



Introducción a Google Maps API



Indice

Introducción	4
Para quién es este libro	4
Acerca del autor	5
Capítulo 1: Introducción	6
Introducción a Google Maps	6
Recomendaciones antes de empezar con Google Maps	6
¿ Cómo funciona está cosa llamada Google Maps ?	8
Fundamentos Básicos sobre coordenadas	8
Cómo leer la documentación de la API	9
Capítulo 2: Conceptos básicos	10
Creando tu primer mapa	10
Jugando con las opciones de mapa	14
Deshabilitar la Interfaz por completo	14
Configurar como mostrar los tipos de mapa	14
Control de navegación	17
keyboard Shortcuts	18
disableDoubleClickZoom	18
draggable	19
scrollwheel	19
streetViewControl	19
Crear controles personalizados	20
Metodos de nuestro objeto mapa.	22
setOptions()	22
getZoom() y setZoom()	23
getCenter() y setCenter()	23
getMapTypeId() y setMapTypeId()	24
Capítulo 3: Overlays	25
Overlay : Marker.	25
Cambiando el icono del marker	27
Agregar una Infowindow	29
Crear eventos	30
Crear evento en el marker	32
Infowindow contenido HTML	33
Cambiado el icono del maker con eventos	34
Maker arrastrable en el mapa y animado	35
Overlay : Polylines.	35
Overlay : Polygons	39
Overlay : Circle	42
Overlay : Rectangle	43

Algunos eventos a tener en cuenta de los overlay	45
Overlay : Ground	45
Drawing Library	46
Capítulo 4 : Tareas Comunes en Google Maps	50
Trabajando con múltiples overlays	50
Demasiados Maker en el mapa	53
MarkerClusterer	54
Marker Spiderfier	56
MarkerManager	57
Usando MarkerManager como Cluster	60
Maplabel	63
Localizar al usuario	65
Places Autocomplete	67
Usando Panoramio	70
Mostrando el clima	72
Reverse Geocoding	74
Static Maps API	79
Capítulo 5 : Guardar y cargar tu overlays	81
Serializar Marker	82
Serializar rectangle	86
Serializar polyline	87
Serializar polygon	90
Serializar circle	91
Serializar el mapa	92
Deserializar Marker	94
Deserializar rectangle	97
Deserializar Polyline	99
Deserializar Polygon	101
Deserializar Circle	102
Actualizar Map	103
Ejemplos Adicionales	106
Obtener puntos LatLng contenidos dentro de un bounds	106
Cálculo de áreas y distancias	107

Introducción

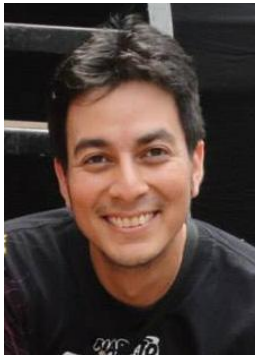
Les presento mi primer libro, que nació a raíz de la poca documentación que existe sobre cómo usar los mapas de google. Quiero compartir mi experiencia de más 3 años de tiempo completo, que he dedicado al desarrollo de mapas con la API de Google Maps. Esto, con el objetivo de que el lector sea capaz de asimilar los conceptos básicos y técnicas necesarias, para crear sus propios mapas en sus proyectos web. En el libro haremos uso de la versión 3.13 que está disponible actualmente, durante el desarrollo del libro.

También, quiero agradecer a mi familia por brindarme su apoyo en el proceso de este libro, especialmente a mi madre que siempre me estuvo apoyando en todo.

Para quién es este libro

Este libro está orientado principalmente a desarrolladores/diseñadores web, quienes desean aprender como crear sus propios mapas para sus proyectos web. No te preocupes, si no estas entre estas áreas; el objetivo del libro es ayudarte a comprender desde lo más básico de la API de Google Maps. Aunque tener cierta experiencia y conocimientos básicos sobre programación web serán una gran ventaja.

Acerca del autor



Mi nombre es Saúl Burgos Dávila, soy ingeniero en computación y trabajo como desarrollador de aplicaciones web, desde hace 4 años a tiempo completo. En mi actual empleo en [OOQIA](#) desarrollo aplicaciones web que incorporan soluciones de mapas y la API de Google Maps es mi principal herramienta de trabajo. Me considero a mi mismo como un entusiasta de Google Maps, siendo una de mis herramientas favoritas de desarrollo. La cual complemento con sólidos conocimientos de Javascript y el uso de tecnologías libres, buscando siempre cumplir con los estándares web de la actualidad.

Enfocado principalmente en el desarrollo front-end, estoy en la constante exploración de nuevas tecnologías libres, que me permitan crear aplicaciones web robustas y sólidas. Herramientas que me permitan abstraerme de los complicados tecnicismos de los lenguajes de programación, para enfocarme en una experiencia de usuario limpia y simple.

Capítulo 1: Introducción

Introducción a Google Maps

En la actualidad, gran parte de las aplicaciones web requieren el uso de tecnología de mapas, para mostrar direcciones, ubicar localidades, mostrar rutas, ubicaciones en tiempo real, etc... Cualquier ubicación puede ser localizada en el mapa que nosotros queramos mostrar al usuario en nuestro sitio.

Google Maps: Nos ofrece todo el poder de su API para que nosotros podamos crear mapas de manera fácil, confiable y sobre todo multiplataforma. Nuestro mapa puede ser visto desde cualquier dispositivo sea tablet, smartphone o desktop, gracias a los nuevos estándares HTML 5, Javascript, CSS3.

En este libro nos enfocaremos en Google Maps API con Javascript.

Recomendaciones antes de empezar con Google Maps

Estas recomendaciones son basadas en mi propia experiencia y viendo a compañeros de trabajo que iniciaron por primera vez con Google Maps, después de leer estas recomendaciones quizás pienses que necesitas aprender muchas cosas antes de empezar Google Maps. Pero déjame decirte que son sólo recomendaciones de mi parte, que vale la pena saber antes de iniciar con Google Maps. Pero, es posible aprender a usar Google Maps sin saber algunas de ellas.

Javascript: Ok esto es más que obvio, ya que en este libro veremos la Google Maps API basado en javascript, me gustaría hacer énfasis en algunos puntos que considero importantes que debes manejar con javascript.

1. [Arreglos](#): Debes saber como trabajar con arreglos, esto es importante porque la mayoría del tiempo tendrás que iterar sobre los elementos, agregar elementos, eliminar elementos y modificar elementos de un arreglo.
2. Programación orientada a objeto: Javascript es un lenguaje orientado objeto, por esta razón debes de saber que es un método, una propiedad, que es el tipo de datos objeto, de como instanciar una clase y el uso de [prototype](#).
3. [JSON](#) (JavaScript Object Notation): Necesitas saber la importancia de este formato de como usarlo, crearlo y manipularlo. Ya que a lo largo de este libro, lo usaremos en nuestros ejemplos todo el tiempo.

4. [AJAX](#) (Asynchronous JavaScript And XML) Es importante saber el concepto de AJAX, de como funciona y para qué sirve.
5. Saber leer una API: Esta una de las partes que me tocará explicar un poco, el primer problema que encontré cuando comencé con Google Maps, es que no sabía cómo leer la documentación de una API y el mismo problema lo visto en otras personas continuamente. Esto puede ocasionar que el desarrollador se frustre muy fácil desde el inicio. Un gran problema a veces con las API es la pobre y mal organizada documentación en los sitios webs que terminan confundiendo más al desarrollador. En este punto tengo que hacer notar, que la documentación de la API de Google Maps es una de las mejores organizadas que he visto. A largo del libro, muchas veces haré referencia a secciones de la documentación de Google Maps, mi objetivo con esto es que aprendas a leer la documentación de la API, ya que considero que este es un problema muy común entre las personas que inician con esta herramienta.
6. [Jquery](#): Este es uno de lo frameworks más populares de Javascript, saber como usarlo es una enorme ventaja.
7. En los ejemplos de este libro estamos usando variables globales, para almacenar valores. Aunque en proyectos pequeños esto no tiene importancia es considerado una mala práctica de programación en javascript. En un proyecto de considerable tamaño esto puede ser un caos, por esa razón sugiero que aprendes un poco de ["Namespacing"](#)

HTML, CSS: Como todo buen desarrollador tienes que saber de HTML, CSS que vendría siendo algo básico para construir aplicaciones web.

Debugger Javascript: En toda aplicación siempre habrá errores en nuestro código, ya sea errores de sintaxis ó errores lógicos. Saber cómo descubrirlos es una parte vital para salir adelante, por eso recomiendo que sepas lo mínimo para debugear tu código javascript. En este libro estaremos usando el modo consola de Google Chrome, pero tú puedes usar el navegador que quieras en su modo debugger. Si nunca haz usado el modo consola de un navegador te recomiendo este [link](#) donde puedes aprender a usar modo consola de Google Chrome.

Editor de texto: Como editor de texto para los archivos yo recomiendo que uses [notepad++](#), pero puedes usar con el que estes mas comodo.

¿ Cómo funciona está cosa llamada Google Maps ?

Recuerdo la primera vez que vi un mapa de Google Maps, pense que era los más increíble que había visto con respecto a mapas, pero la mayoría de las cosas en este mundo tiene su explicación.

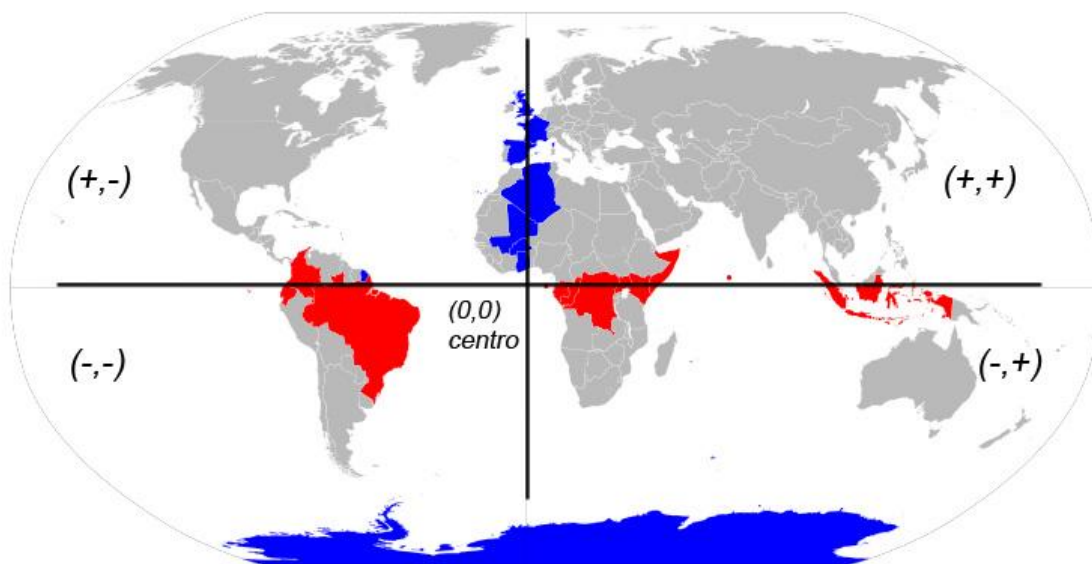
Google Maps: Es un conjunto de códigos: HTML, javascript y CSS. Trabajando juntos con cada una de las secciones del mapa, puedes notar que son cargadas cuando te mueves o haces zoom en el mapa. Esto ocurre mediante el métodos AJAX y después son insertadas en un elemento <DIV> basado en coordenadas del mapa.

La API de Google Maps es un conjunto de clases con métodos y propiedades que pueden ser usadas para manipular el mapa como tu quieras.

Fundamentos Básicos sobre coordenadas

Para poder trabajar con la API necesitamos primero conocer el sistema de coordenadas que usa Google Maps. La cual es [WGS84](#) el mismo usado en los sistema GPS, estas coordenadas son expresadas en Latitud que representa el valor "Y" y Longitud que representa "X".

La latitud se mide de sur a norte y longitud se mide de oeste a este. En el ecuador la latitud es 0, esto nos indica, que todo lo que esta arriba del ecuador será un valor positivo y lo que esta abajo de él, será un valor negativo. Lo mismo pasa con longitud, pero está localizado en el [Meridiano de Greenwich](#) lo que se encuentra al este será un valor positivo y al oeste un valor negativo. En la siguiente imagen podemos verlo claramente.



Las coordenadas en Google Maps siguen en este orden: (Latitud, Longitud) esto es muy importante de recordar, para crear nuestros puntos en el mapa. Los valores son en decimales separados por coma.

Ejemplo : 38.97 , -1.23.

Cómo leer la documentación de la API

La documentación de Google Maps por lo general sigue la siguiente estructura para los métodos:

```
Map(mapDiv:Node, opts?:MapOptions)
```

Lo primero es el nombre del método, después entre los paréntesis se encuentran los parámetros del método. En los parámetros, el nombre que va antes de los 2 puntos “:” es un nombre descriptivo para el parámetro solamente y lo que está después de los 2 puntos “:” es el tipo de variable el cual debe ser el parámetro. Si existe un signo de interrogación en el nombre descriptivo del parámetro, quiere decir que es un parámetro opcional.

Los tipos de datos primitivos javascript son descritos por ellos mismos por ejemplo : string, boolean , number. Pero los arreglos y MVCArray son descritos de una manera distinta.

Array.<MapTypeId> : Esto quiere decir que el valor de esta propiedad es un arreglo y que cada elemento del arreglo es de tipo “[MapTypeId](#)”, que es un objeto específico de Google Maps.

MVCArray.<MapType> : Esto quiere decir que el valor de esta propiedad es un MVCArray que contiene objetos del tipo “[MapType](#)”.

Otro dato importante es que todas las clases en Google Maps se encuentran dentro del namespace “google.maps.”. Esto quiere decir que si quieres usar algún método, clase o constante de la API, tienes que escribir siempre primero “google.maps.” seguido por lo que quieras usar. Por ejemplo aquí estamos usando la clase [LatLng](#):

```
new google.maps.LatLng();
```

Capítulo 2: Conceptos básicos

Creando tu primer mapa

Lo primero que necesitamos para crear nuestro primer mapa, según la documentación de Google Maps es obtener una “API Key” para poder acceder a la API. Aunque es posible, usar la API sin una API KEY es recomendable que uses tu propia API KEY en tus proyectos.

Por motivos de práctica, en este libro no usaremos una API KEY. Pero recomiendo que consigas una, cómo lo recomienda google en este [Link](#).

Antes de crear nuestro mapa necesitamos crear una página web sencilla, que incluya el CSS y Javascript necesarios. En esta práctica usaremos los archivos de práctica de la carpeta número “1”, donde encontrarás los archivos necesarios para ver el ejemplo funcionando. A continuación explicaré el código.

En nuestra carpeta de ejemplo tenemos 3 archivos llamados: index.html, googlemaps.js, style.css, estos son los archivos que componen nuestra aplicación:

index.html: Aquí tenemos el [DOM](#) de nuestra aplicación, si abres el archivo podrás ver, que es bastante simple, el cual tiene un DOCTYPE de HTML 5. En la sección <head> tenemos los archivos CSS y Javascript que usa la aplicación, también está cargado el framework javascript [Jquery](#), que es usado a lo largo de este libro.

Esta es la parte más importante:

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?sensor=false"></script>
```

Con este script cargamos la API de Google Maps, para poderla usar en nuestro proyecto. Para nuestros ejemplos no la usaremos. En la URL existe un parámetro llamado: “sensor” y tiene el valor “false” esto es para indicarle a nuestro mapa, si nuestra aplicación usa algún dispositivo tipo GPS para determinar nuestra posición. Este parámetro es obligatorio y debe de ser “true” o “false”, en nuestro caso será “false”.

También hemos cargado en la etiqueta <head> el script “googlemaps.js” que es donde está nuestro código javascript para cargar el mapa. En la etiqueta <body> tenemos un elemento DIV con el id “map” que será nuestro contenedor, para nuestro mapa ya que la API nos pide un contenedor donde insertar el mapa.

Nuestro elemento DIV que contendrá nuestro mapa, necesita una altura y un ancho para que el mapa sea mostrado, por eso en el archivo CSS llamado: “style.css” hemos definido su altura y ancho al 100% para que ocupe toda la pantalla.

En el archivo “googlemaps.js” estamos usando el evento “[ready](#)” de jquery, para cargar el mapa. Este evento nos permite ejecutar nuestro código cuando el DOM está listo. A continuación voy a explicar el código javascript del archivo “googlemaps.js”.

Lo primero que necesitamos hacer es definir las opciones para nuestro mapa, las cuales serán pasadas como parámetro; de otro modo el mapa no aparecerá. Para crear las opciones de nuestro mapa, primero tenemos que leer cuáles son las opciones disponibles a usar. Esto lo podemos ver en la documentación de la API en este [Link](#).

El uso de GOOGLE.MAPS.NAMESPACE: *Al leer la documentación puedes notar que todos los métodos y clases comienzan con “google.maps.” esto es una técnica javascript llamada: “namespace” que es usada para evitar colisiones de nombre de métodos y variables con otros scripts en tu aplicación, es comúnmente usada cuando tienes muchos scripts en tu aplicación cargados.*

Como puedes ver hay muchas opciones que podemos usar para nuestro mapa, puedes leerlas todas para que tengas una idea de lo que puedes hacer y que no. Las opciones del mapa tienen que pasarse como parámetro en un objeto javascript, y cada opción del mapa tiene que ser una propiedad de ese objeto. Esta parte es muy importante que la recuerdes porque a lo largo de toda la documentación, siempre se especifica que tipo de objeto tienes que usar o crear con determinada clase.

En la lista de opciones del mapa, puedes ver en la tabla las columnas tienen los nombre de: Properties, Type y Description. “Type” nos indica que tipo objeto tenemos que usar en esa propiedad. Esto es muy importante.

Para poder crear un mapa necesitamos como mínimo 3 opciones: “center”, “zoom”, “mapTypeId”. Si consultas la documentación, podrás encontrar los posibles valores de estas propiedades.

Para nuestro ejemplo, hemos creado las opciones del mapa con un objeto simple de esta manera :

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

Este objeto tiene 3 propiedades que son las opciones de nuestro mapa. Ahora las propiedades **center** y **mapTypeId** tienen definidos otros objetos pero ¿ porque esto es así ? Bueno, si consultamos nuevamente la documentación y vemos el tipo de dato que requiere estas propiedades encontraremos esto:

```
center : LatLng
mapTypeId : MapTypeId
zoom : number
```

Nos dice que la propiedad center (lugar donde se centrará el mapa) tiene que ser de tipo “LatLng”, pero ¿Qué es un LatLng ? Bueno, en la documentación podemos hacer click en el nombre y nos llevará a la sección de la API de la clase [LatLng](#) y los parámetros que se necesitan para crear el objeto.

Entonces lo que necesitamos es crear un objeto LatLng para asignar a nuestra propiedad **center** pero ¿ como se crea ?. Pues instanciando la clase [google.maps.LatLng](#) y pasandole los parametros requeridos que según la documentación en este caso dice que son : **lat:number**, **lng:number** , siendo el tercero opcional.

```
LatLng(lat:number, lng:number, noWrap?:boolean)
```

Lat y Lng son la latitud y longitud que definen una position en el mapa usando el sistema [WGS84](#). Para crear posiciones siempre tenemos que saber estos datos de antemano. Ahora sabiendo la definición de esta clase podemos crear nuestro objeto de esta manera :

```
center: new google.maps.LatLng(-34.397, 150.644)
```

De esta manera creamos nuestro objeto LatLng, aquí estoy usando la palabra “[new](#)” que es usada en javascript para crear instancias de objetos, lo que significa que crea una instancia del objeto LatLng que define una posición en el mapa, recuerda que en google Maps un punto geográfico en el mapa es definido por su latitud y longitud como lo vimos anteriormente.

Ahora pasamos a la propiedad mapTypeId , de nuevo si consultamos la documentación de [mapTypeId](#) ,podremos ver que este objeto son constantes definidas y según la documentación

de Google Maps son las siguientes :

1. ROADMAP : Muestra los mosaicos normales en 2D predeterminados de Google Maps.
2. SATELLITE : Muestra imágenes de satélite.
3. HYBRID : Muestra una mezcla de mosaicos fotográficos y una capa de mosaicos para los elementos del mapa más destacados (carreteras, nombres de ciudades, etc.)
4. TERRAIN : Muestra mosaicos de relieve físico para indicar las elevaciones del terreno y las fuentes de agua (montañas, ríos, etc.).

Simplemente tenemos que escoger cual de estas opciones queremos usar , escribiéndola como propiedad de la clase “[google.maps.MapTypeId](#)”, en este caso usaremos :

`google.maps.MapTypeId.ROADMAP`

Después tenemos Zoom que simplemente según la documentación es un número comprendido de 0 a 19 que determina el nivel de zoom en el mapa . Ok con esto terminamos con nuestras opciones ahora pasamos a la siguiente línea de código.

