

Red Social con Geo-Localización de lugares basado en tecnología Ruby on Rails

Edmundo Figueroa Herbas

26 de octubre de 2012

Índice general

1. Ruby on Rails y patrones Web 2.0	1
1.1. Porque usar Ruby on Rails para desarrollar una aplicación web ?	1
1.2. Patrones de diseño de la Web2.0	3
1.2.1. Patrones de diseño	3
1.2.2. REpresentational State Transfer (REST)	4
1.2.3. MVC	6
1.2.4. Mashup	9
1.3. Conclusión	12

Capítulo 1

Ruby on Rails y patrones Web 2.0

1.1. Porque usar Ruby on Rails para desarrollar una aplicación web ?

La gran propaganda de Ruby on Rails (RoR) o más sencillamente Rails se basa en el rápido desarrollo de aplicaciones web, conocido como agile development. Como parte de su construcción, Rails maneja las filosofías *DRY*¹ y *convención sobre configuración*.

No te repitas(DRY) según el creador de RoR, David Heinemeier Hansson, significa que cada pieza de conocimiento en un sistema debe ser declarado en un solo lugar.[1] Esto lo logra gracias al patrón Modelo-Vista-Controlador², y el lenguaje multiparadigma Ruby sobre el cual está construido Ruby on Rails.

Convención sobre configuración significa que Rails tiene parámetros por defecto para casi todos los aspectos que mantiene unida una aplicación, ya que se logro observar que la gran mayoría de aplicaciones web compartían la misma configuración inicial, siguiendo las convenciones de Rails se llega a simplificar el código escrito en una aplicación, también se agiliza el desarrollo ya que no es necesario tomar decisiones sobre características básicas que se necesita implementar.

¹Don't Repeat Yourself

²MVC

David Heinemeier cita en su libro [1], que *“Rails es Ágil porque simplemente la agilidad es parte de su construcción”*.

Se puede analizar esta afirmación teniendo en cuenta los principios del **manifiesto por el desarrollo ágil de software**³:

- **Individuos e interacciones** sobre procesos y herramientas
- **Software funcionando** sobre documentación extensiva
- **Colaboración con el cliente** sobre negociación contractual
- **Respuesta ante el cambio** sobre seguir un plan

Rails se enfoca bastante en conseguir un prototipo funcional en muy poco tiempo y sobre ese prototipo seguir incrementalmente hasta conseguir una aplicación de calidad.

Los más grandes obstáculos que se enfrenta una aplicación en el tiempo es el mantenimiento y escalabilidad, actualmente se estima que existen 230,000 websites[2] desarrolladas sobre RoR entre ellas se puede nombrar a GitHub, Hulu, Yellow Pages. Son sitios con miles de visitas diarias con una alta carga del servidor y son un claro ejemplo de que Rails puede manejar sitios de alto perfil.

Los detractores de Rails sostienen que escalar una aplicación construida sobre RoR es muy difícil pero los defensores argumentan que lo que se tiene que escalar es el código de la aplicación no el framework.

Twitter nació sobre Ruby on Rails y no fue hasta que era un servicio usado a nivel mundial y manejaba millones de request por día que empezaron a surgir problemas debido a que Ruby no estaba optimizado para un trabajo muy pesado, según palabras de Alex Payne, Twitter developer, “Ruby es lento”[3]. Actualmente Twitter migró su backend a Scala, framework basado en Java(que esta mas optimizado que Ruby), pero para su front-end no cambian a Rails.[4]

Se puede agregar que Rails es una muy buena opción a la hora de empezar cualquier proyecto web, ya que implementa las herramientas necesarias para un desarrollo ágil, sólido y de calidad respaldado por un modelo de

³<http://agilemanifesto.org/iso/es/>

desarrollo basado en pruebas(TDD⁴), las filosofías DRY y convención sobre configuración. y cuando la aplicación haya crecido y empiecen a aparecer los problemas es decisión de los programadores el ver si mantener el código actual y parchearlo o cambiar de tecnología para mejorar el rendimiento y la experiencia del usuario

1.2. Patrones de diseño de la Web2.0

Que es la Web2.0 ?, primeramente se debe explicar que este término fue acuñado por 1999 para describir paginas web que usaban tecnologías mas alla de las simples estaticas paginas web.

