a id="4" href="#pgEditarRuta">Ruta 4</a></li>


        </ul>
    </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>
</div>
```

Ilustración 15 Código html de la aplicación de representación de rutas

A partir de ahí se crea una nueva Vista con todas las rutas, y se vincula 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. Se registrarán manejadores de todos estos eventos en el constructor de la vista:

```
var ListaRutas = Backbone.View.extend({
    initialize: function() {
    var self = this;
    console.log("+++EDITAR RUTAS+++");
    this.collection.on('remove', function() { self.render(); });
    this.collection.on('sync', function() { self.render(); });
    this.render();
},
```

Ilustración 16 Código de representación de ListaRutas

Al refrescar la vista se pintará el listado de rutas, eliminando el listado anterior y creándolo de nuevo. Esta estrategia no es eficiente, pero es simple y efectiva. El código sería el siguiente:

```
    this.$el.html('<ul datarole="listview" datafilter="true"></ul>'); // pintar rutas
    for(var i = 0; i < this.collection.size(); i++){
        var m = this.collection.at(i);
        var str = '<li><a id="' + m.id + '" href="#">' + m.get('titulo') + '</a></li>'
        this.$el.find('[datarole="listview"]').append(str);
    }
    this.$el.find('[datarole="listview"]').listview();
},
```

Ilustración 17 código para refrescar la vista

Por último, cuando el usuario presiona sobre una ruta se le debería dirigir a la página de editar ruta. Se trata de la única interacción que se debe controlar en el listado.Para ello, se define en el diccionario de eventos lo siguiente:

```
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();
}
 });
```

Ilustración 18 Código de Editar Ruta

La vista asociada aparecece en la ilustración 2.

Vista EditarRutas

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.
Se implementa 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');
},
```

Ilustración 19 Pintar controles en GUI

Por otro lado, la vista debe detectar cualquier actualización en los controles de la GUI y modificar el modelo convenientemente. Para ello, se registrarán 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');
    }
}
```

Ilustración 20 Eventos de captura de modificar ruta

La vista asociada aparece en la ilustración 3.

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.
En primer lugar se creará 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. Se eliminará de la página 'pgMapa' todo lo

relacionado con el mapa, que incluye todo el código JavaScript y el panel 'pnMapa'. Todo esto se integrará 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), se creará el contenedor del mapa 'pnMapa', e inmediatamente se pintará el mapa en su interior. Una vez el mapa ha sido creado y pintado por primera vez, se recorrerá la colección de rutas y se pintará aquellas rutas que sean visibles con el color correspondiente.
2. **El refresco**: cuando la colección de rutas es alterada, se debe refrescar las rutas que se han pintado en el mapa. En este caso, las actualizaciones que 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. Así se simplifica el proceso de renderizado.
(ii) detectar la/s ruta/s que ha/n sido modificada/s, borrarla/s y volver a pintarla/s. Es más óptimo pero complica la lógica del refresco.
En esta primera versión de la aplicación se adoptará 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();
        });

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

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

} ,
```

Ilustración 21 Código constructor de la Vista

El método .render() debe pintar las rutas en el mapa. Para ello, se hará uso de Polyline de Google Maps API v3, que permiten pintar una polilínea sobre un mapa.
Por tanto, cuando se refresque el mapa, se limpiará todas las polilíneas pintadas hasta el momento, y se volverá a pintar. Para poder limpiar las polilíneas pintadas en el último refresco es necesario mantener información sobre ellas. Se hará en el vector polylines[]. Los distintos puntos que configuran una polilínea deben ser objetos de tipo google.maps.LatLng, pero en las rutas, se dispone de objetos JavaScript con las propiedades {lat:xxx, lng:xxx}, por lo tanto es necesario hacer una transformación; se usa _.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 se muestra 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) {
                        console.log("LATITUDE " + coords.lat);
                        console.log("LONGITUDE" + coords.lng);
                        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);
    }
});
```

Ilustración 22 Código Asociado a Pintar Rutas

La vista asociada aparece en la ilustración 4.

Conclusiones

Grado de cumplimiento de objetivos

En principio los objetivos principales a cumplir eran los siguientes:

1-Desarrollo de una aplicación móvil para que el usuario pueda introducir rutas.

2- Inclusión de una base de datos SQLite en la aplicación para que se puedan guardar los datos cuando el usuario apague la aplicación.

3-Inclusión de nuevas tecnologías en la aplicación de generación de rutas, tales como JQuery Mobile, npm o backBone.js.

En principio, el objetivo básico de desarrollo de las aplicaciones se ha conseguido, si se tiene en cuenta el hecho de que se ha utilizado Node.js para poder iniciar la aplicación web , JQueryMobile para el desarrollo de la aplicación Web y Backbone.js para el modelado de las clases. Son tecnologías web muy novedosas a la vez que eficientes.

La aplicación nativa ha sido desarrollada y, por otro lado la utilización de una Base de Datos SQLite ha sido una nueva funcionalidad que se ha conseguido incluir en la aplicación.

Líneas abiertas

Básicamente hay dos principales líneas abiertas:

1- Realizar un sistema multiusuario, donde haya un administrador y varios usuarios, cada uno de los cuales pueda acceder a una determinada información específica de la aplicación. De tal forma que hay varios usuarios y cada uno puede poner todas sus rutas. Por otro lado, un usuario puede acceder a ver todas las rutas del resto de usuarios.
2- Incluir una mayor cantidad de información en las rutas, aparte de la información que éstas tienen actualmente como título, color o si se representa en un mapa o no.

Consideraciones Personales

Desde el punto de vista personal he mejorado mis conocimientos en diferentes tecnologías como las que he dicho anteriormente y mis conocimientos sobre aplicaciones Web. He tocado tecnologías punteras como son Node.js y JQueryMobile, que me han servido para el desarrollo de proyectos en mi trabajo.

Por otro lado, he cumplido el objetivo de hacer una aplicación para poder visualizar, rutas, que era una de mis aficiones desde siempre.

Anexos

Listado de fuentes entregadas / Código fuente en GitHub

Todo el código desarrollado se ha dejado en la siguiente url de GitHub:

https://github.com/gsanz/FinalProjectMaster.

En este repositorio aparecerá la siguiente información:

1-rutaSenderismo.apk: es el .apk asociado a la aplicación de rutas senderistas

2-rutaSenderismo: es el código fuente asociado a la aplicación Android para hacer Rutas Senderistas

3-rutaSenderistaWeb: es el código fuente asociado a la aplicación Web que está desplegada en el servidor Heroku

Manual de usuario

Se adjunta en el fichero de Word Manual de Usuario.