```
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
```

Para crear nuestro mapa necesitamos crear un objeto mapa instanciando la clase [google.maps.Map](#) y según la documentación tenemos que pasar como parametro el contenedor donde aparecerá nuestro mapa y las opciones del mapa:

```
Map(mapDiv:Node, opts?:MapOptions)
```

En este caso estamos pasando nuestro elemento DIV con el id="map" y las opciones que creamos anteriormente. Bueno y con solo esas líneas de código bastará para crear nuestro primer mapa en nuestra aplicación de prueba.

En esta primera parte fui muy cuidadoso de ir paso a paso de como debes leer la documentación de la API espero que haber sido claro en qué partes de la documentación debemos poner atención a la hora de crear nuestros objetos y saber que son , apartir de aqui no volveré a ser tan descriptivo solo mencionare el tipo de objeto que necesitamos usar en nuestro código y consultando la documentación deberías deberías de saber que necesitas para crearlo y sus parámetros necesarios.

Jugando con las opciones de mapa

En la sección anterior vimos como crear un simple mapa , ahora vamos a jugar con algunas de las opciones que nos ofrece la API de Google Maps. En este libro veremos algunas de las opciones más comunes en el uso de mapas, pero aliento al lector a explorar las opciones restantes por tu propia cuenta.

Deshabilitar la Interfaz por completo

`disableDefaultUI`: Pasando esta propiedad a `true` podemos deshabilitar todo control visual en nuestro mapa.

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  disableDefaultUI: true
};
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
```

Configurar como mostrar los tipos de mapa

Con la propiedad llamada “`mapTypeControlOptions`” podemos controlar los tipos de mapa a seleccionar y su position en la interfaz. Esta propiedad toma como valor un objeto del tipo: [MapTypeControlOptions](#) y de acuerdo a la documentación este objeto tiene 3 parámetros : **mapTypeIds** , **position** , **style**. Al usar esta propiedad debemos poner `mapTypeControl` del objeto mapa a `true`. A continuación veremos algunos ejemplos.

Style : Esta propiedad determina la apariencia del control y su valor tiene que ser del tipo [MapTypeControlStyle](#), las cuales son constantes definidas en la clase. Para poder usar estas constantes solo tenemos que usar el nombre la clase “`google.maps.MapTypeControlStyle.`” y al final el nombre usar la constante que usaremos. por ejemplo:

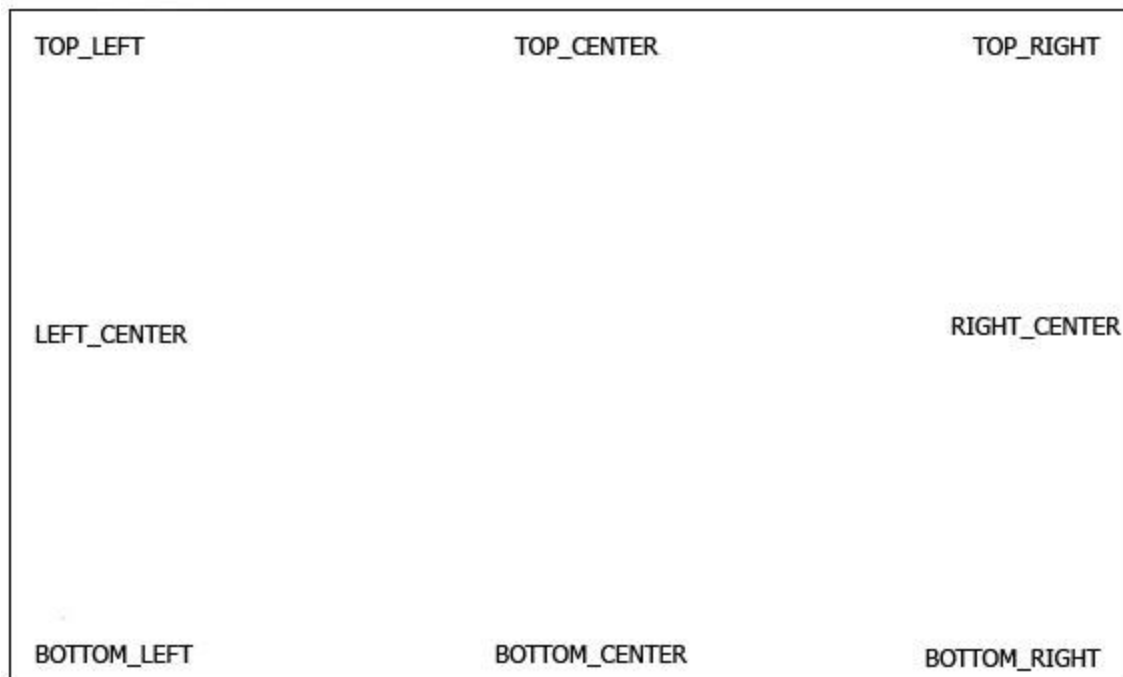
```

var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  mapTypeControl: true,
  mapTypeControlOptions: {
    style: google.maps.MapTypeControlStyle.DROPDOWN_MENU
  }
};
var map = new google.maps.Map(document.getElementById("map"), mapOptions);

```

Puedes probar los diferentes valores de las constantes y ver los resultados en el mapa.

Position : Por defecto el control de tipos de mapa siempre esta arriba a la derecha, pero podemos cambiar la position con esta propiedad, pasando como valor una de las constantes de [ControlPosition](#) . Alguno de los valores que podemos usar son :



Por ejemplo podemos usar :

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  mapTypeControl: true,
  mapTypeControlOptions: {
    position: google.maps.ControlPosition.TOP_CENTER
  }
};
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
```

mapTypeIds : Esta propiedad te permite elegir que tipos de mapa podrá el usuario elegir en la interfaz, el valor que necesitamos pasar a esta propiedad será un arreglo donde cada elemento será un tipo de datos [MapTypeId](#) . Por ejemplo el siguiente código solo mostrará 2 tipos de mapas al usuario:

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  mapTypeControl: true,
  mapTypeControlOptions: {
    mapTypeIds: [
      google.maps.MapTypeId.ROADMAP,
      google.maps.MapTypeId.SATELLITE
    ]
  }
};
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
```

Ahora vamos a poner todos estos códigos juntos:

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  mapTypeControl: true,
  mapTypeControlOptions: {
    style: google.maps.MapTypeControlStyle.DROPDOWN_MENU,
    position: google.maps.ControlPosition.TOP_CENTER ,
    mapTypeIds: [
      google.maps.MapTypeId.ROADMAP,
      google.maps.MapTypeId.SATELLITE
    ]
  }
};
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
```

Control de navegación

Los controles de navegación por defecto están situados al lado izquierdo del mapa. Son para controlar el zoom, pan y el pegman (hombrecito color naranja) de streetview.



Podemos mostrar ó ocultar estos controles con las siguientes propiedades : panControl, streetViewControl y zoomControl. Todas estas propiedades reciben un valor booleano para ser mostradas en la interfaz. Por ejemplo :

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  panControl : false ,
  streetViewControl : false ,
  zoomControl : false
};
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
```

keyboard Shortcuts

Esta propiedad habilita / deshabilita el uso del teclado para controlar nuestro mapa por ejemplo las teclas + / - para controlar el zoom del mapa.

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  keyboardShortcuts : false
};
```

disableDoubleClickZoom

Esta propiedad controla el doble click sobre el mapa que ejecuta un zoom sobre el mapa.

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  disableDoubleClickZoom : true
};
```

draggable

Esta propiedad controla si podemos arrastrar el mapa con nuestro mouse.

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  draggable : false
};
```

scrollwheel

Esta propiedad controla el uso de scroll para hacer zoom en el mapa.

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  scrollwheel : false
};
```

streetViewControl

Esta propiedad esconde o muestra los controles para el streetview comúnmente conocido como “pegman”.

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP ,
  streetViewControl : false
};
```

Las propiedades panControl, streetViewControl y zoomControl. También tiene sus propias opciones donde podemos controlar la posición en donde queramos que aparezcan usando los siguientes objetos: [panControlOptions](#) , [streetViewControlOptions](#) , [zoomControlOptions](#) de la misma manera como lo hicimos anteriormente con el control de tipos de mapa, usando la clase [google.maps.ControlPosition](#) .

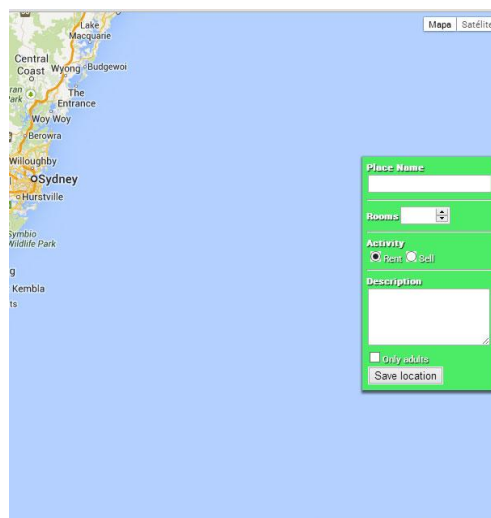
Hasta aquí hemos visto algunas de las opciones que podemos configurar para nuestro mapa , invito al lector a explorar las restantes de la API.

Crear controles personalizados

Los controles por defecto de Google Maps cumplen con su propósito básico, pero habrá ocasiones en el que queremos tener nuestros propios controles con un estilo visual distinto, acorde con el diseño de nuestra página web. Para lograr esto necesitamos crear controles personalizados e insertarlos en el mapa.

El proceso de crear controles personalizados es el mismo que maquetar un formulario HTML normal, creando los controles con HTML, aplicar estilo con CSS y luego insertarlos dentro del mapa, los cuales flotaran dentro del viewport del mapa. Los controles personalizados tienen que estar dentro de elemento `<DIV>` antes de ser insertados. En el siguiente ejemplo hemos creado un pequeño formulario, el cual nos dará una idea de lo que podemos hacer.

Podemos abrir el archivo de ejemplo de la carpeta número “1-2”. Si ejecutamos el archivo “index.html” podremos ver nuestro mapa cargado y del lado derecho veremos un panel de color verde con un pequeño formulario.



Para agregar este formulario al mapa primero tiene que ser creado con HTML y tener sus estilos definidos en CSS antes de ser insertado. Algo que tienes que tener en cuenta es que la API de Google Maps tiene sus propios estilos CSS para la UI del mapa, por lo que tienes que estar alerta si algunos de tus estilos son sobrescritos por la API. En el archivo “style.css” de nuestro ejemplo podrás ver los estilos usados para el formulario.

Anteriormente dije que el formulario tiene que ser creado antes de ser insertado en el mapa ojo aquí, solamente creado y no insertado en el DOM de la página web ya que de la inserción se hará cargo Google Maps por nosotros luego.

Ahora podemos abrir el archivo “googlemaps.js” donde podemos ver que existe una variable llamada “htmlForm” a la cual le estamos asignando nuestro código HTML del formulario como una cadena de texto.

```
var htmlForm = '<div class="customControl">' +
    '<div>' +
        '<b>Place Name</b>' +
        '<input type="text" />' +
    '</div>' +
    '<hr>' +
    '<div>' +
        '<b>Rooms</b>' +
        '<input type="number" name="quantity" min="1" max="10">' +
    '</div>' +
    '<hr>' +
    '<div>' +
        '<b>Activity</b><br>' +
        '<input type="radio" name="activity" value="male"
checked>Rent' +
        '<input type="radio" name="activity" value="female">Sell' +
    '</div>' +
    '<hr>' +
    '<div>' +
        '<b>Description</b><br>' +
        '<textarea rows="4" cols="16"></textarea>' +
    '</div>' +
    '<input type="checkbox" name="call" value="money">Only adults' +
    '<button id="saveForm">Save location</button>' +
    '</div>';
```

El contenido de “htmlForm” es una cadena de texto que será parseada usando JQuery para crear los elementos HTML con las siguientes lineas de código:

```
var controlDiv = document.createElement('div');
controlDiv.className = 'container';
$(controlDiv).append(htmlForm);
```

Aquí lo primero que hacemos es crear el elemento DIV que contendrá nuestro formulario por medio de javascript y usando el método “[append](#)” de JQuery agregamos el formulario al elemento DIV creado. En este ejemplo hemos creado nuestro formulario parseando una cadena de texto HTML. Pero si lo prefieres puedes crear los elementos con puro javascript para luego agregarlos al DIV contenedor. Ahora podemos agregar nuestro formulario al mapa:

```
var map = new google.maps.Map(document.getElementById("map"), mapOptions);
map.controls[google.maps.ControlPosition.RIGHT_CENTER].push(controlDiv);
```

Los controles personalizados son colocados usando la propiedad “controls” del objeto mapa , esta propiedad contiene un arreglo de posiciones de “[google.maps.ControlPosition](#)” , en las cuales podemos insertar nuestro formulario, solo tenemos que agregar nuestro elemento DIV al arreglo usando el método “[push](#)” de javascript en la posición deseada definida en la API y Google Maps automáticamente insertará nuestro formulario. Y si removemos nuestro elemento del arreglo será eliminado del mapa también. Para los eventos de los botones puedes usar Javascript de modo normal.

Metodos de nuestro objeto mapa.

Hasta el momento hemos creado nuestro mapa pasando como parámetro las opciones en una variable llamada “mapOptions”, donde cada propiedad ha sido previamente configurada. Pero también tenemos a nuestra disposición métodos muy útiles del objeto mapa que podemos usar para cambiar las opciones en cualquier momento durante la ejecución de nuestra aplicación.

La API de Google Maps ofrece una serie de [métodos](#) para cambiar el comportamiento de nuestro mapa cuando nosotros queramos. A continuación veremos algunos métodos de los más comunes del objeto mapa.

En esta sección usaremos los archivos de ejemplo que están en la carpeta “2” si abres el index.html podrás ver el mapa a la derecha y un botón al lado izquierdo llamado “Click me”. Este botón tiene un evento click de JQuery donde ejecutaremos los códigos de ejemplo que veremos a continuación.

Si abres el archivo “googlemaps.js” debes de notar que al inicio del archivo he definido Fuera del evento ready de JQuery.

```
var map;
```

Esto es para hacer la variable “map” global y poder acceder a ella desde cualquier lugar de mi aplicación y usar sus métodos en cualquier momento.

Por esta razón debes copiar dentro del evento click del botón, cada fragmento de código que quieras ver en acción de los siguientes ejemplos que veremos.

setOptions()

Este método es usado para cambiar o asignar nuevas opciones en nuestro mapa . Como se puede ver en la documentación recibe como parámetro un objeto con las opciones a configurar.

Por ejemplo : vamos a cambiar el zoom y el tipo de mapa.

```
$('#actionButton').on('click',function() {
    var newOptions = {
        zoom: 3,
        mapTypeId: google.maps.MapTypeId.SATELLITE
    };
    map.setOptions(newOptions);
});
```

Ahora si damos click en el botón “click me” veremos como el zoom y el tipo de mapa son cambiados. Este método puede ser usado para cambiar cualquiera de las [opciones](#) del objeto mapa en cualquier momento.

getZoom() y setZoom()

Como se puede leer los metodos de Google Maps son bien descriptivos por si solos. Por ejemplo: getZoom y setZoom().

getZoom() : obtiene el zoom actual del mapa.

setZoom() : asigna un zoom específico.

setZoom() segun la documentacion recibe como parámetro un número que vendría siendo el zoom deseado

Ejemplo:

```
$('#actionButton').on('click',function() {
    alert(map.getZoom());
    map.setZoom(13);
});
```

Este ejemplo mostrará en el alert el zoom del mapa que tiene en ese momento y después lo cambiara a 13.

getCenter() y setCenter()

Siempre es importante saber como centrar el mapa en cualquier momento con los siguientes métodos puedes lograrlo.

getCenter() : Obtiene el centro del mapa y devuelve un objeto latLng.

setCenter() : Asigna un nuevo centro al mapa, recibe como parámetro un objeto latLng.

Ejemplo:

```
$('#actionButton').on('click',function(){
    alert(map.getCenter());
    var newCenter = new google.maps.LatLng(-33.728, 151.049);
    map.setCenter(newCenter);
});
```

Si hacemos click en el botón veremos como nuestro alert muestra el centro actual del mapa , para después asignar un nuevo centro , automaticamente Google Maps mueve nuestro visor a ese punto. Hay que notar que para asignar el nuevo centro primero tuve que crear un objeto latLng para poder pasarlo como parámetro al método setCenter() .

getMapTypeId() y setMapTypeId()

Con estos métodos podemos saber cual es el tipo de mapa actualmente y asignar uno nuevo.

Ejemplo:

```
$('#actionButton').on('click',function(){
    alert(map.getMapTypeId());
    map.setMapTypeId(google.maps.MapTypeId.SATELLITE);
});
```

Ok hasta aquí hemos visto los métodos básicos para manipular nuestro mapa, pero como puedes ver en la [documentación de la API](#) existen muchos más que podemos usar. Mientras avancemos en el libro usaremos algunos de estos métodos en los ejemplos siguientes .

Creo que una aclaración válida sobre la creación de un objeto latLng, es cuántos decimales usar para crear nuestro objeto. Según este [link](#) para los usos comunes esta bien usar 4 o 3 decimales , recuerda que el máximo que puedes usar es de 5 a 6 decimales.

Capítulo 3: Overlays

En este capítulo vamos a ver una serie de objetos llamados "Overlay" tome la decisión de llamarlos por su nombre en ingles, porque no me gusta la traducción a español que vendría siendo algo como "Superposiciones". Para ser sincero no me gusta nada como suena, así que de ahora en adelante en el libro los llamaremos "overlays"

¿ Qué son los overlays ? La mejor respuesta a esta pregunta, es que nos da google en su API :

"Los overlays son objetos del mapa que están vinculados a coordenadas de latitud y longitud, por lo que se mueven al arrastrar el mapa o aplicar el zoom sobre él. Los overlays son los objetos que 'añades' al mapa para designar puntos, líneas, áreas o grupos de objetos."

En resumen los overlays son objetos que podemos incluir en el mapa, para localizar puntos de interés geográficamente. Después de todo para eso es que sirven un mapa no?

A continuación veremos los distintos overlays que podemos usar.

Overlay : Marker.

El overlay [marker](#) es el objeto básico para localizar un punto en el mapa, dependiendo de la versión de Google Maps el icono de un marker puede variar. Al momento de escribir este libro el icono predeterminado es este :



Si revisamos la documentación de Google Maps vemos que podemos crear un overlay marker usando la clase [google.maps.Marker](#), que recibe como parámetro un objeto donde tienen que estar definidas las opciones de nuestro marker. Entre todas las opciones que tenemos para escoger, estas son las más importantes:

position : Es la posición en el mapa donde queremos localizar nuestro marker y recibe como parámetro un objeto [latLng](#).

map (opcional) : Es el mapa en el cual queremos que aparezca nuestro marker, ya que en una aplicación web podemos tener más de un mapa cargado. Una cosa importante aquí es que si no especificamos el mapa, se creará el marker pero no se añadirá a ningún mapa.

Para los ejemplos siguientes usaremos los archivos de practica numero “3”, en el cual tenemos a nuestra disposición un botón llamado “click me”, para ejecutar nuestro código. Recuerda copiar y pegar los códigos de ejemplo en el evento click del botón.

Primero crearemos un marker con un click en el botón.