No fue hasta que en el 2004 en la conferencian sobre la Web2.0 que se popularizo este termino, y asi mismo como la Web que evoluciona, la definicion se actualiza con el tiempo, y Tim O'Reilly trato de definirla en su articulo “*What is Web 2.0*”[5], articulo que se puede considerar como la guia de referencia para cualquier persona que quiera entender que es la Web2.0 y sus origenes, en un articulo posterior “*Web 2.0 Compact Definition: Trying Again*”[6] del que se puede extraer la siguiente definición:

“Web 2.0 is the business revolution in the computer industry caused by the move to the Internet as a platform, and an attempt to understand the rules for success on that new platform. Chief among those rules is this: build applications that harness network effects to get better the more people use them.”

–Tim O'Reilly

En resumen se puede definir que una aplicación web2.0 es aquella que mejora y crece con la participación activa de sus usuarios.

“Software que mejora mientras más gente la usa” [5]

1.2.1. Patrones de diseño

Un patrón de diseño es una solución general, reusable y flexible que describe cómo resolver algún problema general en el desarrollo de software,

⁴Test Driven Development

un patrón puede ser usado y modificado según el problema al cual se está aplicando.

Se pueden observar los siguientes patrones de diseño en la aplicación:

- **REST**
- **MVC**
- **Mashup**

1.2.2. REpresentational State Transfer (REST)

REST es un término descrito por Roy Fielding en su tesis doctoral “*Architectural Styles and the design of Network-based Software Architectures*”[7], describe estilos arquitectónicos de sistemas interconectados por red.

REST es un estilo arquitectónico que especifica cómo los recursos van a ser definidos y direccionados, especifica la importancia del protocolo *cliente-servidor-sin estado*, ya que cada request tiene toda la información necesaria para entenderla.

En el contexto de Rails, REST significa que los componentes del sistema por ejemplo los usuarios son modelados como recursos que pueden ser creados, leídos, actualizados y borrados, estas acciones corresponden a las operaciones CRUD (Create, Read, Update, Delete) de las bases de datos relacionales y a los cuatro operaciones fundamentales POST, GET, PUT, DELETE definidos en el protocolo HTTP.

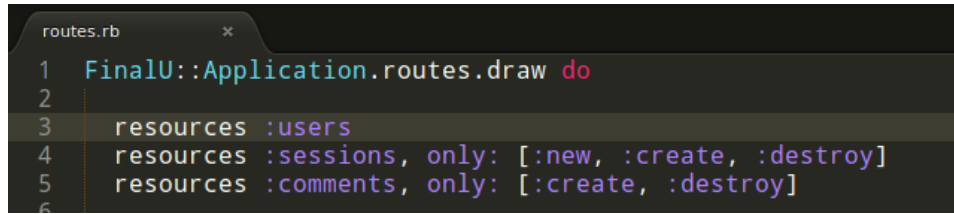
En Rails el estilo de desarrollo RESTful⁵ ayuda a determinar acerca de qué controlador y cuál será la acción que se ejecutará, solamente procesando el request HTTP hecho al servidor.

GET es la operación HTTP más común, es usado para leer datos en este caso páginas, se puede leer como “get a page”, **POST** es la operación que se usa cuando se ejecuta un formulario, en la convención de Rails **POST** se usa para crear objetos o recursos, **PUT** se usa para actualizar objetos, **DELETE** es usado para borrar objetos. Los browsers actuales no son capaces de manejar las operaciones **PUT** y **DELETE** de forma nativa, por lo que Rails usa un pequeño truco que consiste en declarar el método que se está

⁵se denomina RESTful a los sistemas que siguen los principios REST

enviando en un *hidden field* en el formulario HTML.

Para lograr todo este comportamiento es necesario declarar, en el archivo que controla las rutas dentro de la aplicación, **routes.rb**, que el recurso **user** es *restful*, tal como se muestra en la figura 1.1



```

1 FinalU::Application.routes.draw do
2
3   resources :users
4   resources :sessions, only: [:new, :create, :destroy]
5   resources :comments, only: [:create, :destroy]
6

```

Figura 1.1: config/routes.rb

La figura 1.1 muestra como se declara a **users** con todas las acciones restful los cuales se listan en el cuadro 1.2.2, Rails también permite declarar solamente algunas acciones restful, como se ve el recurso **sessions** solamente tiene las acciones de **new**, **create** y **destroy**.

HTTP request	URL	ACCIÓN	USADO PARA
GET	/users	index	mostrar todos los usuarios
GET	/users/new	new	genera un formulario HTML para crear un nuevo usuario
POST	/users	create	crea un nuevo usuario
GET	/users/1	show	muestra un usuario especifico
GET	/users/1/edit	edit	genera un formulario HTML para editar un usuario
PUT	/users/1	update	actualiza los datos de un usuario especifico
DELETE	/users/1	destroy	destruye un usuario

Cuadro 1.1: las posibles rutas que se generan

Tal como se ve en el cuadro 1.2.2, Rails maneja los request HTTP de acuerdo con el tipo de llamada que se realice, este trabajo lo realiza el **router**,

que reconoce las URLs y los despacha a una **acción** del controlador, todo este proceso ya está implementado en el núcleo de Rails por lo tanto es automático y el programador no necesita más configuración que la mostrada en la figura 1.1, obedeciendo al principio de *Convención sobre configuración*

Por ejemplo, si se genera una petición GET hacia la dirección `/usuarios/1` el servidor interpreta la dirección y responde mostrando la información del usuario “1” ejecutando la acción **show** del controlador `usuarios` y en cambio si se genera una petición PUT a la misma dirección `/usuarios/1` el router procesa la información y ejecuta la acción **update** del controlador `usuarios` actualizando la información del usuario “1”.

Esta convención de Rails ayuda a entender de mejor manera el flujo que tiene un recurso, las URL son legibles y únicos para cada recurso. La implementación de los recursos se hace de forma más limpia y ordenada, situaciones que son claves para el mantenimiento y la extensibilidad del sistema.

1.2.3. MVC

MVC (Modelo Vista Controlador) es un patrón arquitectónico que separa los datos de la aplicación en la interfaz del usuario y la lógica del negocio, en tres partes cada uno especializado para su tarea, la vista maneja lo que es la interfaz del usuario, puede ser gráficamente o solo texto, el controlador interpreta las entradas del teclado, mouse, o los cambios de la vista de la mejor forma posible y finalmente el modelo maneja el comportamiento de los datos de la aplicación.[8] Concepto que se desarrolló en 1979 por Trygve Reenskaug el cual da una solución al problema de separar la lógica del negocio de la lógica de la presentación.

Rails está construido sobre el patrón MVC, esto significa que para cada pieza de código existe un lugar predeterminado y todas las piezas de código de la aplicación interactúan de forma predeterminada.

Modelo Representa la información o los datos y contiene las reglas o métodos para manipular estos datos. En el caso de Rails, los modelos son usados principalmente para manejar la interacción con las tablas de la Base de Datos, en la que cada tabla corresponde a un modelo en la aplicación, para nombrar a estos modelos existe una simple convención que es la de nombrar a la tabla en forma plural, mientras que el modelo tiene el

nombre en singular. El Modelo, en Rails es el principal encargado de manejar la *logica del negocio*.

Vista Representa la interfaz de la aplicación. En Rails las vistas son archivos HTML con código Ruby embebido⁶ que realiza la tarea de representar los datos. Las Vistas son las que generan los datos que son enviados al navegador web.

Controlador Es el encargado de interactuar entre el modelo y la vista, procesando los datos enviados en el request del navegador web, llamando a métodos del modelo para conseguir *información* de la base de datos, posteriormente el controlador envia esta *información* a la vista.

El Controlador y la Vista son los encargados de manejar la *lógica de la presentación*.

Se puede apreciar el comportamiento de este patrón, en la figura 1.2

1. se genera un request “**GET /users/1**” desde el navegador web.
2. este request primeramente es analizado por el **router**, que siguiendo el principio REST lo direcciona a la acción *show* del **controlador** *users*.
3. el controlador extrae el id del usuario de la llamada⁷ **GET /users/1**.
4. la acción *show* del controlador se encarga de hacer la llamada al **modelo** “User”, mediante el metodo `find_by_id(1)`.
5. el modelo “User”, inicialmente valida o modifica los datos recibidos.
6. el modelo “User”, mediante el modulo ActiveRecord⁸ extrae la información de la Base de datos.
7. esta información es almacenada en una *instance variable*⁹ `@user`
8. la vista “show.html.erb”¹⁰ incluye la variable `@user` mediante los *ERB output tag* (`<%= %>`)

⁶ ERB (Embedded Ruby)

⁷ request

⁸ Es una implementación del patron Object-Relational Mapping(ORM), que mapea las tablas de la Base de datos relacional en clases, filas en objetos y columnas en atributos de los objetos

⁹ los *instance variable* son variables especiales que empiezan con una “@” y están disponibles en la vista