```
$('#actionButton').on('click',function() {  
    var marker = new google.maps.Marker({  
        position: new google.maps.LatLng(-34.397, 150.644),  
        title : "Hola soy el marcador",  
        map: map  
    });  
});
```

Como puedes ver usamos la clase “[google.maps.Marker](#)” para crear nuestro maker y estamos pasando como parámetro un objeto con 3 propiedades básicas para ver nuestro marker.

- Para la propiedad “position” se ha creado un objeto [LatLng](#), que representa la posición geográfica a señalar.
- Para la propiedad “map” pasamos nuestra variable global map.
- La propiedad “title” es un tooltip que aparece cuando el cursor del mouse esta sobre el marker, muy útil para mostrar alguna información relevante.

Si das click en el botón, aparecerá el marker en el mapa.

Cambiando el icono del marker

Si por alguna razón quieres cambiar el icono de tu marcador, puedes hacerlo usando la propiedad llamada “icon”, donde puedes usar la url de alguna imagen.

Por ejemplo:

```
$('#actionButton').on('click',function() {
    var urlImg = 'http://gmaps-samples.googlecode.com/svn/trunk/markers/green/blank.png';
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        icon:urlImg,
        map: map
    });
});
```

Como puedes ver solo hemos pasado una URL, que fue guardada en la variable “urlImg” a la propiedad “icon”, que acepta una URL de imagen válida para el icono del overlay marker.

Existen sitios web donde puedes encontrar iconos para tus overlay marker, algunos son :

1. <http://gmaps-samples.googlecode.com/svn/trunk/markers/>
2. https://developers.google.com/chart/image/docs/gallery/dynamic_icons?hl=es#pins
3. <http://mapicons.nicolasmollet.com/>
4. <http://www.benjaminkeen.com/google-maps-coloured-markers/>
5. <http://www.mapito.net/map-marker-icons.html>
6. http://powerhut.co.uk/googlemaps/custom_markers.php
7. <http://www.cycloloco.com/shadowmaker/shadowmaker.htm>




En el link numero 2 puedes encontrar como configurar iconos con texto y color, construyendo una URL con los parámetros descritos en la documentación. Por ejemplo la siguiente URL crea un pin rojo con una letra A:

`https://chart.googleapis.com/chart?chst=d_map_pin_letter&chld=A%7CFF0000%7C000000`

El API Google Maps ofrece algunos símbolos integrados en la API que se pueden mostrar en overlays marker o polyline. Los símbolos predeterminados incluyen un círculo y dos tipos de flechas. Ejemplo :

```
$('#ActionButton').on('click',function() {
    var arrow = {
        path: google.maps.SymbolPath.FORWARD_OPEN_ARROW,
        scale: 5
    };
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        icon: arrow ,
        map: map
    });
});
```

Para usar estas formas, tenemos que crear un objeto “[google.maps.Symbol](#)”. Con la propiedad “scale” controlas el tamaño del símbolo y en la propiedad “path” pasamos una de las constantes de “[google.maps.SymbolPath](#)” de las siguientes:

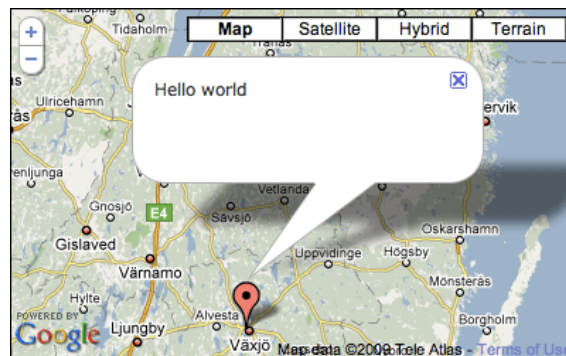
Nombre	Descripción	Ejemplo
<code>google.maps.SymbolPath.CIRCLE</code>	Un círculo	
<code>google.maps.SymbolPath.BACKWARD_CLOSED_ARROW</code>	Una flecha que señala hacia atrás y que está cerrada en todos sus lados	
<code>google.maps.SymbolPath.FORWARD_CLOSED_ARROW</code>	Una flecha que señala hacia delante	

	y que está cerrada en todos sus lados	
<code>google.maps.SymbolPath.BACKWARD_OPEN_ARROW</code>	Una flecha que señala hacia atrás y que está abierta por un lado	
<code>google.maps.SymbolPath.FORWARD_OPEN_ARROW</code>	Una flecha que señala hacia delante y que está abierta por un lado	

Agregar una Infowindow

Ok ahora que hemos agregado un punto de interés en nuestro mapa usando un overlay maker, seria de mucha ayuda para nuestros usuarios, mostrar más información que solamente su posición.

Para esto usaremos las Infowindows, que son una especie de globo que aparece sobre el marker, mostrando alguna información relevante. En dependencia de la versión de Google Maps, el estilo de la infowindow puede variar, al momento de escribir este libro es estilo es el siguiente:



Para crear un objeto infowindows usamos la clase [google.maps.InfoWindow](#), que según la API recibe un objeto, donde tienen que estar definidas sus opciones. Entre estas opciones la más importante es :

content (content:string|Node) : El valor que asignamos a esta propiedad será el contenido que mostrará la infowindows, puede ser texto plano o HTML. Lo que significa que puedes insertar videos o imagenes en tu infowindows. Para crear una infowindows solo necesitamos hacer una instancia de su clase. Por ejemplo :

```
var infowindow = new google.maps.InfoWindow({
    content : 'Hola soy una infowindows'
});
```

Con esto hemos creado nuestro objeto infowindows, que mostrará el texto “Hola soy una infowindows”, pero no aparecerá en nuestro mapa aun. Lo que haremos, es que se muestre cuando el usuario haga click en nuestro overlay marker, para lograr esto necesitamos vincular un evento click a nuestro marker.

Crear eventos

Nosotros podemos agregar eventos sobre casi cualquier objeto que hayamos creado, cuando hablo de eventos me refiero a acciones, que nosotros queremos realizar cuando el usuario hace click sobre algún elemento del mapa . Por ejemplo ejecutar algún método que nosotros queramos.

Para crear eventos en Google Maps, usamos la clase “[google.maps.event](#)” con la cual podemos vincular cualquier evento del mouse que nosotros queramos. Esta clase tiene distintos métodos estáticos que podemos usar, pero los más importantes son :

addListener(instance:Object, eventName:string, handler:Function) : Vincula un evento al cual se ejecutará cada vez que ocurra.

addListenerOnce(instance:Object, eventName:string, handler:Function) : Vincula un evento solo una única vez , después será removido.

Estos 2 metodos reciben 3 parámetros :

- El objeto al cual se vinculara el evento.
- El evento que queremos usar.
- El método que queremos ejecutar, cuando el evento sea realizado.

Ahora vamos hacer un ejercicio usando esta clase . Por ejemplo, hasta ahora hemos usado el evento `READY` de jquery para ejecutar nuestro código, pero esto no es aconsejable cuando queremos ejecutar algún código referente a nuestro mapa, por que el evento [READY](#) solamente se ejecuta cuando la página web está cargada .

El problema con esto, es que el mapa se carga por medio de [AJAX](#). Lo que indica que no sabemos en qué momento el mapa esta totalmente cargado y listo. Si nosotros intentamos ejecutar algun codigo y el mapa no esta cargado todavía, seguramente tendremos algún error.

La mejor manera que abordar este problema, es usar un evento que nos indique cuando el mapa esta totalmente cargado y listo. Para esto usamos la clase [google.maps.event](#) . Si abrimos el archivo de ejemplo de la carpeta número “4” y ejecutamos el archivo “index.html”, podremos ver como una ventana nos muestra el mensaje “El mapa esta cargado” . Ahora analicemos un poco el código del archivo “googlemaps.js”:

```
//document ready wait until the page is loaded
$(document).ready(function() {
    //obj with option
    var mapOptions = {
        center: new google.maps.LatLng(-34.397, 150.644),
        zoom: 10,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    //create a instance
    map = new google.maps.Map(document.getElementById("map"), mapOptions);
    // trigger when map is load
    google.maps.event.addListenerOnce(map, 'idle', function(){
        alert ('El mapa esta cargado');
    });
});
```

Como podemos ver, lo nuevo en nuestro código es el uso de [google.maps.event.addListenerOnce\(\)](#). Al cual estamos pasando los 3 parámetros requeridos : el objeto , el evento y el método a ejecutar. En este caso el segundo parámetro es el evento llamado “idle” este es un evento del objeto mapa. El cual nos indica cuando el mapa esta listo, recuerda que este método de esta clase solo se ejecutará una única vez. Si consultamos la API podemos ver muchos más eventos del objeto mapa que veremos más adelante.

Desde ahora en adelante, usaremos este evento para ejecutar todo nuestros códigos de ejemplo ya que es la manera adecuada de asegurarnos que el mapa esta listo.

En el siguiente enlace podrás ver todos los eventos que pueden ser ejecutados en el objeto “[google.maps.Map](#)” en acción : [All events map](#).

Crear evento en el marker

Ahora que sabemos como crear eventos , podemos proceder a crear el evento click en nuestro marker.

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        map: map
    });
    // Adding a click event to the marker
    google.maps.event.addListener(marker, 'click', function() {
        alert('click en el marker');
    });
});
```

Si copias y pegas este código en nuestro archivo de ejemplo número “4”, podrás ver cómo al hacer click en el overlay marker, ahora aparece una ventana que dice “click en el marker”. Como puedes ver estamos pasando los 3 parámetros necesarios : el objeto al que queremos vincular el evento , el evento que queremos escuchar “click” y el método que queremos ejecutar cuando este evento sea realizado.

Ahora que tenemos nuestro evento click en nuestro marker, podemos proceder a abrir nuestra infowindows sobre el overlay marker de la siguiente manera:

```
var infowindow = new google.maps.InfoWindow({
    content : 'Hola soy una infowindows'
});
google.maps.event.addListenerOnce(map, 'idle', function() {
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        map: map
    });
});
```

```
// Adding a click event to the marker
google.maps.event.addListener(marker, 'click', function() {
    infowindow.open(map, marker);
});
});
```

Como puedes ver estamos usando el método "[open](#)" del objeto infowindow, el cual lo hace aparecer en el mapa y recibe 2 parametros, el primero es el objeto mapa donde aparecer (en caso de que tengas más de un mapa en tu web), el segundo es un objeto sobre el cual aparecerá la infowindow en nuestro caso estamos usando nuestro overlay marker.

Infowindow contenido HTML

La propiedad "content" del objeto infowindow, puede tomar contenido HTML en formato cadena para mostrar cualquier información que tu quieras por ejemplo un video de youtube.

```
var infowindow = new google.maps.InfoWindow({
    content : '<iframe width="200" +
              'height="150"
src="http://www.youtube.com/embed/-jMrufHTwrY?list=PLsWFeoHgznCxviDIuVAp4yPk09dt8Sr79"' +
              'frameborder="0" allowfullscreen></iframe>'
});

google.maps.event.addListenerOnce(map, 'idle', function() {
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        map: map
    });
    // Adding a click event to the marker
    google.maps.event.addListener(marker, 'click', function() {
        infowindow.open(map, marker);
    });
});
```

Si copias y pegas este código en nuestro archivo de ejemplo, podrás ver que al hacer click sobre el overlay marker, ahora la infowindow muestra un video de youtube. De esta manera puedes mostrar imágenes, tablas y cualquier otro contenido HTML. Un tip en este punto es que para controlar el tamaño de la infowindow debes de hacerlo por medio de CSS asignando una clase o ID al elemento HTML.

Usando la clase [google.maps.event.trigger](#) puedes abrir la info windows mediante código, si así lo prefieres. Por ejemplo

```
google.maps.event.trigger(marker, 'click');
```

Cambiado el icono del maker con eventos

En el siguiente ejemplo vamos a cambiar el icono del overlay marker, cuando ocurra lo siguiente : Cuando el usuario coloque el cursor del mouse sobre el overlay y cuando haga click sobre el overlay.

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        map: map,
        icon : 'http://gmaps-samples.googlecode.com/svn/trunk/markers/green/blank.png'
    });

    google.maps.event.addListener(marker, 'mouseover', function() {
marker.setIcon('http://gmaps-samples.googlecode.com/svn/trunk/markers/orange/blank.png');
    });

    google.maps.event.addListener(marker, 'mouseout', function() {
marker.setIcon('http://gmaps-samples.googlecode.com/svn/trunk/markers/green/blank.png');
    });

    google.maps.event.addListener(marker, 'click', function() {
marker.setIcon('http://gmaps-samples.googlecode.com/svn/trunk/markers/circular/yellowcirclemarker.png');
    });

});
```

Si copias y pegas este código, podrás ver como el icono del overlay marker cambia cuando situamos el cursor sobre el y cuando hacemos click . Algunos otros eventos que puedes usar de la API de Google Maps son:

- click
- dblclick
- mouseup
- mousedown
- mouseover
- mouseout

Maker arrastrable en el mapa y animado

Con la propiedad “draggable” del objeto marker podemos habilitar, que el marker sea arrastrado por el usuario.

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        map: map ,
        draggable : true
    });
});
```

Y con la propiedad “animation” podemos darle cierta animación al marker usando la clase [google.maps.Animation](#).

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var marker = new google.maps.Marker({
        position: new google.maps.LatLng(-34.397, 150.644),
        title : "Hola soy el marcador",
        map: map ,
        draggable : true
    });
    google.maps.event.addListener(marker, 'click', function() {
        if (marker.getAnimation() != null) {
            marker.setAnimation(null);
        } else {
            marker.setAnimation(google.maps.Animation.BOUNCE);
        }
    });
});
```

Aquí usamos el método **getAnimation()** del objeto marker, para saber si hay alguna animación activa y si la hay, entonces la detenemos pasando el valor null a **setAnimation()**.

Overlay : Polylines.

Ahora pasaremos a ver cómo dibujar figuras geométricas en nuestro mapa, que nos permitirá crear caminos , bordes y áreas. Primero vamos comenzar con la overlay Polyline , Una simple polyline esta formada por un punto inicial y otro final, los cuales se conectan por una línea . También una polyline puede ser una colección de puntos para formar formas más complejas.



Cada punto de la polyline será un punto `LatLng` en el mapa, que debemos crear de antemano, como lo hemos hecho anteriormente. Aquí la parte difícil, es saber cuales son esos puntos `LatLng` en el mapa. Por ahora vamos a usar valores fijos, que sabemos de antemano. Pero más adelante vamos a dejar que el propio usuario dibuje su propia polyline donde desee.

A continuación podemos abrir el archivo de ejemplo de la carpeta número "5" y ejecutamos el archivo "index.html" y veremos a continuación una polyline creada por el siguiente código:

```
var polylineCoordinates = [
    new google.maps.LatLng(37.772323, -122.214897),
    new google.maps.LatLng(21.291982, -157.821856),
    new google.maps.LatLng(-18.142599, 178.431),
    new google.maps.LatLng(-27.46758, 153.027892)
];
var polyline = new google.maps.Polyline({
    path: polylineCoordinates,
    map : map
});
```

Lo primero que estamos haciendo, es crear un arreglo de puntos `LatLng` llamado "polylineCoordinates", donde cada elemento del arreglo es un punto [LatLng](#). Seguido creamos nuestra polyline usando la clase "[google.maps.polyline](#)" y pasamos nuestro objeto que tiene como propiedades las opciones básicas :

map : Que es nuestro objeto mapa donde queremos que aparezca la polyline.

Path : Que son las coordenadas de puntos `LatLng` , en este caso hemos definido un arreglo de puntos. Ya que esta propiedad recibe un arreglo de puntos.

Con esto basta para crear una polyline básica. A continuación, jugaremos un poco con las opciones adicionales disponibles. Por ejemplo podemos hacer nuestra polyline de otro color ,

ancho y que sea editable para el usuario, esto permitirá al usuario cambiar la forma de la polyline.

```
var polyline = new google.maps.Polyline({
  path: polylineCoordinates,
  strokeColor: "#FF0000",
  strokeWeight: 3,
  editable: true,
  map: map
});
```

Si copiamos y pegamos este código, reemplazando el anterior y recargamos la página podemos ver que nuestra polyline es de color rojo y más gruesa. Esto es debido a “strokeColor” y “strokeWeight”. También habrás notado que ahora aparecen unos círculos en cada punto de la polyline esto significa que puedes arrastrar con el mouse estos puntos por el mapa para cambiar la forma de la polyline a libertad. Esto se logra con la propiedad “editable”.

Una opción de polyline que me gustaría mencionar es “strokeOpacity” que sirve para volver la polyline transparente. Según la documentación en línea se definen así :

strokeColor: Especifica un color HTML hexadecimal del formato “#FFFFFF”. La clase Polyline no admite colores con nombre.

strokeOpacity : Especifica un valor fraccional numérico entre 0.0 y 1.0 (predeterminado) correspondiente a la opacidad del color de la línea.

strokeWeight : Especifica el grosor del trazo de la línea en píxeles.

Ahora quisiera hablar un poco sobre la opción “path”, según la API se define de la siguiente manera :

path	MVCArray.<LatLng> Array.<LatLng>
------	--

Como puedes ver, según la definición podemos usar un arreglo de LatLng normal, a como lo hicimos en el ejemplo anterior o un MVCArray . ¿ Pero que es un MVCArray ?

Bueno es un tipo de objeto en la API de google Maps, básicamente es un arreglo que contiene puntos LatLng, pero la diferencia es que contiene métodos especiales para insertar, borrar y obtener valores de sus elementos. La gran ventaja de usar un MVCArray en lugar de un arreglo normal, es que si modificamos nuestro MVCArray en cualquier momento, después de que la

polyline fue dibujada, esta se actualizará automáticamente con los nuevos valores que tenga el MVCArray , ya sea borrando un punto o agregando uno nuevo.

Si abrimos el “index.html” de la carpeta de práctica número “6”, podemos ver que tenemos nuestra polyline dibujada en el mapa de color rojo, pero hacemos click en el botón “click me”, podrás ver como la polyline se le resta un punto LatLng de su path, cada vez que haces click en el botón. Ahora pasamos a ver el código:

```
//event click jquery
$('#actionButton').on('click',function(){
    MVCArray.pop();
});
// when the map is ready
google.maps.event.addListenerOnce(map, 'idle', function() {
    MVCArray = new google.maps.MVCArray([
        new google.maps.LatLng(37.772323, -122.214897),
        new google.maps.LatLng(21.291982, -157.821856),
        new google.maps.LatLng(-18.142599, 178.431),
        new google.maps.LatLng(-27.46758, 153.027892)
    ]);
    var polyline = new google.maps.Polyline({
        path: MVCArray,
        strokeColor: "#FF0000",
        strokeWeight: 3,
        editable: true,
        map:map
    });
});
```

Ok lo primero a notar en el archivo “googlemaps.js” de carpeta número “6”, es que he definido la variable “MVCArray” de forma global para poder acceder a ella desde cualquier lugar de la aplicación.

Segundo para crear un MVCArray, usamos la clase [google.maps.MVCArray](#) y según la API el único parámetro necesario es un arreglo de puntos LatLng. Una vez creado lo pasamos en la propiedad “path” de las opciones de nuestra Polyline.

Tercero en el evento click del botón, ejecutamos el método “pop” del objeto [MVCArray](#), que habíamos creado anteriormente y guardado en la variable del mismo nombre. Lo que hace el método “pop” es remover el último elemento del arreglo y automáticamente nuestra polyline se actualiza con el nuevo conjunto de elementos en el MVCArray. Esto mismo lo podemos hacer

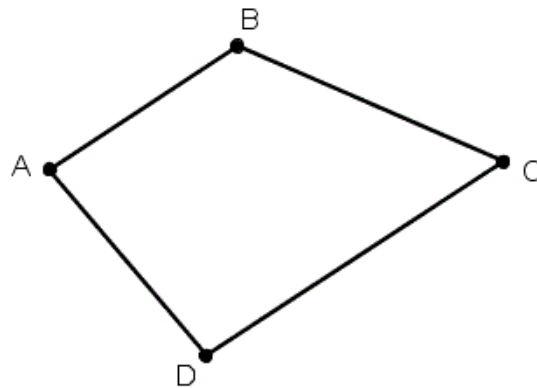
con todos los métodos disponibles para un objeto MVCArray, que podemos consultar en la API. Por ejemplo : removeAt , push , setAt , etc.

Para finalizar les presento este código de ejemplo sacado de este [enlace](#) en la documentación de Google Maps, sobre cómo agregar un flecha al final de nuestra polyline aplicado a nuestro ejemplo:

```
var lineSymbol = {
  path: google.maps.SymbolPath.FORWARD_CLOSED_ARROW
};
var polyline = new google.maps.Polyline({
  path: MVCArray,
  strokeColor: "#FF0000",
  icons: [{
    icon: lineSymbol,
    offset: '100%'
  }],
  map: map
});
```

Overlay : Polygons

Polygons son una serie de puntos LatLngs, que tienen un punto inicial y un punto final y que se conectan para formar una figura cerrada en el mapa. Las polylines se pueden usar para marcar rutas y los polygons son muy útiles para marcar áreas geográficas en el mapa.



Ahora pasemos a ejecutar el archivo "index.html" de la carpeta de practica numero "7", en el cual podremos ver en el mapa un overlay polygon de 3 puntos, el cual es formado por el siguiente código.


```

var points = [
    new google.maps.LatLng(37.7671, -122.4206),
    new google.maps.LatLng(36.1131, -115.1763),
    new google.maps.LatLng(34.0485, -118.2568),
];
var polygon = new google.maps.Polygon({
    paths: points,
    map: map
});

```

Al igual que la polyline, definimos un arreglo de puntos LatLng que forman la figura de nuestro polygon, para crear nuestro polygon usamos la clase "[google.maps.Polygon](#)". El cual recibe como parámetro un objeto con las opciones de la clase "google.maps.Polygon", de la misma manera como hicimos con el overlay polyline. En este caso, solo usamos la propiedad "paths" que son una secuencias de puntos LatLng y "map" que es el mapa donde queremos dibujar nuestra overlay polygon. Debes de notar que en nuestro arreglo de LatLng no esta el punto final de cierre de nuestro polygon, esto es debido a que el punto inicial siempre es considerado el punto final de cierre del overlay polygon.

De esta manera, es que se crea un polygon simple. Si consultas la API de la clase "[google.maps.Polygon](#)" Podrás notar que en las opciones, tenemos propiedades muy similares a las que tiene polyline, pero hay 2 propiedades que vale la pena discutir de polygon:

fillColor : Define el color del área dentro del polygon.

fillOpacity : Define la opacidad del área dentro del polygon.

```

var points = [
    new google.maps.LatLng(37.7671, -122.4206),
    new google.maps.LatLng(36.1131, -115.1763),
    new google.maps.LatLng(34.0485, -118.2568),
];

var polygon = new google.maps.Polygon({
    paths: points,
    map: map ,
    strokeColor: '#0000ff',
    strokeOpacity: 0.6,
    strokeWeight: 1,
    fillColor: '#40FF00',
    fillOpacity: 0.35
});

```

Si copias y pegas este código para reemplazar el anterior y refrescamos la página, veremos ahora como nuestro polygon tiene un área verde transparente y con las líneas de los bordes azules.

Entre todas las propiedades de polygon existe una que vale la pena mencionar es **zIndex**, esta propiedad sólo es útil cuando tienes varios polygons en el mapa y con esta propiedad puedes controlar el orden en el que son situados en el mapa. Por ejemplo un polygon con el valor de zIndex de 2 será dibujado sobre el polygon que tenga el zIndex de 1. Si no especificas el valor de esta propiedad los polygons se colocarán en el orden que se vaya ejecutando en el código.

Ahora vamos a hacer algo interesante para practicar los eventos del usuario en nuestro mapa:

```
var points = [
    new google.maps.LatLng(37.7671, -122.4206),
    new google.maps.LatLng(36.1131, -115.1763),
    new google.maps.LatLng(34.0485, -118.2568),
];
var polygon = new google.maps.Polygon({
    paths: points,
    map: map,
    strokeColor: '#0000ff',
    strokeOpacity: 0.6,
    strokeWeight: 1,
    fillColor: '#40FF00',
    fillOpacity: 0.35
});
google.maps.event.addListener(polygon, 'click', function() {
    alert('Hola soy el polygon');
});
google.maps.event.addListener(polygon, 'mouseover', function() {
    polygon.setOptions({
        fillColor: '#0000ff',
        strokeColor: '#0000ff'
    });
});
google.maps.event.addListener(polygon, 'mouseout', function() {
    polygon.setOptions({
        fillColor: '#40FF00',
        strokeColor: '#0000ff',
    });
});
```

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

Ahora si copiamos y reemplazamos el código anterior, por este de arriba y refrescamos la página, podemos notar que al pasar el mouse por el overlay polygon cambia de color y al retirarlo vuelve a su estado normal. También si hacemos click sobre él, nos abrirá una ventana que dice “Hola son el polygon”. Esto lo podemos lograr usando una propiedad común entres los overlays llamada “setOptions” que es para cambiar opciones cuando nosotros queramos .

Overlay : Circle

Circle es otro overlay que podemos crear dentro de nuestro mapa. Es similar a un Polygon ya que puedes definir colores, grosores y niveles de opacidad personalizados. A diferencia de un Polygon, en overlay circle no se definen paths. En su lugar tiene dos propiedades que definen su forma:

center : Especifica los valores de latitud y longitud en Google Maps (google.maps.LatLng) del centro del círculo.

radius: Especifica el radio del círculo, en metros.

Ahora podemos abrir el archivo “index.html” de la carpeta de práctica numero “8”, y podremos ver como en nuestro mapa ahora aparece un overlay circle, el cual es creado por el siguiente código:

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    circle = new google.maps.Circle({
        strokeColor: "#FF0000",
        strokeOpacity: 0.8,
        strokeWeight: 2,
        fillColor: "#FF0000",
        fillOpacity: 0.35,
        map: map,
        center: new google.maps.LatLng(34.052234, -118.243684),
        radius: 3844829
    });
});
```

Para poder crear un circle, necesitamos usar la clase “[google.maps.Circle](#)” y pasamos un objeto con las opciones como lo hemos hecho antes. En este código podemos ver muchas propiedades similares que hemos usado anteriormente, las que tienes que notar son “center” y “radius” que son las únicas propiedades obligatorias para crear un overlay circle.

Hasta aquí es todo con esta overlay no tenemos mucho que hablar sobre ella, solo que es un círculo que podemos usar en nuestro mapa.

Overlay : Rectangle

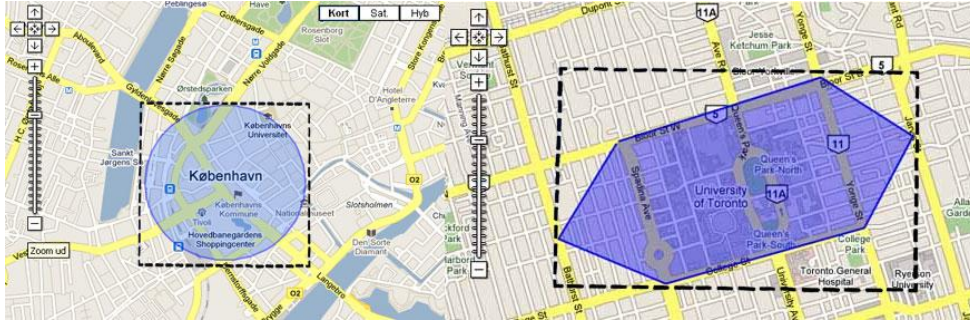
Un overlay rectangle es muy similar a circle, hablando sobre las opciones que se pueden configurar. El único parámetro obligatorio de un rectangle es :

bounds: Es un objeto específico de Google Maps, que se crea con la clase "[google.maps.LatLngBounds](#)", al cual necesitamos pasar 2 parámetros que son las coordenadas "southwest" y "northeast" que corresponden a las esquinas suroeste y noreste del bounds según la API. La siguiente imagen ilustra cuales son estos puntos.

`LatLngBounds(sw?:LatLng, ne?:LatLng)`



En otras palabras, bounds es un rectángulo que define un área y sus esquinas son coordenadas geográficas definidas por LatLng, por lo tanto todo lo que se encuentra dentro se considera dentro de sus límites ó bounds. Esto puede ser muy útil para calcular si un objeto está dentro de cierta área en el mapa, ya que algunos overlay poseen sus propios bounds como por ejemplo circle y polygon. Tengo que mencionar que el viewport de nuestro mapa donde aparecen todas las figuras también es un bounds, y cada vez que hacemos zoom y nos movemos por el mapa, ese bounds cambia constantemente. La siguiente imagen muestra en líneas punteadas negras los bounds imaginarios de circle y polygon, que aunque no los veamos en el visor los bounds siempre estan presente.



Para crear un bounds lo hacemos de la siguiente manera, usando la clase [google.maps.LatLngBounds](#) de Google Maps.

```
var bounds = new google.maps.LatLngBounds(
    new google.maps.LatLng(37.775, -122.419),
    new google.maps.LatLng(47.620, -73.986)
);
```

Recomiendo revisar los métodos del objeto bounds en la documentación de la API, que ya posee métodos muy útiles a tener en cuenta, como por ejemplo :

`contains(latLng:LatLng)` que retorna true si un objeto LatLng esta dentro del bounds.

Ahora volviendo a nuestro overlay rectangle, vamos a abrir el “index.html” de la carpeta de practica numero “9”, donde vamos a ver un overlay rectangle dibujado en el mapa con el siguiente código:

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var bounds = new google.maps.LatLngBounds(
        new google.maps.LatLng(37.775, -122.419),
        new google.maps.LatLng(47.620, -73.986)
    );
    rectangle = new google.maps.Rectangle({
        strokeColor: "#FF0000",
        strokeOpacity: 0.8,
        strokeWeight: 2,
        fillColor: "#FF0000",
        fillOpacity: 0.35,
        map: map,
        bounds: bounds
    });
});
```

Algunos eventos a tener en cuenta de los overlay

Anteriormente vimos como crear eventos para nuestros overlay, ahora vale la pena revisar una tabla que nos muestra que tipo de eventos adicionales podemos capturar para ellos. Según la documentación del API, cuando un overlay tiene la propiedad "editable" en true, tenemos los siguientes eventos que podemos capturar para realizar las cualquier acción.

Forma	Eventos
Círculo	<code>radius_changed</code> , <code>center_changed</code>
Polígono	<code>insert_at</code> , <code>remove_at</code> , <code>set_at</code>
Polilínea	<code>insert_at</code> , <code>remove_at</code> , <code>set_at</code>
Rectángulo	<code>bounds_changed</code>

Un ejemplo de como usarlo:

```
google.maps.event.addListener(circle, 'radius_changed', function() {
    //acciones que tu quieras realizar
});
```

Overlay : Ground

Con este tipo de overlay, podemos crear un objeto que nos permitirá mostrar un imagen dentro del mapa, para crearlo usamos la clase "[google.maps.GroundOverlay](#)" que permite especificar la URL de una imagen y el objeto `LatLngBounds`, donde la imagen será insertada en forma de parámetros. Ahora abramos el archivo "index.html" de la carpeta "10" donde podremos ver nuestro ejemplo en acción con el siguiente código.

```
var imageBounds = new google.maps.LatLngBounds(
    new google.maps.LatLng(37.775,-122.419),
```

```

        new google.maps.LatLng(47.620, -73.986)
    );
    var imageLink = "http://wowslider.com/images/demo/pinboard-fly/data/images/desert_landscape.jpg";
    overlay = new google.maps.GroundOverlay(imageLink, imageBounds);
    overlay.setMap(map);

```

Primero definimos el bounds para nuestra imagen, recuerda que las imágenes siempre son rectangulares por esa razón necesitamos un bounds, que es el lugar donde queremos colocar nuestra imagen. Después guardamos en una variable el link de nuestra imagen que pasaremos como parámetro a la clase [“google.maps.GroundOverlay”](#).

Hasta aquí, creo que no hay más que discutir sobre el código. Creamos nuestra instancia de la clase y pasamos sus 2 parámetros necesarios, seguido asignamos el mapa donde queremos que aparezca.

Hace unos meses modifique un ejemplo de Google Maps que permite al usuario ingresar un link de una imagen para insertarla en el mapa , la diferencia con el ejemplo anterior que vimos, es que permite redimensionarla y moverla a por todo el mapa , si quieres ver el código y el ejemplo funcionando visita este link :

<http://webmasternoob.blogspot.com/2013/01/google-maps-overlayview-con-imagen.html>

Drawing Library

Bueno y llegamos al punto donde seguramente has pensado : ”voy crear una aplicación que permita al usuario dibujar formas dentro del mapa”. Si lo has hecho, ya habrás anticipado que necesitas crear cada boton de accion que dibuje cada overlay en el mapa, lo que se traduce en mucho trabajo por hacer.

Antes de que empieces con ese proyecto, déjame presentarte la “Drawing Library” de Google Maps, que son un conjunto de herramientas que permitirán al usuario dibujar todas las overlay en el mapa, sin que nosotros tengamos que escribir código. En palabras de Google Maps la definición sería :

“La clase DrawingManager, proporciona una interfaz gráfica que permite que los usuarios dibujen polígonos, rectángulos, polilíneas, círculos y marcadores en el mapa.”

Ok para comenzar a usar esta clase, lo primero que tenemos que hacer es cargar la “Drawing Library” en nuestro “index.html” de la carpeta de práctica número “11”.

En nuestros ejemplos siempre cargamos Google Maps de la siguiente manera :

```

<script type="text/javascript"

```



```
src="http://maps.googleapis.com/maps/api/js?sensor=false"></script>
```

Pero con esto, no estamos cargando la “Drawing library” aún, tenemos que agregar un parámetro adicional para cargar la “Drawing library” .

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?sensor=false&libraries=drawing"></script>
```

Con el código de arriba, cargamos la “Drawing library” pasando en la query string el parámetro “libraries” con el valor “drawing”. Ahora con nuestra “Drawing library” cargada podemos abrir nuestro “index.html” y veremos en viewport del mapa una caja de herramientas arriba con la podremos dibujar todas las overlay vistas anteriormente con solo 2 líneas de código:

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var drawingManager = new google.maps.drawing.DrawingManager();
    drawingManager.setMap(map);
});
```

Para crear nuestra Toolbar de dibujo, usamos la clase “ [google.maps.drawing.DrawingManager](#)” y solamente necesitamos usar el método setMap() de la clase, para situarlo en el mapa en su posición por defecto. Algunas de la opciones que podemos pasar a la clase son:

drawingMode: Con esta propiedad podemos especificar qué herramienta será la inicial cuando cargue el mapa, el parámetro que recibe es una constante de la clase “[google.maps.drawing.OverlayType](#)” de las que podemos escoger las siguientes :

```
google.maps.drawing.OverlayType.MARKER
google.maps.drawing.OverlayType.CIRCLE
google.maps.drawing.OverlayType.POLYGON
google.maps.drawing.OverlayType.POLYLINE
google.maps.drawing.OverlayType.RECTANGLE
```

Ejemplo :

```
drawingManager = new google.maps.drawing.DrawingManager({
    drawingMode: google.maps.drawing.OverlayType.MARKER
});
```

drawingControl: Valor booleano para mostrar o ocultar las herramientas.

drawingControlOptions: Usamos esta propiedad para definir cuáles herramientas queremos mostrar y en qué posición en el viewport del mapa mostrar las herramientas. Recibe como

parámetro un objeto “[google.maps.drawing.DrawingControlOptions](#)” que esta formado por 2 propiedades “**drawingModes**” donde especificamos cuales herramientas queremos mostrar y “**position**” donde especificamos la position en el viewport del mapa de las herramientas.

- **drawingModes** : Acepta un arreglo donde cada elemento del arreglo, es una constante de la clase “[google.maps.drawing.OverlayType](#)” que vimos anteriormente: `MARKER` , `CIRCLE` , `POLYGON` , `POLYLINE` , `RECTANGLE`.
- **position** : Acepta una constante de la clase “[google.maps.ControlPosition](#)” que podrian ser `BOTTOM_CENTER` , `BOTTOM_LEFT` , `BOTTOM_RIGHT` , `LEFT_BOTTOM` , `LEFT_CENTER` , `LEFT_TOP` , `RIGHT_BOTTOM` , `RIGHT_CENTER` , `RIGHT_TOP` , `TOP_CENTER` , `TOP_LEFT` , `TOP_RIGHT` .

Ejemplo :

```
drawingManager = new google.maps.drawing.DrawingManager({
  drawingControlOptions: {
    position: google.maps.ControlPosition.RIGHT_CENTER,
    drawingModes: [
      google.maps.drawing.OverlayType.MARKER,
      google.maps.drawing.OverlayType.CIRCLE
    ]
  }
});
```

markerOptions, **polygonOptions**, **polylineOptions**, **rectangleOptions** : Todas estas opciones trabajan de la misma manera. Lo que hacen es definir la configuración de cada overlay que se dibujara en el mapa, cada una de estas opciones acepta un objeto con la configuración de su tipo de overlay específico que ya hemos visto anteriormente.

Un método de “Drawing library” muy útil es “**setOptions()**”, el cual podemos usar para cambiar el comportamiento de nuestras herramientas, con solo pasar un objeto con las nuevas opciones que queremos configurar.

Ejemplo :

```
drawingManager.setOptions({
  drawingControl: false,
  drawingControlOptions: {
    position: google.maps.ControlPosition.BOTTOM_CENTER,
  }
});
```

Adicionalmente “Drawing library” nos ofrece eventos de los que podemos tomar ventaja para realizar ciertas acciones.

- **overlaycomplete** : Este evento ocurre cuando cualquier tipo de overlay a sido dibujado en el mapa.
- **{tipo-overlay}complete** : Este evento ocurre cuando un overlay específico ha sido dibujado en el mapa donde tenemos que reemplazar el nombre del overlay entre las llaves por ejemplo : “circlecomplete” , “rectanglecomplete” .

Si abrimos el archivo “index.html” de la carpeta practica numero “12”, podemos ver como cuando dibujamos un overlay rectangle ocurren 2 eventos , pero si dibujamos cualquier otro solo ocurre un evento con el siguiente código en el archivo “googlemaps.js”:

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    drawingManager = new google.maps.drawing.DrawingManager();
    google.maps.event.addListener(drawingManager, 'rectanglecomplete', function(event) {
        alert('event : Rectangle is complete');
    });
    google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
        alert('event : a overlay was drew');
    });
    drawingManager.setMap(map);
});
```

En el código vemos que pasamos el objeto “drawingManager”, con sus respectivos eventos que necesitamos para ejecutar acciones. El parámetro “event” que es pasado al callback cuando se ejecuta el evento, contiene un objeto con 2 propiedades:

type: Que es una cadena de texto, donde nos describe el tipo de overlay.

overlay: El objeto overlay creado, podemos usar este objeto cuando queramos en nuestra aplicación.

Capítulo 4 : Tareas Comunes en Google Maps

Trabajando con múltiples overlays

Hasta ahora hemos aprendido como crear overlays simples de uno en uno, pero existen ocasiones donde tenemos que crear más de un overlay a la vez. Por ejemplo supongamos que tenemos que agregar 5 overlays markers en el mapa, esto lo podríamos hacer perfectamente creando uno por uno ¿ Pero, qué pasa si tenemos que agregar 20 overlays markers al mapa ? nuestro código serian líneas repetidas 20 veces.