¹⁰ la extensión **erb** indica que la pagina HTML puede tener contener código Ruby

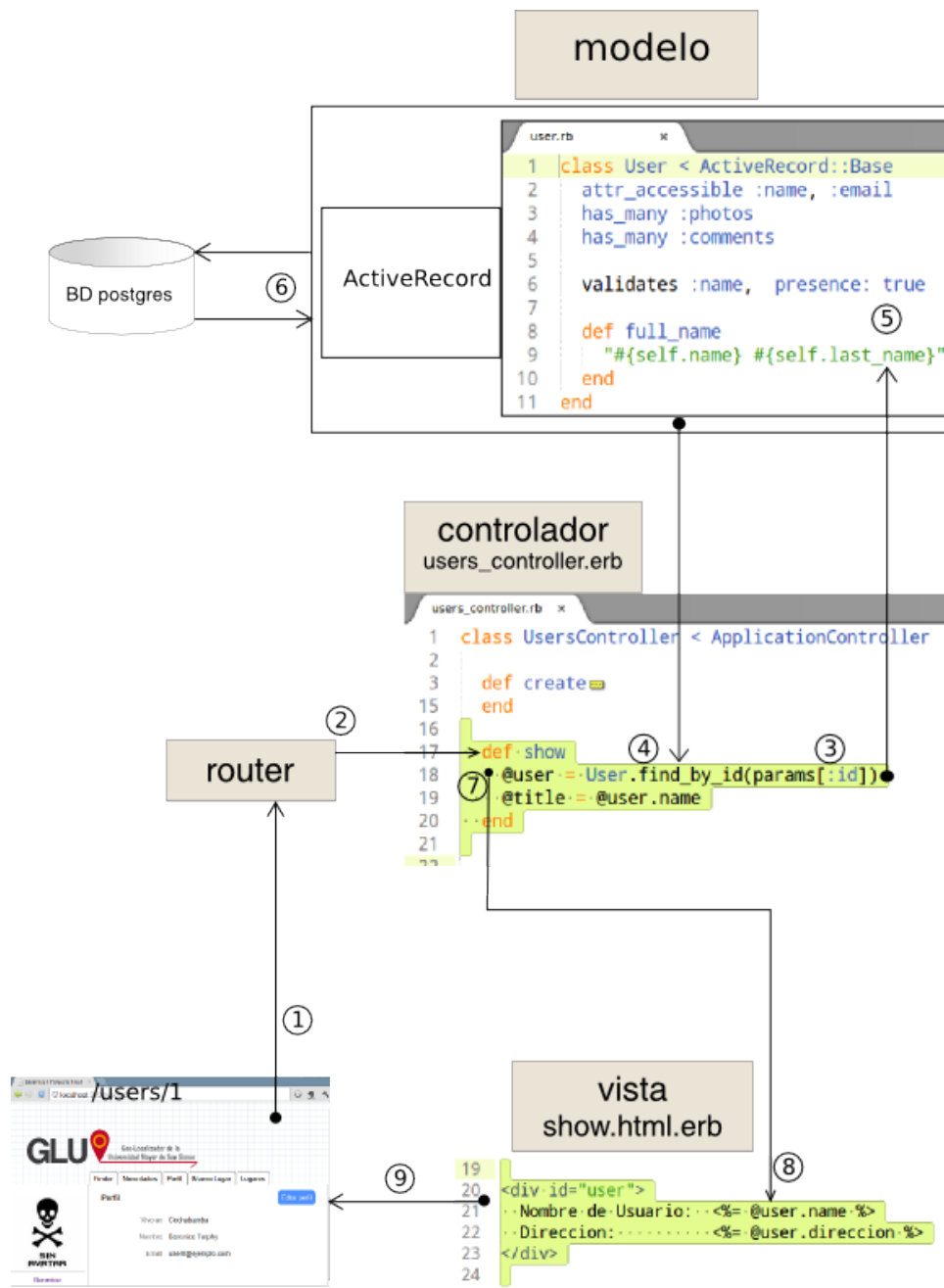


Figura 1.2: Un diagrama detallado del patrón MVC en Rails

9. este template es procesado en el servidor, y renderizado como una pagina HTML que es presentada en el navegador web.