En estos casos es necesario usar un ciclo FOR, para crear los overlays de manera automática y liberarnos del problema. Podemos abrir la carpeta de ejemplo numero “13-0” y ejecutar el archivo “index.html”, la cual cargará nuestro mapa. Inicialmente solo se muestra el mapa pero si hacemos click en el botón del lado izquierdo “click me” verás aparecer 15 overlays markers. Para ver como funciona este ejemplo, podemos abrir el archivo “googleMaps.js” en el cual encontraremos 2 métodos que son ejecutados en el evento de click del botón “click me”.

```
$('#actionButton').on('click',function(){
    var arrayLatLng = CreateLatLngArray();
    CreateMarkers(arrayLatLng);
});
```

El método “CreateLatLngArray” lo usamos para generar puntos latlng aleatorios, que usaremos luego para crear los overlays markers. Este método nos devuelve, un arreglo de objetos donde cada elemento del arreglo es un objeto que tiene 2 propiedades “lat” y “lng” de la siguiente manera :

```
{
    lat:41.4220,
    lng:-97.4373
}
```

Con este arreglo guardado en la variable “arrayLatLng”, crearemos los overlays pasándolo como parámetro al método llamado “CreateMarkers” que es el encargado de crear los overlays.

```
function CreateMarkers (latlngs){
    var arrayMarkers = [];
    for (var i = 0; i < latlngs.length; i++) {
        var latlng = new google.maps.LatLng(latlngs[i].lat,latlngs[i].lng);
        var marker = new google.maps.Marker({
            position: latlng,
            map: map
        });
        arrayMarkers.push(marker);
    }
    return arrayMarkers;
}
```

En este método usamos un ciclo FOR, para iterar en el arreglo de objetos que contiene los puntos latlngs. Dentro del ciclo primero creamos el punto latlng que será usado por nuestro overlay marker, usando las propiedades “lat” y “lng” del elemento actual. Con el latlng creado solo queda pasarlo como parámetro a la clase [google.maps.Marker](#) para crear nuestro overlay. Cada overlay creado es guardado en el arreglo “arrayMarkers” y devuelto por el método. Una cosa importante que tienes que notar es que la variable “map” es global para nuestro ejemplo, de esa manera podemos usarla desde nuestro método.

Hasta aquí todo trabaja perfectamente en nuestro ejemplo, pero vamos hacer algunas modificaciones para que cada overlay marker tenga un infowindows. En nuestro ejemplo existe una variable global llamada “infowindow”, donde se ha guardado un objeto infowindow creado a partir de la clase “[google.maps.InfoWindow](#)”, usaremos este objeto para todos los overlays markers.

Vamos a crear un método llamado “AttachEventInfowindow” que recibe como parámetro un overlay marker y el index del ciclo FOR, y a continuación creará el evento click para este overlay.

```
function AttachEventInfowindow(currentMarker,index) {
    google.maps.event.addListener(currentMarker, 'click', function() {
        infowindow.setContent("Hi I am the marker number : " + index);
        infowindow.open(map,this);
    });
}
```

Cada vez que este método es ejecutado, creará un evento click sobre el overlay marker. Cuando el usuario haga click sobre el overlay marker, se asignará el contenido a mostrar al objeto infowindow con “setContent” y después abrirá el infowindow en el mapa con el método “open”, que recibe como parámetros el mapa donde aparecer y un punto ancla que segun la

documentacion es aconsejable que sea un overlay marker.

Si te estás preguntando por que el segundo parámetro del método “open” se llama “this”, es por que cuando el evento click ocurre y se ejecuta este método,”this” esta haciendo referencia al objeto al cual se ha hecho click, en este caso es un overlay marker.

Ahora necesitamos llamar al método desde “CreateMarkers” :

```
function CreateMarkers (latlngs){
    var arrayMarkers = [];
    for (var i = 0; i < latlngs.length; i++) {
        var latlng = new google.maps.LatLng(latlngs[i].lat,latlngs[i].lng);
        var marker = new google.maps.Marker({
            position: latlng,
            map: map
        });
        AttachEventInfowindow(marker,i);
        arrayMarkers.push(marker);
    }
    return arrayMarkers;
}
```

Cada vez que un overlay marker es creado, lo pasamos como parámetro al método “AttachEventInfowindows” junto con el índice del ciclo FOR, para crear el evento sobre el overlay actual. Y esto es todo, para que al hacer click en cualquiera de los overlay marker creados, aparezca una infowindow. Para ver un ejemplo completo, puedes ver el archivo de ejemplo en la carpeta numero “13-1”.

Nota: La creacion del evento click para los overlay marker, tiene su propio método fuera del ciclo FOR, por que no es posible crear eventos dentro de un ciclo. Si creas un evento dentro de un ciclo FOR, el método que crea el evento recibe la variable en sí misma y no su valor, entonces cuando el ciclo FOR termina de ejecutarse, las variables contienen el ultimo valor asignado por el ciclo, por lo cual se pierden todos los datos anteriores para el evento.

Esto sucede porque el evento click, se ejecuta mucho después cuando el ciclo FOR a terminado, por lo tanto el método click solo puede acceder al valor de la última iteración del ciclo FOR. Para entender un poco esto, recomiendo leer este artículo de [Douglas Crockford](#) y también uno es español [aqui](#).

Demasiados Marker en el mapa

Para que tengamos una mejor idea de lo que vamos tratar en este capítulo, vayamos a la carpeta de práctica número “13-2” y abramos el “index.html”. Veremos nuestro mapa vacío, si ahora hacemos click en el botón “click me”, verás como aparecen muchos marker en el mapa de forma aleatoria , entre más veces hagas click mas overlay marker van aparecer.



Como puedes ver hay demasiados markers en el mapa, lo cual hace difícil para el usuario enfocarse en un punto determinado, y que el desempeño del mapa sea más lento entre más markers hay en el viewport . Hay que aclarar algo primero, en dependencia del navegador que el usuario esté usando, el mapa se pondra mas lento. Por ejemplo Google Chrome soporta más markers en pantalla que Internet Explorer.

Si revisamos el código de nuestro ejemplo, podemos ver que el responsable de generar los markers aleatorios es un método llamado “GenerateRandomMarkers”, que se ejecuta cada vez que hay un click en el botón. Hablando de forma general el método “GenerateRandomMarkers”, toma el bounds del viewport, después calcula un punto medio dentro del bounds, el cual después es multiplicado por un número al azar para generar latLng aleatorios. No explicare como funciona este método en detalle porque esta fuera del ámbito del libro, que es aprender usar Google Maps.

Ahora lo importante, resolveremos este problema de muchos markers en pantalla para el usuario con librerías externas a Google Maps.

MarkerClusterer

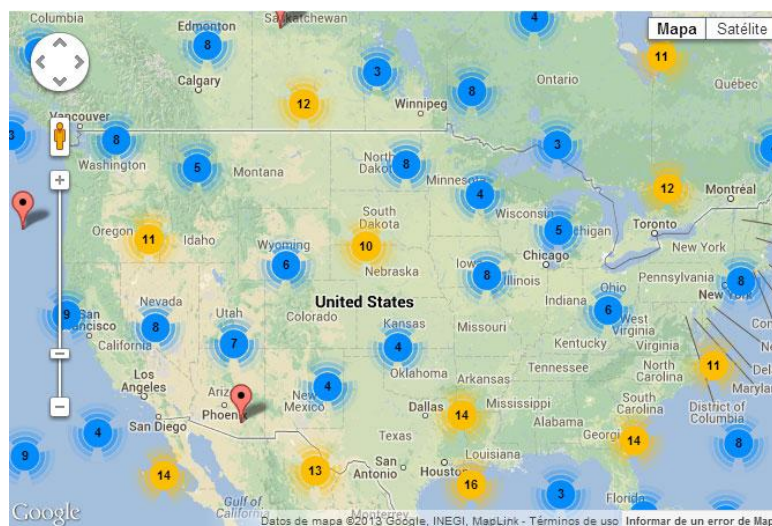
MarkerCluster es una librería del lado del cliente que nos ayuda a reducir el número de markers visibles en pantallas, y reemplazarlos por una representación en número de acuerdo al zoom del mapa y a la proximidad de cada uno.

La librería puede ser encontrada en este link:

<http://google-maps-utility-library-v3.googlecode.com/svn/trunk/markerclusterer/>

El archivo que necesitamos es “markerclusterer.js” que se encuentra en “src” , en el mismo link puedes encontrar ejemplos y la ayuda de referencia para ver las opciones de la librería.

Para ver esta librería en acción podemos abrir el archivo de práctica de la carpeta “14” y abrimos el “index.html”. Veremos a continuación que aparecen unos círculos de colores llamados “cluster” con un número en el centro, este número es la cantidad de markers en esa zona .



Hablemos un poco del código, lo primero que debes notar es que en la carpeta “14”, hay un nuevo archivo llamado “markerclusterer.js” este archivo es la librería que hemos descargado y también en el archivo “index.html” ahora tenemos esta nueva línea, que agrega la librería a nuestro ejemplo.

```
<script type="text/javascript" src="markerclusterer.js"></script>
```

Esto es todo lo que necesitamos para cargar la librería en nuestro ejemplo. Ahora pasemos a ver un poco de como usarla. Lo primero que tienes que notar en este ejemplo, es el llamado al método “GenerateRandomMarkers”, ahora se encuentra en nuestro evento “idle” del mapa para crear los markers al inicio de la aplicación. También el método “GenerateRandomMarkers”

ha sido modificado un poco para adecuarlo para el uso de la librería:

```
function GenerateRandomMarkers(){
    var markerRandom = [] ;
    var bounds = map.getBounds();
    var southWest = bounds.getSouthWest();
    var northEast = bounds.getNorthEast();
    var latSpan = northEast.lat() - southWest.lat();
    var lngSpan = northEast.lng() - southWest.lng();
    for (var i = 0; i < 300; i++) {
        var lat = southWest.lat() + latSpan * Math.random();
        var lng = southWest.lng() + lngSpan * Math.random();
        var latlng = new google.maps.LatLng(lat, lng);
        var marker = new google.maps.Marker({
            position: latlng,
            map: map
        });
        markerRandom.push(marker);
    }
    return markerRandom;
}
```

Primero se declara un arreglo llamado “markerRandom”, después por cada marker generado aleatoriamente lo guardamos en una variable llamada “marker”, la cual es usada para agregar un nuevo elemento al arreglo, de esta manera por cada marker generado agregamos un elemento al arreglo. Después cuando el ciclo FOR termina, devolvemos el arreglo con todos los elementos, donde cada uno de ellos es un marker.

Hasta aquí sólo hemos generado un arreglo, donde cada elemento del mismo es un marker aleatorio en el mapa. Ahora pasamos a la siguiente línea de código:

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    var markerArray = GenerateRandomMarkers();
    var markerclusterer = new MarkerClusterer(map, markerArray);
});
```

El resultado de “GenerateRandomMarkers” , lo usamos como parámetro para crear nuestro “markerclusterer” haciendo un instancia de la clase, la cual recibe los siguientes parametros:

[MarkerClusterer](#)(map:Map , markers?:Array , options?:Object)

Con esto es suficiente, para que nuestro markerclusterer funcione en nuestro mapa. A

continuación veremos un poco qué es lo que significan esos colores de los clusters.

- **Azul** : menos de 10 markers.
- **Amarillo**: 10 a 100 Markers.
- **Red** : Menos de 1000 markers.
- **Purpura** : Menos de 10.000 markers.
- **Purpura Oscuro** : Mas de 10.000 markers.

Para métodos adicionales y configuración, consulta la ayuda de referencia de la [librería](#).

Marker Spiderfier

Esta librería no oculta los markers en el mapa, sino que los expande en una especie de árbol para una mejor visualización, por lo que no mejoraría el rendimiento del mapa en caso de que tengas muchos overlay markers. Abramos el archivo “index.html” de la carpeta de practica numero “15”, y hacemos click en unos de los marker veremos como se expanden y contraen con cada click.

La librería y su documentación puede ser encontrada aquí [Overlapping Marker Spiderfier](#)

El codigo para usar esta librería se ha modificado un poco. Ahora pasaremos a explicar un poco de el. La parte principal donde hay que poner mucha atención es esta:

```
function GenerateRandomMarkers(){
    var bounds = map.getBounds();
    var southWest = bounds.getSouthWest();
    var northEast = bounds.getNorthEast();
    var latSpan = northEast.lat() - southWest.lat();
    var lngSpan = northEast.lng() - southWest.lng();
    var lat = southWest.lat() + latSpan * Math.random();
    var lng = southWest.lng() + lngSpan * Math.random();
    var latLng = new google.maps.LatLng(lat, lng);
    var marker = new google.maps.Marker({
        position: latLng,
        map: map
    });
    return marker;
}
```

Ahora nuestro método que genera nuestros random markers, ya no devuelve un arreglo de

markers. Ahora solo devuelve un marker cada vez que es ejecutado este método, y también tenemos lo siguiente.

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    oms = new OverlappingMarkerSpiderfier(map);
    for (var i = 0; i < 300; i++) {
        var markerCreated = GenerateRandomMarkers();
        oms.addMarker(markerCreated);
    }
    map.setZoom(3);
});
```

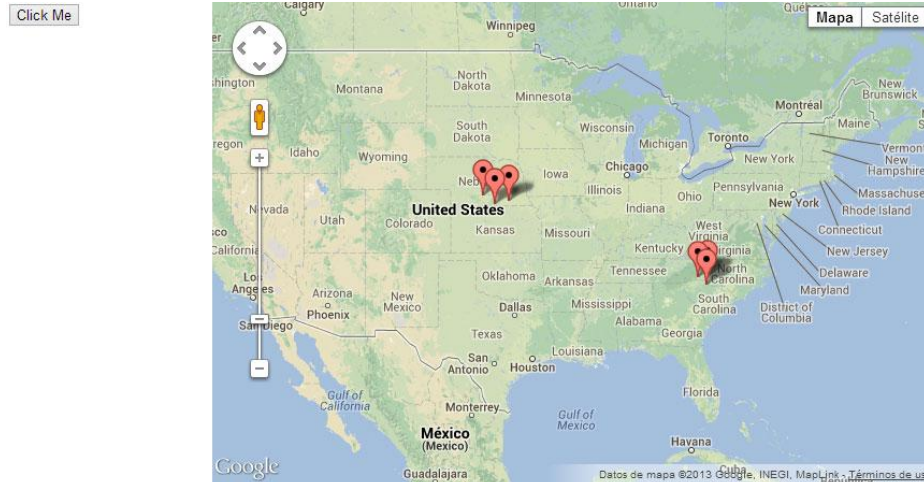
Lo primero que estamos haciendo es una instancia de la clase “[OverlappingMarkerSpiderfier](#)”, pasando como único parámetro el objeto mapa. Ahora nuestro ciclo FOR, esta afuera de nuestro método y ejecutamos el método dentro de él. Lo que conseguimos con esto, es un marker en cada ciclo que podemos asignar a la “oms.addMarker()” que según la documentación, solo recibe como parámetro un único marker y no un arreglo, es por esa razón que el código fue escrito de esta manera.

Para usar esta librería adecuadamente, primero tienes que identificar las zonas donde los marker esta aglomerados y muy cercanos y después agregarlos. Para ver en detalle más métodos y configuraciones la página oficial de esta [librería](#) .

MarkerManager

[MarkerManager](#) es otra librería para trabajar con markers, aunque implica un poco más de trabajo la configuración. Básicamente lo que hace es, por cada nivel de zoom tu decides que Markers mostrar en el mapa, de esta manera cuando el usuario hace zoom In o zoom out en el mapa, MarkerManager oculta ó muestra Markers basado en la configuración que decidas.

Para esta práctica vamos a trabajar de un manera un poco diferente, anteriormente siempre teníamos el ejemplo completo y después explicaba el código, Ahora vamos hacerlo paso a paso para un mejor comprensión. Así que para comenzar abre el archivo de practica “16”, y ejecuta el “index.html” el cual muestra un mapa con 6 markers localizados en diferentes zonas.



Lo primero a notar en este ejemplo es el archivo “index.html”, donde hemos agregado la librería [MarkerManager](#). He usado la versión “packed” de la librería que es más pequeña.

```
<script type="text/javascript" src="markermanager_packed.js"></script>
```

Después tenemos en nuestro script “googlemaps.js”, el siguiente método que se encarga de crear los markers y ubicarlos en el mapa.

```
function GetMarkers() {
    var locationMarker = [
        {"lat":40.1284,"lng": -98.4594},
        {"lat":40.6181,"lng": -99.3988},
        {"lat":40.3214,"lng":-97.3223},
        {"lat":35.3980,"lng": -81.8536},
        {"lat":35.4360,"lng": -81.0186},
        {"lat":34.8859,"lng": -81.0928}
    ];
    var markers = [];
    for (var i=0; i < locationMarker.length;i++)
    {
        var myLatLng = new google.maps.LatLng(locationMarker[i].lat,locationMarker[i].lng);
        var marker = new google.maps.Marker({
            position: myLatLng,
            title:"Hello I am " + i
        });
        markers.push(marker);
    }
    return markers;
}
```

En este método, existe un arreglo de objetos que contienen las coordenadas LatLng de los markers y lo único que hacemos es recorrer ese arreglo con el FOR, los cuales vamos agregando a un arreglo llamado “markers” que es devuelto por el método para usarlo después. Hay que notar que no estamos asignando todavía los overlay marker al mapa. Este método es llamado en evento “idle” del mapa.

Ahora vamos a crear nuestro objeto MarkerManager, el único parámetro obligatorio para crearlo es el objeto mapa.

MarkerManager(map:Map, options?:Object)

Sabiendo esto vamos agregar la siguiente línea de código :

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    var markersInMap = GetMarkers();
    mgr = new MarkerManager(map);
});
```

Con esto hemos creado nuestro objeto MarkerManager, ahora solo basta con agregar los markers que hemos creado anteriormente. MarkerManager tiene un método llamado “addMarkers()” el cual recibe como parámetros un arreglo de overlays arker y el nivel de zoom mínimo en el cual los markers deben aparecer.

`addMarkers(markers:Array of Marker, minZoom:Number, opt_maxZoom:Number)`

Antes de usar este método tienes que saber que MarkerManager trabaja a modo asíncrono, esto significa que tenemos que esperar hasta que MarkerManager esté listo para trabajar con el, igualmente a como lo hacemos con el objeto mapa de Google Maps. Esto lo podemos hacer escuchando el evento “loaded” del objeto MarkerManager.

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    var markersInMap = GetMarkers();
    mgr = new MarkerManager(map);
    google.maps.event.addListener(mgr, 'loaded', function() {
        //your code here
    });
});
```

Ahora podemos usar “addMarkers”, pasando nuestro arreglo de markers que devuelve el método “GetMarkers” y especificando el zoom mínimo donde queremos que aparezcan. En este caso usaremos 5 como zoom mínimo.

```
google.maps.event.addListener(mgr, 'loaded', function() {
    mgr.addMarkers(markersMap, 5);
    mgr.refresh();
});
```

Notar aquí que no estamos asignando los markers al mapa, sino al objeto MarkerManager. El método “refresh” se asegura que el objeto MarkerManager funciona correctamente, aunque no es necesario en nuestro ejemplo, puedes usarlo para actualizar el MarkerManager si agregas markers dinámicamente a tu objeto.

Ahora si actualizas nuestro ejemplo podrás ver que no aparece ningún marker en el mapa, pero si te acercas haciendo zoom lo suficiente aparecerán los markers en el mapa. Esto es porque inicialmente nuestro mapa tiene un zoom de 3 y al llegar al nivel 5, MarkerManager muestra los markers que fueron agregados.

Para ver la documentación MarkerManager puedes visitar este [link](#)

Usando MarkerManager como Cluster

Un uso muy conveniente que podemos lograr con MarkerManager, es crear nuestros propios cluster personalizados para nuestros markers. Para lograr esto haremos uso de algunos iconos que podemos encontrar en : <http://mapicons.nicolasmollet.com/>, usando el mismo ejemplo anterior.

Ahora lo primero que vamos a crear, será un arreglo que contenga nuestros clusters :

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    var markersMap = GetMarkers();
    mgr = new MarkerManager(map);

    var phantom = new google.maps.Marker({
        position: new google.maps.LatLng(40.1284, -98.4594),
        icon: 'phantomWhite.png'
    });
    var phantomWhite = new google.maps.Marker({
        position: new google.maps.LatLng(35.4360, -81.0186),
        icon: 'phantom.png'
    });
});
```

```

var cluster = [phantom ,phantomWhite];
google.maps.event.addListener(mgr, 'loaded', function() {
    mgr.addMarkers(markersMap, 5);
    mgr.refresh();
});
});

```

Primero creamos 2 markers con iconos personalizados en una ubicación que nosotros queramos, por ejemplo esos valores Latlng en nuestro ejemplo son en el centro del grupo de markers del ejemplo anterior. Después asignamos estos markers a un arreglo llamado “cluster”

El siguiente paso, es agregar nuestro arreglo de clusters al MarkerManager, pero esta vez usaremos el parámetro opcional “opt_maxZoom”.

`addMarkers(markers:Array of Marker, minZoom:Number, opt_maxZoom:Number)`

Con este parámetro opcional, podemos especificar un rango de niveles de zoom donde solo queremos que aparezcan los markers que deseamos. Ahora agregamos nuestro cluster a MarkerManager.

```

google.maps.event.addListenerOnce(map, 'idle', function(){
    var markersMap = GetMarkers();
    mgr = new MarkerManager(map);
    var phantom = new google.maps.Marker({
        position: new google.maps.LatLng(40.1284, -98.4594),
        icon: 'phantomWhite.png'
    });
    var phantomWhite = new google.maps.Marker({
        position: new google.maps.LatLng(35.4360, -81.0186),
        icon: 'phantom.png'
    });
    var cluster = [phantom ,phantomWhite];
    google.maps.event.addListener(mgr, 'loaded', function() {
        mgr.addMarkers(markersMap, 5);
        mgr.addMarkers(cluster, 1 , 4 );
        mgr.refresh();
    });
});

```

Lo que estamos haciendo aquí con la primera línea de código es :

```
mgr.addMarkers(markersMap, 5);
```

Estamos diciendo: Solo queremos que estos markers aparezcan desde el nivel de zoom 5 en adelante, y con esta línea:

```
mgr.addMarkers(cluster, 1 , 4 );
```

Estamos diciendo: Estos markers solo queremos que aparezcan, cuando el zoom esté entre 1 y 4 en el mapa. Lo que conseguimos con esto es un efecto de cluster muy bonito que podemos personalizar con nuestros propios iconos.

Si ahora actualizamos nuestro ejemplo en el navegador, podrás notar que aparecen unos iconos con unos fantasmas, pero si haces zoom lo suficiente podrás ver como desaparecen y se muestran las demas markers y viceversa.

Hasta ahora tenemos nuestro ejemplo muy bonito, pero creo que podemos mejorarlo aún más. ¿ Que te parece si agregamos un funcionalidad, que al hacer click en los fantasmas aparezcan los markers ocultos ? ok manos a la obra.

Vamos a modificar nuestro código de la siguiente manera, antes de asignar nuestros markers fantasmas al arreglo “cluster” les vamos a crear un evento click, donde vamos a ejecutar ciertas acciones.

```
var phantom = new google.maps.Marker({
    position: new google.maps.LatLng(40.1284, -98.4594),
    icon: 'phantomWhite.png'
});
google.maps.event.addListener(phantom, 'click', function() {
    map.setZoom(5);
    map.setCenter(phantom.getPosition());
});
var phantomWhite = new google.maps.Marker({
    position: new google.maps.LatLng(35.4360, -81.0186),
    icon: 'phantom.png'
});
google.maps.event.addListener(phantomWhite, 'click', function() {
    map.setZoom(5);
    map.setCenter(phantomWhite.getPosition());
});
```

Lo que pasara ahora, es que cuando hagamos click en los markers fantasmas vamos a asignar el nivel de zoom 5 al mapa, que es el nivel donde los markers ocultos aparecen y adicional a eso, vamos a centrar el mapa en esa misma posición usando “getPostion” del marker.

Ahora si actualizamos nuestro ejemplo, podremos comprobar este comportamiento. En la carpeta de práctica número “17”, podrás encontrar el código de MakerManager final de todos estos ejercicios.

Maplabel

Ahora veremos una librería muy útil para mostrar información junto con los markers llamada [MapLabel](#). Es muy sencilla de usar, primero abramos nuestro archivo de práctica de la carpeta número “18”, después ejecutamos el “index.html” veremos a continuación 3 markers con texto debajo de ellos.



Si arrastras los overlay marker, el texto también se arrastrará junto con ellos. Ahora veamos el código.

Lo primero, es que hemos agregado la librería Maplabel a nuestro ejemplo en el archivo “index.html”, la cual podemos encontrar [aquí](#), de esta manera.

```
<script type="text/javascript" src="maplabel-compiled.js"></script>
```

Ahora en el archivo “googlemaps.js”, tenemos el método “SetMarkerInMap” el cual es el siguiente:

```
function SetMarkerInMap() {
    var locationMarker = [
        {"lat":40.1284,"lng": -98.4594},
        {"lat":40.6181,"lng": -99.3988},
        {"lat":40.3214,"lng":-97.3223}
    ];
};
```



```

for (var i=0; i < locationMarker.length;i++)
{
    var myLatLng = new google.maps.LatLng(locationMarker[i].lat,locationMarker[i].lng);
    var marker = new google.maps.Marker({
        position: myLatLng,
        map: map,
        draggable:true,
    });
    var mapLabel = new MapLabel({
        text: 'I am ' + i,
        position: myLatLng,
        map: map,
        fontSize: 20,
        align: 'center',
        fontColor:'#00FF00',
        strokeColor:'#D2691E'
    });
    marker.bindTo('position', mapLabel);
}
}

```

Ya hemos explicado este método en los ejemplos anteriores, aquí lo nuevo sería que después de crear el objeto LatLng y el overlay marker, creamos nuestro objeto “mapLabel” y definimos sus opciones de como debería de lucir el texto. Las opciones obligatorias para “mapLabel” son :

map : El mapa donde aparecerá el texto.

text : El texto que queremos mostrar.

position: La posición en el mapa.

Puedes consultar las demás opciones de Maplabel [aquí](#). Con esto es suficiente para mostrar texto junto con nuestros markers. Ahora con las siguientes líneas:

```
marker.bindTo('position', mapLabel);
```

Aquí vinculamos nuestro objeto “maplabel” con el marker, así cuando el marker es arrastrado, el texto también lo hará junto con él.

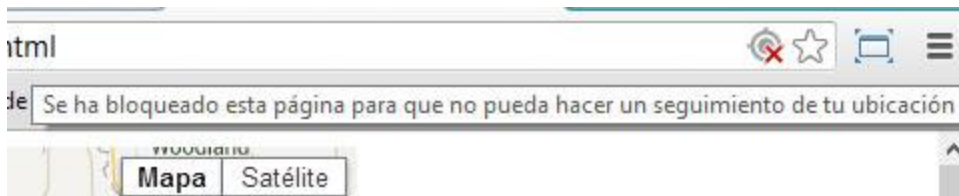
Con eso terminamos con Maplabel, que es una librería muy sencilla de usar. Existen muchas librerías en internet con las que deberías de experimentar, en el siguiente enlace puedes encontrar algunas de ellas [link](#).

Localizar al usuario

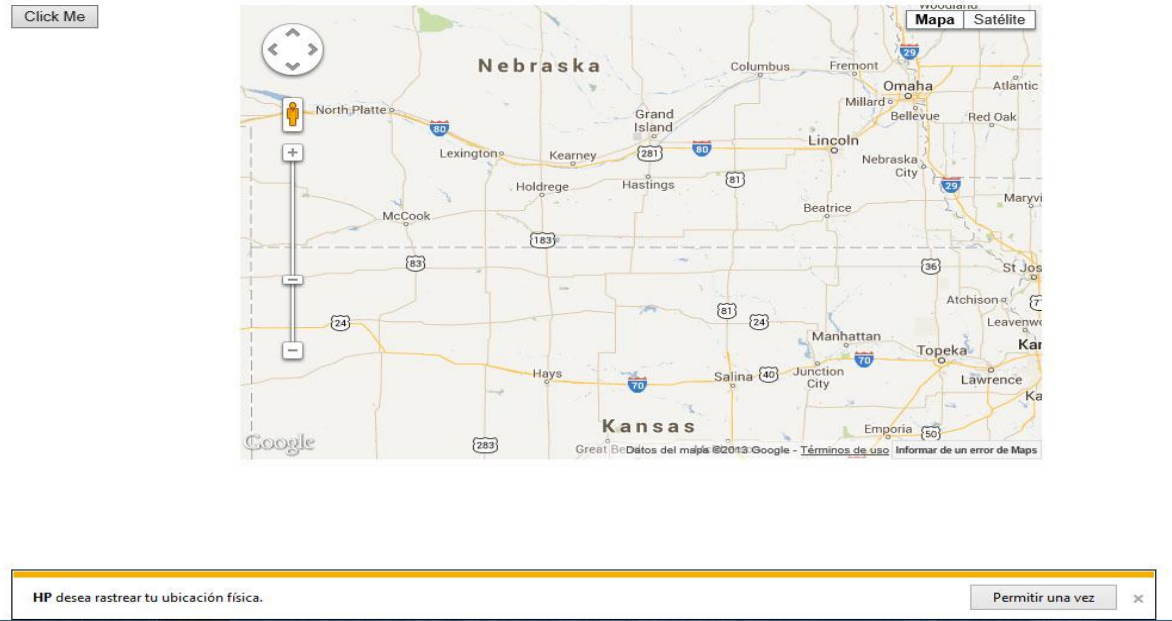
Localizar a nuestros usuarios en el mapa es una tarea muy común hoy en día. Muchas Apps en el mercado hacen uso de esta información para mostrar contenido basado en la ubicación del usuario y ofrecer una mejor experiencia.

Localizar al usuario en el mapa es muy sencillo gracias a HTML 5 y su nueva [API de geolocalización](#), haciendo uso del objeto navigator. Podemos abrir el archivo de práctica de la carpeta “19” y ejecutamos el archivo “index.html”.

En este punto hay que hacer algunas aclaraciones. Dependiendo de la configuración de privacidad del navegador del usuario, se le preguntará si quiere proporcionar su ubicación a la aplicación que la esta solicitando, en caso de que el usuario esté de acuerdo podremos tener acceso a sus coordenadas Latlng. Pero si el usuario no accede a esta confirmación es imposible saber sus coordenadas. Por ejemplo la siguiente imagen muestra como en Google Chrome se muestra cuando el usuario a negado el permiso a ser localizado.



Estas coordenadas no son la ubicación exacta del usuario, sino son coordenadas relativas del proveedor de internet del usuario, pero aun asi es una ubicación aproximada de donde podria encontrarse el usuario. En internet explorer por ejemplo esta seria la pantalla de confirmación.



Y el código que ubica al usuario es el siguiente :

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    GeoLocate();
});

function GeoLocate () {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(function (position) {
            var lat = position.coords.latitude;
            var lng = position.coords.longitude;
            var position = new google.maps.LatLng(lat,lng);
            map.setCenter(position);
        });
    }else {
        alert('Geolocation is not supported by this browser.');
```

Aquí podemos ver que cuando tenemos el mapa listo llamamos al método “Geolocate”, donde lo primero que hacemos es verificar que el objeto “navigator” de HTML 5 está disponible. Una vez confirmado llamamos al método “getCurrentPosition” que nos proporcionará las coordenadas LatLng, si te fijas bien hemos pasado un método anónimo como parámetro a este método, la razón es que “getCurrentPosition” funciona a modo asíncrono y ejecutará nuestro método cuando haya terminado, pasando para parámetro el objeto “position” donde se encuentran las coordenadas.

Después solamente resta crear nuestro objeto LatLng, para centrar nuestro mapa con el. Para saber mas informacion sobre el objeto “navigator” puedes consultar el siguiente [enlace](#).

Places Autocomplete

Una importante característica de cualquier aplicación de mapas, es la de permitir al usuario buscar direcciones o nombres de lugares y ubicarlos en el mapa. En Google Maps existe una librería llamada “[place](#)” de la que podemos hacer uso para buscar lugares , países y puntos de interés definidos en la API. Digo definidos en la API, por que si el termino de busqueda no esta registrado en la API el usuario no obtendrá resultados. Por esta razón no funciona muy bien en todos los países, pero funciona perfecto en países donde Google Maps está presente.

Algunos de los resultados que podemos obtener pueden ser establecimientos (como restaurantes, tiendas, oficinas) y resultados de "codificación geográfica", que indican direcciones, regiones políticas (como pueblos y ciudades) y otros puntos en la zona.

Para continuar podemos abrir nuestro archivo de práctica en la carpeta número “20” y ejecutar el archivo “index.html”, donde encontraremos nuestro mapa con una caja de texto para nuestras búsquedas, este será nuestro archivo base para comenzar a trabajar.

Lo primero que tienes que notar es que en el archivo “index.html”, hemos cargado la librería “places” para poder utilizarla , ya que la librería no es cargada por defecto en Google Maps.

```
<script type="text/javascript"
  src="http://maps.googleapis.com/maps/api/js?sensor=false&libraries=places">
</script>
```

Y en nuestro código html también tenemos un elemento input con el id “searchInput”. Lo primero que haremos es crear nuestro autocomplete en el evento “idle” del mapa.

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    var input = document.getElementById("searchInput");
    var autocomplete = new google.maps.places.Autocomplete(input);
    autocomplete.setBounds(map.getBounds());
});
```

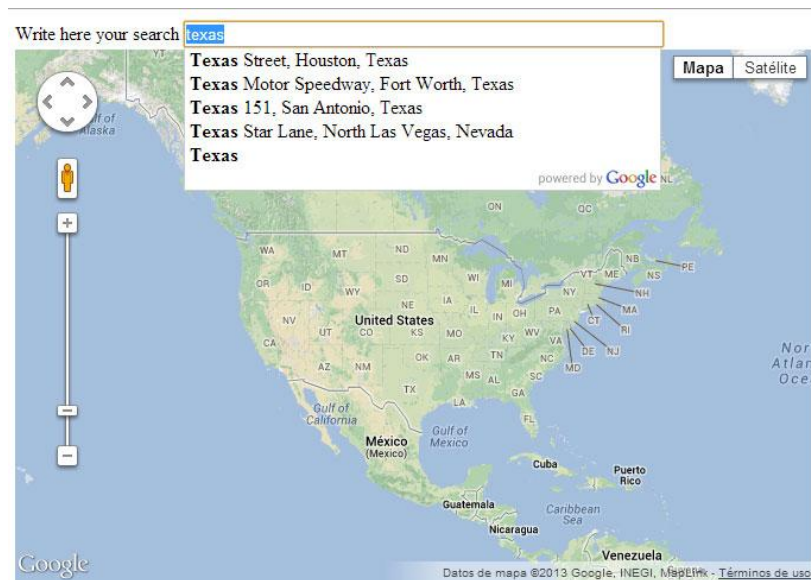
Con estas 3 líneas basta para crear nuestro autocomplete, si refrescamos nuestra página y escribimos en el input por ejemplo “texas” veremos como abajo del input aparece una lista de resultados basado en el texto de búsqueda.

Lo primero que hacemos en la primera línea es seleccionar el elemento input HTML de tipo texto que se utilizará para escribir los terminos de busqueda. En la segunda línea creamos una instancia de la clase “[google.maps.places.Autocomplete](#)” que espera como parámetro nuestro input.

`Autocomplete(inputField:HTMLInputElement, opts?:AutocompleteOptions)`

Con la tercera línea de código, definimos el área de búsqueda en la cual se basará el servicio “places” para mostrar sus resultados, pero no se restringe obligatoriamente sólo a esta zona logrando mostrar resultados adicionales fuera de esta. Para definir el área usamos el método “setBounds” del autocomplete, que recibe como parámetro un bounds de la zona en la que basará sus resultados, Por lo general siempre es el viewport del mapa.

`setBounds(bounds:LatLngBounds)`



También puedes probar escribir direcciones como por ejemplo “4441 Collins Avenue Miami Beach ” o “Eiffel Tower” o “Torre Eiffel”.

Por ahora solo hemos creado el autocomplete, pero sí seleccionamos algunos de los resultados de la lista notarás que no pasa nada en nuestro mapa. Esto es debido a que es nuestro trabajo decidir qué hacer con el resultado que el usuario elija.

Lo siguiente que haremos, es que al seleccionar un resultado de la lista ubicaremos el viewport del mapa en ese lugar. Nuestro objeto autocomplete tiene un evento que se dispara cada vez que un resultado es seleccionado por el usuario, es en este evento que necesitamos ejecutar nuestras acciones. Para esto usamos el siguiente código:

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    var input = document.getElementById("searchInput");
    var autocomplete = new google.maps.places.Autocomplete(input);
    autocomplete.setBounds(map.getBounds());
});
```

```

google.maps.event.addListener(autocomplete, 'place_changed', function() {
    var place = autocomplete.getPlace();
    if (place.geometry.viewport) {
        map.fitBounds(place.geometry.viewport);
    } else {
        map.setCenter(place.geometry.location);
    }
});
});

```

Ok en este código hemos creado un evento que será ejecutado cada vez que ocurra un “place_change” del objeto autocomplete, lo que ocurrirá cada vez que el usuario seleccione un resultado de la lista.

Cada vez que este evento ocurre, podemos tener acceso a un objeto llamado “[placeResult](#)” a través del método “getPlace()” del objeto autocomplete. El cual guardamos en una variable llamada “place”, la cual contiene toda la información sobre el lugar, nosotros en nuestro ejemplo sólo necesitamos la localización del resultado seleccionado, pero este objeto contiene mucha información relevante sobre el lugar como por ejemplo: fotos , críticas , rating , etc. puedes consultar la API para tener mayor detalle sobre esto.

En el siguiente código :

```

if (place.geometry.viewport) {
    map.fitBounds(place.geometry.viewport);
} else {
    map.setCenter(place.geometry.location);
}

```

El objeto “geometry” de “placeResult” es donde se almacenan las coordenadas que necesitamos para ubicar al usuario, este objeto contiene “viewport” que es un bounds y “location” qué es el Latlng del lugar.

En nuestro código, verificamos que “place.geometry.viewport” esta disponible en caso contrario usamos “location” en su lugar. Esto es debido a que “viewport” en algunas ocasiones devuelve NULL si no es conocido el bounds del lugar. Por eso yo personalmente uso “location” pero quería demostrar el uso de los 2 elementos.

Una vez que sabemos cual objeto esta disponible, solo nos resta usar el método apropiado del objeto mapa para ubicar al usuario, ya sea con bounds o con Latlng. Si lo prefieres puedes crear un marker y ubicarlo en el lugar usando las coordenadas Latlng de “location”.

Puedes probar nuestro ejemplo completo en la carpeta “21” y veras como el mapa es ubicado según el resultado seleccionado.

Usando Panoramio

Una interesante librería que podemos usar con Google Maps es la de [panoramio](#), la cual nos permite cargar imágenes en nuestro mapa sobre una zona concreta, lo que resulta muy útil para entregar más información sobre el lugar. Insertar la capa de panoramio es muy sencillo solo tenemos que hacer uso de la clase “[google.maps.panoramio](#)”.

Podemos abrir nuestro archivo de practica numero “22” y seguidamente abrimos el “index.html”, el cual nos cargara nuestro mapa y enseguida veremos muchos iconos de imágenes aparecer en el mapa.



Cada imagen que aparece en el mapa tiene una geolocalización que fue registrada en panoramio por cada usuario que sube una imagen, por lo cual podemos estar seguros que esa imagen corresponde al lugar correcto (en caso de que el usuario no se ha equivocado). Si recorres el mapa sobre distintos países, veras como se cargan imágenes de acuerdo a la zona donde te encuentres y al hacer click sobre el icono se abrirá una infowindows con una la imagen un poco más grande.

Algunas veces puede ser un poco molesto tener estos iconos siempre activos en el mapa, por tal razón es buena idea desactivar la capa panoramio, esto lo puedes lograr haciendo click en el botón “click me” del lado izquierdo. Ahora veamos un poco el código :

Primero en nuestro en nuestro archivo “index.html” hemos cargado la librería de panoramio.