La capacidad de declarar una variable en el controlador y que esta variable esté disponible en la vista, así como toda la configuración que envuelve el comportamiento del patrón MVC ya esta implementada y no es necesario escribir archivos de configuración, El patrón MVC es la base de la aproximación RESTful de Rails, así como de las filosofías *DRY* y *Convención sobre Configuración*, en conclusión se pudo apreciar que al usar esta arquitectura el código se vuelve mas ordenado, con el consiguiente resultado de ser un código mucho mas limpio, legible y entendible.

1.2.4. Mashup

Mashup es una técnica en el desarrollo de aplicaciones web que consiste en combinar datos de diferentes proveedores. Un mashup usa, uno o más servicios y mezcla algunas características de estos servicios, los servicios generalmente son datos que son distribuidos mediante las APIs¹¹.

Con la finalidad de ofrecer estos datos de forma más fácil de entender y usar para el cliente.

Los primeros mashups empezaron como experimentos con los web services que ofrecían las grandes empresas, por ejemplo uno de los primeros mashups mas importantes fue el de housingmaps¹² que combina los datos de Craigslist (listas de casas, departamentos, etc, para rentar, vender) y Google Maps (mapas a nivel global), en una aplicación que ofrece servicios y los posiciona en un mapa. Desde entonces surgieron diferentes tipos y aproximaciones dependiendo del tipo de datos y el servicio que se quiere ofrecer.[9]

Se pueden apreciar diferentes implementaciones de mashups, los mashups orientados al consumo y los mashups orientados a las empresas, básicamente estos mashups de consumo se refieren a aplicaciones en las cuales los datos obtenidos de diferentes fuentes son mezclados y presentados al cliente, y los mashups empresariales o de negocios aparte de las fuentes de datos externas, mezclan sus propios datos en el mashup ofreciendo una mejor experiencia

¹¹Application Programming Interface, es el conjunto de funciones y procedimientos que se ofrece para ser utilizado por otro software

¹²<http://www.housingmaps.com/>

del usuario.

Uno de los componentes básicos de la Web 2.0 son los **datos**, [5] sobre los cuales se puede obtener un producto con valor agregado, generalmente los datos están disponibles para que sean usados, pero dependiendo del tipo de “datos”, obtenerlos y almacenarlos puede llegar a costar tiempo o dinero o ambos, es por eso que los datos son ofrecidos con o sin restricción dependiendo de las políticas de uso que tenga la empresa que ofrece los datos.

El mashup implementado se podría catalogar como un mashup de negocio, ya que se está integrando datos externos (Google Maps API) y datos internos (locaciones dentro del campus de la UMSS), inicialmente se almacenaron varias locaciones pero eventualmente estos datos se incrementarían mediante el uso que se le da por parte de los usuarios.

Para usar el API de Google Maps es necesario importar la librería declarándola en la cabecera del documento HTML, como se ve muestra en la figura 1.3, para visualizar el mapa es necesario declarar un elemento único (se recomienda un div con id de preferencia “map” o similar) dentro del DOM de la página, figura 1.4, de esta forma se logra visualizar el mapa geográfico que ofrece Google, y su manejo es mediante el lenguaje de programación del lado del cliente, *javascript*, figura 1.5.

Para el uso de los datos internos se hizo uso intensivo de la interfaz REST de Rails, ya que se necesitaba desplegar en el mapa los lugares o locaciones de la UMSS, esto se debe a que la interfaz mashup se encuentra en el lado del cliente (navegador Web) y la información recopilada está almacenada en el servidor (Rails), entonces necesitaba extraer los datos del servidor y presentarlos al cliente, este procedimiento se lo puede demostrar con el uso del comando `curl`, y con la información obtenida se puede desplegar la locación en el mapa.

```
$ curl http://localhost:3000/places/23.json

{
  "id": 23,
  "type": "caseta",
  "address": "",
  "lng": -66.1473755998898,
  "lat": -17.3938599682795,
```

```
"name": "informaciones",
"photos": [
  {
    "id": 28,
    "desc": "informaciones",
    "url": "photo/image/28/square_shin.jpg"
  }
]
```

Como se ve en el ejemplo, el servidor responde mandando los datos en formato **json**¹³, que al ser un metodo comun para enviar datos desde el servidor al cliente, es facil de implementar y de manipular, y se genera atraves de una plantilla en la que se puede manipular los datos y presentarlos de la mejor forma posible. Por ejemplo para mostrar los lugares se escribio la siguiente plantilla.

```
# views/places/show.json.rabl

object @place
  attributes :id, :name, :desc, :address

  node(:type) { |place| place.type_place.name }
  node(:lng)  { |place| place.coord_geographic.x }
  node(:lat)  { |place| place.coord_geographic.y }

  child :photos do
    attributes :id, :desc
    node(:url) { |photo| photo.image.url(:square) }
  end
```

De esta forma se pueden ofrecer los datos internos para que otras aplicaciones la puedan usar. Es lo que se denominaria *REST Web Services*

En la aplicación desarrollada se necesitaba el uso de mapas para que el despliegue de información sea de forma más interactiva y de fácil uso, los

¹³Notación de Objetos de JavaScript, es un formato ligero de intercambio de datos y está basado en un subconjunto del Lenguaje de Programación JavaScript [10]

datos geográficos públicos generalmente están disponibles para ser usados pero para su uso se necesitaba que esté representado de forma gráfica en la pantalla de la computadora, este trabajo lo llevan a cabo empresas las cuales generalmente cobran por el uso de los mapas, se puede apreciar que se están obteniendo datos en forma de mapas a través del API de Google Maps, Google permite usar sus datos con restricciones dependiendo de la cantidad de llamadas que se haga a su API por la aplicación que la usa.

El gran beneficio de usar los datos de Google Maps en un mashup es que nos permite desplegar información que de otra forma costaría mucho tiempo y dinero para implementar.

1.3. Conclusión

Los patrones no son entidades separadas, como ya se vio todos los patrones web2.0 se integran y trabajan de manera conjunta, no se podría implementar software que solo implemente algún patrón.

Los patrones empiezan como soluciones a algún problema de diseño y descritas en concepto, gracias a que los problemas y experiencias que tuvieron los programas que lograron pasar de la Web Estática y los que nacieron en la Web2.0 están documentados y analizados, después depende de los desarrolladores el implementarlas de la mejor forma posible.

Rails se diseñó, para que trabaje sobre las últimas tecnologías disponibles, para poder desarrollar software que se pueda catalogar como “*Aplicación Web2.0*”, pero al final que una aplicación obtenga esta etiqueta está dada por múltiples factores, tales que pueden ser, la capacidad de entender la web como un algo que evoluciona con el tiempo, e implementar software que pueda utilizar todos los avances tecnológicos en favor de mejorar la experiencia del usuario, siempre apoyados por la interacción que supone el intercambio y flujo de ideas de la comunidad de usuarios que usan la aplicación.

```

3 <head>
4 <title><%= full_title(@title) %></title>
5 <%= include_gon %>
6 <%= stylesheet_link_tag "application", :media => "all" %>
7 <%= javascript_include_tag "http://maps.google.com/maps/api/js?sensor=false" %>
8 <%= javascript_include_tag "application" %>
9 <%= csrf_meta_tags %>
10 </head>

```

Figura 1.3: Se importa el API de Google Maps declarando en <head> del documento

```

10 .....
17 .....<article id="finder_place" class="perfil">
18 .....<div id="map"></div>
19 .....

```

Figura 1.4: El mapa se visualiza dentro de la etiqueta div con id map

```

11 .....
12 .....
13 .....init: function( opt_ ) {
14 .....var opt = opt_ || {};
15 .....var options = {
16 .....zoom: opt.zoom || 17,
17 .....center: new google.maps.LatLng( (opt.lat || -17.3937285), (opt.lng || -)
18 .....mapTypeId: google.maps.MapTypeId.ROADMAP,
19 .....disableDefaultUI: true,
20 .....navigationControl: true
21 .....}
22 .....
23 .....this.map = new google.maps.Map( $(opt.selector || "#map").get(0), options );
24 .....
25 .....},
26 .....
27 .....

```

Figura 1.5: Se captura el div map del documento y se inicializa el mapa mediante el lenguaje javascript

Bibliografía

- [1] Sam Ruby, Dave Thomas, David Heinemeier Hansson, *Agile Web Development with Rails, Fourth Edition*
- [2] <http://trends.builtwith.com/topsites/Ruby-on-Rails>
- [3] tumblr.yasulab.jp/post/10271634919/5-question-interview-with-twitter-developer-alex-payne
- [4] http://www.artima.com/scalazine/articles/twitter_on_scala.html
- [5] <http://oreilly.com/web2/archive/what-is-web-20.html>
- [6] <http://radar.oreilly.com/2006/12/web-20-compact-definition-tryi.html>
- [7] <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [8] <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>
- [9] <http://msdn.microsoft.com/en-us/architecture/bb906060.aspx>
- [10] <http://www.json.org/json-es.html>