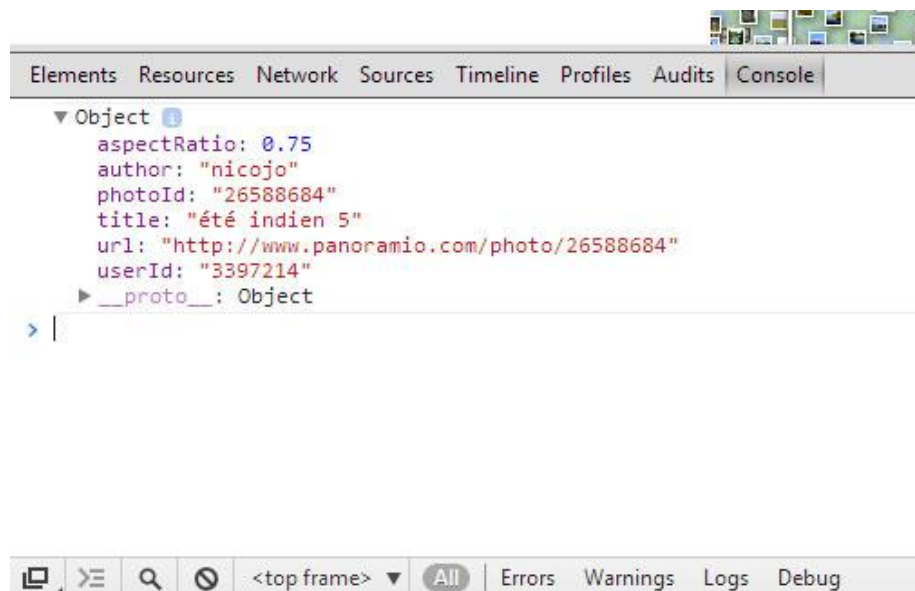
```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?sensor=false&libraries=panoramio"></script>
```


Después en el archivo “googlemaps.js” :

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    panoramioLayer = new google.maps.panoramio.PanoramioLayer();
    panoramioLayer.setMap(map);
    google.maps.event.addListener(panoramioLayer, 'click', function(event) {
        console.log(event.featureDetails);
    });
});
```

Ok como lo hemos venido haciendo en otros ejemplos, primero hacemos una instancia de la clase “[google.maps.panoramio.PanoramioLayer](#)” que guardamos en una variable global llamada “panoramioLayer”, después con el método “setMap()” del objeto panoramio asignamos en que mapa queremos que la capa panoramio aparezca. Hasta aquí con eso es suficiente para que los iconos de las imágenes se muestran en nuestro mapa.

El bloque de código donde creamos un evento “click” para el objeto “panoramioLayer” es para demostrar, de qué manera podemos tener acceso a la información de la imagen a la cual el usuario ha hecho click. Al hacer click en cualquiera de los iconos tenemos acceso a un objeto llamado “[panoramioFeature](#)”, el cual nos entrega cierta información de la imagen. En nuestro código de ejemplo solo estamos imprimiendo en consola el objeto. Para verlo tienes que entrar en modo consola en tu navegador y veras el objeto. Por ejemplo en Google Chrome usas (CTRL + SHIFT + i), Aquí un screenshot del objeto:



Tu puedes decidir qué hacer con esta información de cada imagen, a la cual se le ha hecho click. Y por último el código que oculta la capa panoramio :

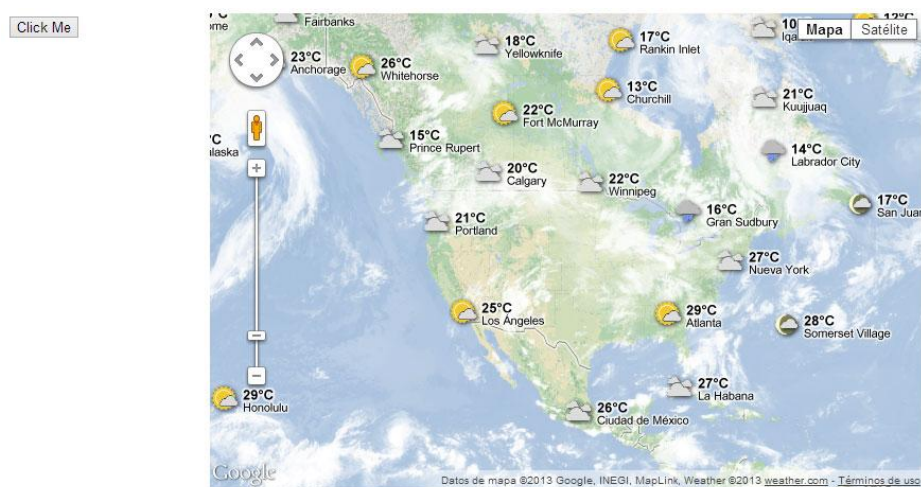
```
$('#actionButton').on('click',function(){
    if(panoramioLayer.getMap()){
        panoramioLayer.setMap(null);
    } else{
        panoramioLayer.setMap(map);
    }
});
```

No hay mucho que explicar en este bloque de código, solamente verificamos si la capa panoramio tiene un mapa asociado, si es así la desactivamos en caso contrario le asignamos el mapa nuevamente.

Mostrando el clima

Otra librería muy útil es la de meteorológica y nubes ó “google.maps.weather”, que nos permite mostrar datos meteorológicos o imágenes de nubes en nuestro mapa. Para no repetir información sobre esta librería, te invito a que consultes su documentación en la API para mayor informacion [aquí](#), en la cual podras encontrar informacion tambien sobre la capa panoramio y otras más.

Ahora podemos abrir nuestro archivo de practica numero “23” y abrimos nuestro “index.html”, el cual funciona muy similar a nuestro ejemplo de panoramio. Ahora tenemos en nuestro mapa iconos que son datos meteorológicos y nubes de las zonas en el mapa.



Igualmente puedes moverte por el mapa y notarás como la información es actualizada en la interfaz en la zonas que visites. Esta información es basada en el servicio de <http://www.weather.com/>. Ahora pasemos al código :

En nuestro “index.html” hemos cargado la librería “weather”

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?sensor=false&libraries=weather"></script>
```

Y en nuestro archivo “googlemaps.js” tenemos lo siguiente:

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    weatherLayer = new google.maps.weather.WeatherLayer({
        temperatureUnits: google.maps.weather.TemperatureUnit.CELSIUS
    });
    weatherLayer.setMap(map);
    cloudLayer = new google.maps.weather.CloudLayer();
    cloudLayer.setMap(map);
    google.maps.event.addListener(weatherLayer, 'click', function(event) {
        console.log(event.featureDetails);
    });
});
```

Con la librería “weather” tenemos acceso a 2 tipos de capas: la “[google.maps.weather.CloudLayer](#)” capa de nubes y “[google.maps.weather.WeatherLayer](#)” capa de datos meteorológicos, las cuales usamos para nuestro ejemplo.

Estas dos capas las instanciamos para crear nuestros objetos y las guardamos en variables globales, luego las asignamos en que mapa queremos que sean mostradas. Con la capa meteorológica hemos usado la opción “temperatureUnits” para especificar que tipo de unidad de temperatura queremos usar basado en el objeto “[google.maps.weather.TemperatureUnit](#)”.

Puedes hacer click en los iconos meteorológicos para tener acceso a información más detallada. En nuestro código, hemos creado un evento click para nuestro objeto “weatherLayer” para demostrar que tipo de información tenemos acceso, igualmente puedes verla en el modo consola de tu navegador.

Y por ultimo si haces click en el botón izquierdo ocultaras o mostraras las capas como en el ejemplo anterior.

Reverse Geocoding

Para entender lo que es Reverse Geocoding, primero tenemos que saber que es Geocoding. Geocoding en Google Maps es cuando escribes una dirección de algún lugar por ejemplo y con esa dirección buscamos un punto Latlng en el mapa para ubicar al usuario, eso es lo que hemos estado haciendo en los ejemplos de autocomplete usando la librería “places”.

Reverse Geocoding es exactamente lo contrario en lugar de buscar un punto Latlng basado en una dirección, vamos a buscar una dirección usando un punto LatLng del mapa. Para realizar nuestro Reverse Geocoding usaremos la clase “[google.maps.Geocoder](#)” para consultar el servicio de codificación geográfica de Google Maps.

Ok vamos manos a obra, podemos abrir nuestro archivo de práctica “24” que contiene el ejemplo base con el comenzaremos este ejercicio. Lo primero a notar es que tenemos nuestro mapa y sobre él tenemos un input con un botón que dicen “search”. En este ejemplo he puesto un valor por defecto de un punto Latlng en el input, que será el que usaremos para buscar nuestra dirección.

Para este ejemplo vamos a usar un marker, para localizar en el mapa el punto Latlng del cual se quiere buscar la dirección y una infowindows para mostrar el resultado del reverse geocoding. Primero vamos crear la infowindow y el objeto “geocoder” para poder hacer nuestra consulta al servicio de codificación geográfica.

```
google.maps.event.addListenerOnce(map, 'idle', function(){
    infowindow = new google.maps.InfoWindow();
    geocoder = new google.maps.Geocoder();
});
```

En este bloque de código hemos creado los objetos “infowindow” y “geocoder” instanciando las clases correspondientes y guardarlas en variables globales, Hasta aquí nada fuera de lo normal. Ahora necesitamos ejecutar nuestro metodo de busqueda cuando el usuario haga click en el botón con el texto “search”, para esto solo creamos un evento click con jquery en el botón que tiene el id “search” y ejecutamos nuestro método de búsqueda, en este caso se llamará “SearchAddress”.

```
$('#search').on('click',function(){
    SearchAddress($('#searchInput').val());
});
```

Al hacer click en el botón “search”, se ejecutará el método “SearchAddress” al cual se le pasará como parámetro el texto que sea introducido en el input de lado izquierdo del botón usando el método “[val\(\)](#)” de jquery para obtener ese valor. Ahora vamos a la definición de nuestro método

“search”

```
function SearchAddress(latlngUser) {
    var latlngArray = latlngUser.split(",");
    var lat = parseFloat(latlngArray[0]);
    var lng = parseFloat(latlngArray[1]);
}
```

En este método estamos recibiendo el texto del input y lo hemos procesado para obtener los valores, en este caso hemos obtenido los valores del texto separados por comas y convertido en un arreglo con el método [“split”](#) de javascript y después convertimos estos valores de cadena en números con el método [“parseFloat”](#).

El objeto “geocoder” que hemos creado previamente en el evento “idle” del mapa, lo usaremos para hacer nuestra consulta al servicio de codificación geográfica. Para hacer esto usamos su unico metodo disponible [“geocode\(\)”](#) el cual recibe solo 2 parámetros: un objeto llamado “GeocoderRequest” y un método callback que se ejecutará cuando la consulta termine, debido a que este método funciona a modo asíncrono.

```
geocode (request:GeocoderRequest, callback:function());
```

Entonces lo primero que tenemos que hacer es crear nuestro objeto [“GeocoderRequest”](#) el cual tiene 4 propiedades, de las cuales nosotros solo vamos usar una llamada “location” para hacer nuestra búsqueda basado en un LatLng.

```
function SearchAddress(latlngUser) {
    var latlngArray = latlngUser.split(",");
    var lat = parseFloat(latlngArray[0]);
    var lng = parseFloat(latlngArray[1]);

    var geocoderRequest = {
        location: new google.maps.LatLng(lat,lng)
    }
}
```

Con los valores anteriores que habíamos guardado creamos un objeto LatLng para asignarlo en la propiedad “location” que usaremos. Ahora solo nos resta llamar al método “geocode”, para hacer la consulta al servicio de codificación geográfica.

```
function SearchAddress(latlngUser) {
    var latlngArray = latlngUser.split(",");
    var lat = parseFloat(latlngArray[0]);
    var lng = parseFloat(latlngArray[1]);
```

```

var geocoderRequest = {
    location: new google.maps.LatLng(lat,lng)
}
geocoder.geocode(geocoderRequest, function(results, status) {
    //código para manejar la respuesta
});
}

```

Con esto estamos haciendo una consulta al servicio de codificación geográfica y pasamos nuestro método callback que se ejecutará cuando el servicio termine. Ahora veremos como manejar la respuesta del servicio. El método “geocode” pasa 2 valores al método callback cuando este devuelve una respuesta, que son “results” y “status”.

status : Este valor nos indica el estado de la consulta, si fue exitosa o no. Los posibles valores que puede contener esta definidos en la clase “[google.maps.GeocoderStatus](#)”, como constantes de la clase, las cuales podemos usar para saber si la consulta fue exitosa. Nosotros usaremos el valor de la constante “OK” para saber si consulta se ejecutó correctamente.

```

function SearchAddress(latlngUser) {
    var latlngArray = latlngUser.split(",");
    var lat = parseFloat(latlngArray[0]);
    var lng = parseFloat(latlngArray[1]);
    var geocoderRequest = {
        location: new google.maps.LatLng(lat,lng)
    }
    geocoder.geocode(geocoderRequest, function(results, status) {
        if (status == google.maps.GeocoderStatus.OK) {

        }

    });
}

```

Recuerda que si el estado de la consulta no es “OK”, tienes que tener alguna manera de informar al usuario que ha ocurrido algún error.

Ahora vamos hablar de “results”, el segundo valor que recibimos en nuestra callback. El cual contiene toda la información del geocoding de la consulta. Es un objeto JSON que contiene más de un resultado, por lo tanto viene en forma de un arreglo. La razón por la cual devuelve más de un resultado nos la ofrece Google Maps en su documentación que dice :

“ El geocoder inverso a menudo devuelve más de un resultado. Las "direcciones" de codificación geográfica no son solo direcciones postales, sino cualquier forma de definir de forma geográfica una ubicación. Por ejemplo, al codificar de forma geográfica un punto de la

ciudad de Chicago, el punto codificado se puede etiquetar como una dirección postal, como la ciudad (Chicago), como su estado (Illinois) o como un país (Estados Unidos). Todas ellas son direcciones para el geocoder. El geocoder inverso devuelve todos esos resultados. El geocoder inverso encuentra coincidencias con entidades políticas (países, provincias, ciudades y barrios), con direcciones y con códigos postales.

Las direcciones se devuelven ordenadas de mayor a menor coincidencia. Normalmente, la dirección más exacta es el resultado más destacado. Ten en cuenta que se devuelven diferentes tipos de direcciones, desde la dirección postal más específica hasta entidades políticas menos específicas como barrios, ciudades, países, estados, etc”

Si quieres leer un poco más de la documentación oficial puedes visitar este [link](#)

Cada elemento del arreglo es un objeto “[google.maps.GeocoderResult](#)”, el cual contiene propiedades que podemos usar para saber la dirección que estamos buscando. Vamos a ver una breve descripción de cada propiedad, pero en la API puedes obtener más información en este [enlace](#).

- **types** : Este es un arreglo que contiene, qué tipo de localización fue devuelto en el resultado. Puede ser por ejemplo un país ó una ciudad. para saber más puedes ver este [link](#)
- **formatted_address**: Esta es una cadena de texto que tiene la información que pueden leer los usuarios más información en este [link](#)
- **address_components** : Es un conjunto que incluye los diferentes componentes de la dirección más información en este [link](#).
- **geometry** : Este es un objeto que tiene muchas propiedades, pero la más importante es “location” el cual contiene la position en un objeto LatLng de la dirección, para más información de este objeto visita este [link](#).

Ahora ya que sabemos que el método “geocoder” puede devolver más de un resultado, lo único que nos queda es confiar en que el primero resultado del arreglo es el más relevante entre todos, recuerda que los resultados se devuelven de mayor a menor coincidencia.

Ahora que hemos leído toda esta información vamos a modificar nuestro método de esta manera :

```
function SearchAddress(latlngUser) {
    var latlngArray = latlngUser.split(",");
    var lat = parseFloat(latlngArray[0]);
    var lng = parseFloat(latlngArray[1]);
    var geocoderRequest = {
        location: new google.maps.LatLng(lat,lng)
    }
}
```

```

geocoder.geocode(geocoderRequest, function(results, status) {
    if (status == google.maps.GeocoderStatus.OK) {
        marker = new google.maps.Marker({
            position: results[0].geometry.location,
            map: map
        });
        infowindow.setContent(results[0].formatted_address);
        infowindow.open(map, marker);
    }
});
}

```

Una vez que hemos verificado el estatus, creamos un objeto marker al cual le asignamos la posición usando “location” del objeto geometry, recuerda que es el que contiene las coordenadas de la dirección, después mostramos la dirección encontrada en nuestra infowindow usando la propiedad “formatted_address”, la cual es un formato que puede leer el usuario. En este código hemos usado el primero valor devuelto del arreglo en results[0].

Bueno y eso sería todo para que nuestro servicio de Reverse Geocoding funcione. Si quieres ver el ejemplo completo, puedes abrir el archivo de ejemplo de la carpeta número “25”.

El objeto geocoder también puede ser usado de la misma manera que usamos el “autoComplete”, si quieres buscar puntos LatLng basado en una dirección solo tienes que usar la propiedad “address” del objeto “GeocoderRequest” en lugar de “location” como lo hemos hecho.

Static Maps API

Hasta ahora hemos creado mapas dinámicos usando la API javascript de Google Maps, pero también tenemos a nuestra disposición la [Static Map API](#) , que nos brinda la posibilidad de insertar un mapa en nuestra web usando una simple imagen. La diferencia es que sera una imagen comun y corriente que solo mostrara la localización en el mapa. Esta API también tiene sus límites de uso por lo cual recomiendo que leas un poco sobre esto en la [documentacion](#).

Static Maps API nos permite construir una URL la cual nos devolverá una imagen, en esta URL podemos especificar la localización del mapa, el tamaño de la imagen, el formato de la imagen y ubicar algunos markers, la estructura de una URL sería de esta manera:

`http://maps.googleapis.com/maps/api/staticmap?parameters`

Donde “paramaters” pueden ser cualquiera de los especificados en este [link](#), siendo la mayoría opcionales , los únicos parámetros obligatorios son :

center : Define el centro del mapa , basado un punto Latlng separados por comas por ejemplo : "40.714728,-73.998672" o una dirección "city hall, new york, ny" recuerda que las direcciones no funcionan en todos los países por eso es recomendable usar Latlng.

zoom : Nivel de zoom del mapa basados en los nivel de la API javascript de Google Maps.

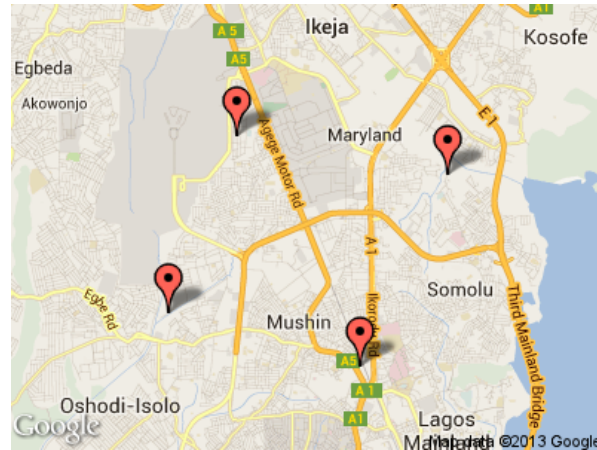
size: El tamaño deseado de la imagen, en el siguiente formato {horizontal_value}x{vertical_value} por ejemplo : “300x200”

sensor : Al igual que la API javascript de Google Maps es usado para determinar si la aplicación geolocalizara al usuario.

Con estos parámetros podemos crear una imagen estática, la cual podemos insertar en cualquier sitio web por medio de la etiqueta con su atributo “src” y mostrar un mapa sencillo sin usar código javascript ni la API de Google Maps, lo que resulta útil para pequeños proyectos web. Ejemplo:

`http://maps.googleapis.com/maps/api/staticmap?center=6.5597,203.3579&zoom=12&size=400x300&sensor=false&format=png32&markers=6.5353,3.3201|6.523,3.3643|6.5670,3.3846|6.5759,3.3358`

Con esta URL estamos creando la siguiente imagen:



Solo hemos visto lo básico para crear una imagen estática simple, pero te invito a que revises la documentación de [Static Map API](#) para consultar parámetros adicional que puedes usar.

Capítulo 5 : Guardar y cargar tu overlays

Hasta este punto hemos aprendido, como cargar un mapa, como crear Markers y overlays, localizar al usuario entre otras cosas. El siguiente paso en cualquier proyecto web basado en mapas, es guardar toda la información creada por el usuario. Esta información por lo general tiene que ser almacenada en una base de datos del servidor, en este capítulo no abordaremos los temas de como se guarda la información en el lado del servidor, en su lugar veremos cómo preparar la información para enviarla al servidor por medio de [AJAX](#).

Al proceso de preparar la información antes de enviarla al servidor para que sea guardada se le conoce como “serialización”.

Para comenzar este capítulo necesitamos un punto de partida, para esto vamos a usar el archivo de práctica de la carpeta número “26”, donde podemos ejecutar el archivo “index.html”. Al cargar nuestro ejemplo vemos que tenemos un mapa sencillo, al lado izquierdo tenemos un panel vacío, también tenemos cargada la “Drawing Library” con la cual podemos dibujar todos los tipos de overlays que queramos, los cuales tienes habilitado su modo edición en las opciones de nuestra “Drawing Library”. Este será nuestro punto de partida. La idea es que el usuario pueda dibujar cualquier overlay en cualquier lugar del mapa y nuestro trabajo será guardar toda esa información.

El primer paso a seguir es identificar qué tipo de overlay el usuario ha dibujado en el mapa, en dependencia del tipo, podremos saber cómo guardar la información. Para esto crearemos un evento “overlaycomplete” para el objeto “Drawing Library” el cual se ejecuta cada vez que un overlay es dibujado.

```
// when the map is ready
google.maps.event.addListenerOnce(map, 'idle', function() {
    var drawingManager = new google.maps.drawing.DrawingManager({
        drawingControlOptions: {
            position: google.maps.ControlPosition.TOP_CENTER
        },
        circleOptions: {editable: true },
        polygonOptions: {editable: true },
        polylineOptions: {editable: true },
        rectangleOptions: {editable: true },
        markerOptions : {draggable:true},
        map: map
    });
    google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {

    });
});
```

En este evento podemos escribir nuestro código para saber que tipo de overlay se ha dibujado, a partir de este momento nos centraremos en este método por lo que no haré referencia al código encargado de crear la “Drawing overlay”.

Cuando el evento “overlaycomplete” es ejecutado, pasa como parámetro a nuestra callback un objeto llamado “event”, el cual contiene 2 propiedades “type” y “overlay”. “type” nos indica el nombre del overlay creado y “overlay” es el objeto overlay creado. Sabiendo esto podemos añadir la siguiente línea de código:

```
google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
    switch(event.type)
    {
        case "marker":
            alert('I am a marker');
            break;
        case "polyline":
            alert('I am a polyline');
            break;
        case "rectangle":
            alert('I am a rectangle');
            break;
        case "circle":
            alert('I am a circle');
            break;
        case "polygon":
            alert('I am a polygon');
            break;
    }
});
```

Con este [switch](#) y usando la propiedad “event.type”, podremos saber exactamente cual overlay fue dibujado por el usuario. Primero vamos a serializar el overlay del tipo Marker.

Serializar Marker

Serializar es el proceso de organizar la información que necesitamos guardar antes de enviarla al servidor, por lo tanto esta sección vamos a serializar un marker. Primero tenemos que hacernos la siguiente pregunta :

¿ Qué información necesitamos para recrear un marker ?

El LatLng para saber donde ubicarlo y sus opciones, en este caso para nuestro ejemplo solo sera la propiedad “editable”.

Con esta respuesta en mente ahora vamos a crear nuestro método para serializar un maker.

```
google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
    switch(event.type)
    {
        case "marker":
            SerializeMarker(event.overlay);
            break;
        case "polyline":
            alert('I am a polyline');
            break;
        case "rectangle":
            alert('I am a rectangle');
            break;
        case "circle":
            alert('I am a circle');
            break;
        case "polygon":
            alert('I am a polygon');
            break;
    }
});
```

El método que usaremos será “serializeMarker”, al cual le pasaremos como parámetro el overlay que ha sido creado por el usuario que nos proporciona la propiedad “event.overlay”, ahora definiremos nuestro método, el cual podremos ubicar fuera del evento ready de JQuery.

```
function SerializeMarker(marker) {
    var object = {};
    object.position = marker.getPosition().toUrlValue();
    object.type = "marker";
    object.draggable = marker.getDraggable();
    return object;
}
```

Ahora nuestro método nos devolverá un objeto, con las propiedades principales para recrear el overlay marker que son “position”, si es “draggable” y que tipo de overlay es con “type”.

Hemos usado cada uno de los métodos del overlay tipo [marker](#) para obtener los datos que necesitamos, por ejemplo “getPosition” que nos devuelve el [LatLng](#) de la posición actual del marker. Una vez tenemos el [LatLng](#) del marker podemos encadenar el método “toUrlValue” del objeto [LatLng](#) para convertir este dato a una cadena, el cual sera mas facil de almacenar.

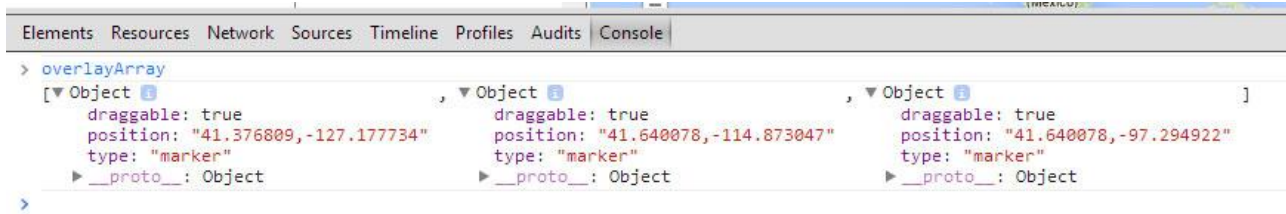
Después usamos “getDraggable” que nos devuelve true ó false, en este caso será true para saber si el marker será editable o no. Nuestro método devolverá un objeto similar a este :

```
Object {
  position: "52.05249,-106.259766",
  type: "marker",
  editable: true
}
```

Una vez que nuestro método “SerializeMarker”, nos devuelve el objeto producto de nuestra serialización, tenemos que guardarlo en algun lugar. Para esto vamos usar un arreglo global llamado “overlayArray” el cual vamos a definir al inicio de nuestro archivo “googlemaps.js”, una vez hecho esto solo tenemos que usar el método [push](#) javascript para guardar nuestro objeto en este arreglo :

```
google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
  switch(event.type)
  {
    case "marker":
      overlayArray.push(SerializeMarker(event.overlay));
      break;
    case "polyline":
      alert('I am a polyline');
      break;
    case "rectangle":
      alert('I am a rectangle');
      break;
    case "circle":
      alert('I am a circle');
      break;
    case "polygon":
      alert('I am a polygon');
      break;
  }
});
```

Ahora supongamos que dibujamos 3 overlays marker en nuestro mapa y consultamos la variable “overlayArray” en el modo consola de Google Chrome, obtendremos algo como esto:

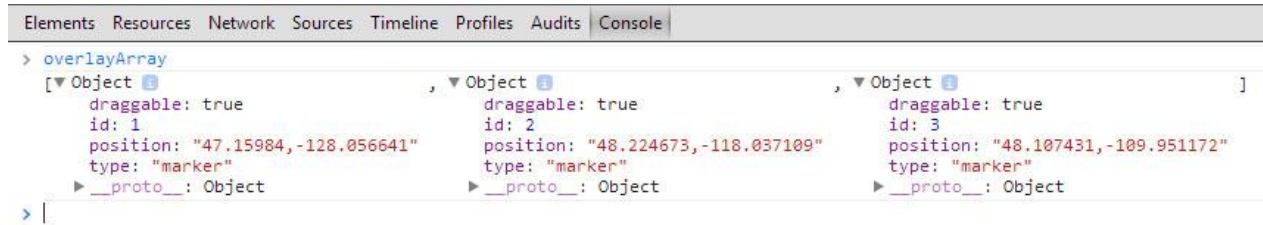


Cada elemento del arreglo “overlayArray”, será un objeto serializado por nosotros. Lo siguiente que necesitamos es una manera de identificar cada elemento del arreglo de forma individual, esto lo podemos conseguir usando un ID para identificar cada overlay. Para esto por ejemplo podemos crear otra variable global llamada “idCounter” la cual debemos inicializar en “0” y será nuestro contador que aumentaremos en 1 cada vez que un overlay sea creado.

```
google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
    idCounter = idCounter + 1;
    switch(event.type)
    {
        case "marker":
            overlayArray.push(SerializeMarker(event.overlay,idCounter));
            break;
        case "polyline":
            alert('I am a polyline');
            break;
        case "rectangle":
            alert('I am a rectangle');
            break;
        case "circle":
            alert('I am a circle');
            break;
        case "polygon":
            alert('I am a polygon');
            break;
    }
});

function SerializeMarker(marker,id) {
    var object = {};
    object.position = marker.getPosition().toUrlValue();
    object.type = "marker";
    object.id = id;
    object.draggable = marker.getDraggable();
    return object;
}
```

Como puedes ver, he actualizado el método “SerializeMarker” para que reciba un segundo parámetro que será nuestro ID que identificara nuestro objeto, en la siguiente imagen podemos ver un ejemplo en modo consola de Google Chrome del objeto overlayArray .



Ahora cada elemento del arreglo “overlayArray”, puede ser identificado por la propiedad ID. Y con esto sería todo para serializar nuestro marker.

Serializar rectangle

Ahora vamos a serializar el overlay rectangle, para poder recrear este overlay hacemos la misma pregunta.

¿ Qué información necesitamos para recrear un rectangle ?

Este overlay es muy sencillo, ya que solo necesitamos saber el “[bounds](#)” del overlay del tipo rectangle para poderlo recrearlo y sus opciones.

```

google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
    idCounter = idCounter + 1;
    switch(event.type)
    {
        case "marker":
            overlayArray.push(SerializeMarker(event.overlay,idCounter));
            break;
        case "polyline":
            alert('I am a polyline');
            break;
        case "rectangle":
            overlayArray.push(SerializeRectangle(event.overlay,idCounter));
            break;
        case "circle":
            alert('I am a circle');
            break;
        case "polygon":
            alert('I am a polygon');
            break;
    }
}

```

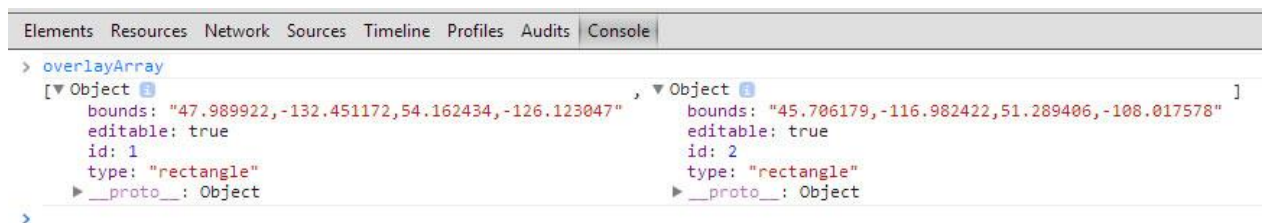
```

    }
  });

function SerializeRectangle(rectangle,id) {
    var object = {};
    object.bounds = rectangle.getBounds().toUrlValue();
    object.type = "rectangle";
    object.id = id;
    object.editable = rectangle.getEditable();
    return object;
}

```

Esta es la misma dinámica en como trabaja nuestro método anterior “serializeMarker”, la diferencia es en los métodos usados para crear nuestro objeto serializado. El método “getBounds” del overlay tipo “rectangle” nos devuelve un “[LatLngBounds](#)” y cuando usamos el método “toUrlValue” nos devuelve una cadena con los valores de ese bounds en el siguiente orden : “lat_lo , lng_lo , lat_hi , lng_hi ” donde la terminación “lo” se refiere al suroeste y “hi” al noreste. En modo consola nuestro objeto serializado sería similar a este ejemplo:



Serializar polyline

Los overlay del tipo polyline deben ser serializados con un paso adicional, debido a que su trazado es definido por un arreglo de puntos LatLng, los cuales tenemos que serializar por separado. Para comenzar vamos definir nuestro método para serializar una polyline.


```

google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
    idCounter = idCounter + 1;
    switch(event.type)
    {
        case "marker":
            overlayArray.push(SerializeMarker(event.overlay,idCounter));
            break;
        case "polyline":
            overlayArray.push(SerializePolyline(event.overlay,idCounter));
            break;
        case "rectangle":
            overlayArray.push(SerializeRectangle(event.overlay,idCounter));
            break;
        case "circle":
            alert('I am a circle');
            break;
        case "polygon":
            alert('I am a polygon');
            break;
    }
});
function SerializePolyline(polyline,id) {
    var object = {};
    object.path = '';
    object.type = "polyline";
    object.id = id;
    object.draggable = polyline.getEditable();
    return object;
}

```

Hemos definido nuestro método llamado “SerializePolyline” y dentro de él tenemos los mismos pasos anteriores con (id , type y draggable), lo único adicional aquí es la propiedad llamada “path”, la cual usaremos para guardar el path ó trazado de la polyline. Debemos recordar que el path de una polyline es un arreglo de puntos LatLng, por lo tanto tenemos que serializarlo por separado, crearemos un método adicional que se encargue de esta operación por nosotros a continuación.

```

function SerializeMvcArray(mvcArray) {
    var path = [];
    for(var i= 0; i < mvcArray.getLength(); i++)
    {
        path.push(mvcArray.getAt(i).toUrlValue());
    }
    return path.toString();
}

```

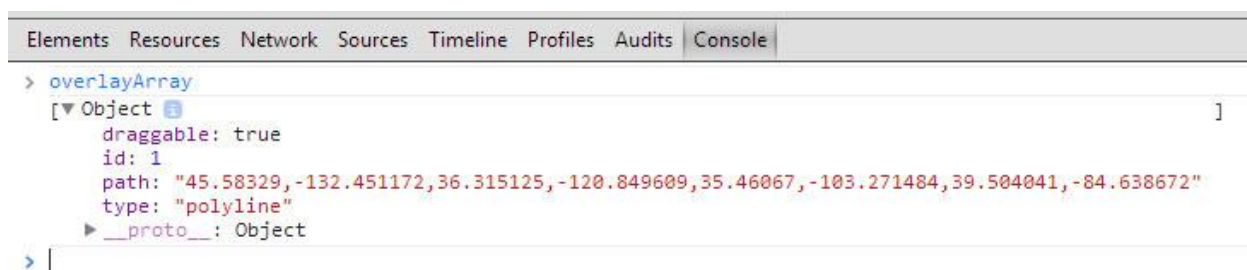
Este será nuestro método para serializar un [MVCArray](#) . ¿Por que un MVCArray? recuerda que para obtener el path de una polyline debemos utilizar el método “getPath” del objeto polyline, el cual nos devuelve un MVCArray.

En este método vamos a recorrer cada elemento del MVCArray por medio de un ciclo FOR, con el método “getLength” del objeto MVCArray obtenemos cuántos elementos existen en el arreglo. Y para obtener cada elemento individual del [MVCArray](#) usamos el método “getAt”, que recibe sólo el índice del elemento, el cual nos devolverá un objeto LatLng, el que seguidamente convertimos a cadena usando “toUrlValue”.

En cada iteración del ciclo FOR guardamos nuestro valor de cadena en un arreglo llamado “path” y cuando el ciclo FOR a terminado el método devolverá esos valores en formato cadena usando el método “toString”. Con nuestro método creado solo nos resta hacer el llamado desde nuestro metodo “SerializePolyline”.

```
function SerializePolyline(polyline,id) {
    var object = {};
    object.path = SerializeMvcArray(polyline.getPath());
    object.type = "polyline";
    object.id = id;
    object.draggable = polyline.getEditable();
    return object;
}
function SerializeMvcArray(mvcArray) {
    var path = [];
    for(var i= 0; i < mvcArray.getLength(); i++)
    {
        path.push(mvcArray.getAt(i).toUrlValue());
    }
    return path.toString();
}
```

Ahora la propiedad “path” de nuestro objeto serializado contendrá el trazado ó path de la polyline en formato de texto. Este es un ejemplo del objeto serializado:



Serializar polygon

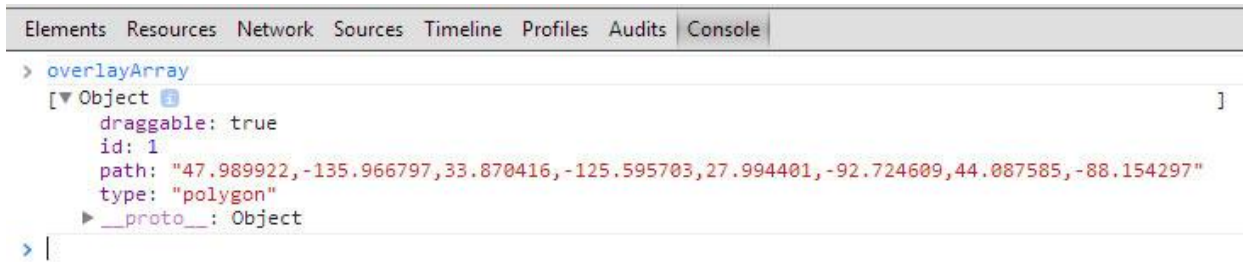
Serializar un overlay del tipo polygon es muy parecido al método usado para polyline, la única diferencia es que la polyline es una serie de puntos LatLng que determinan un trayecto abierto, mientras que un polygon es una serie de puntos LatLng que definen una región cerrada. La API de Google Maps automáticamente unirá el punto final con el inicial para cerrar el polygon. Con esto en mente podemos definir nuestro método:

```
google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
    idCounter = idCounter + 1;
    switch(event.type)
    {
        case "marker":
            overlayArray.push(SerializeMarker(event.overlay,idCounter));
            break;
        case "polyline":
            overlayArray.push(SerializePolyline(event.overlay,idCounter));
            break;
        case "rectangle":
            overlayArray.push(SerializeRectangle(event.overlay,idCounter));
            break;
        case "circle":
            alert('I am a circle');
            break;
        case "polygon":
            overlayArray.push(SerializePolygon(event.overlay,idCounter));
            break;
    }
});

function SerializePolygon(polygon,id) {
    var object = {};
    object.path = SerializeMvcArray(polygon.getPath());
    object.type = "polygon";
    object.id = id;
    object.draggable = polygon.getEditable();
    return object;
}
```

El funcionamiento es idéntico que el método para serializar un polyline, dado que el método para obtener el path de un [polygon](#) tiene el mismo nombre en la API y devuelve también un MVCArray, por lo tanto necesitamos hacer uso de “SerializeMvcArray” que habíamos creado anteriormente.

Con esto será suficiente, a continuación un ejemplo de como sería el objeto serializado:



Un dato importante que debes recordar es que en nuestro ejemplo estamos usando el metodo “getPath” del objeto polygon para obtener su path, por es un polygon simple cerrado. Pero un polygon también puede estar formado por más de una path, por ejemplo para los polygon huecos, en este caso tendrias que usar el método “getPaths” para obtener todas las paths relacionadas.

Serializar circle

El overlay tipo circle solo necesita 2 propiedades para ser creados que son center y radius, los cuales son fáciles de obtener por medio de los métodos del objeto [circle](#).

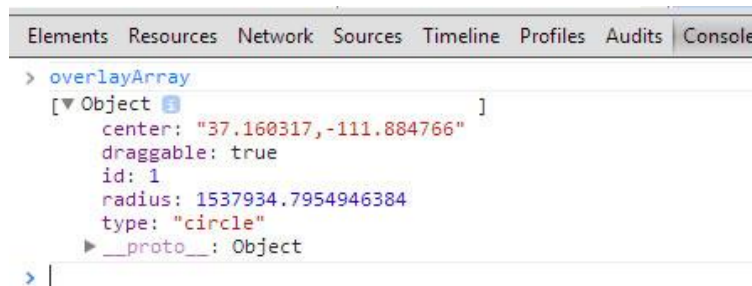
```

google.maps.event.addListener(drawingManager, 'overlaycomplete', function(event) {
  idCounter = idCounter + 1;
  switch(event.type)
  {
    case "marker":
      overlayArray.push(SerializeMarker(event.overlay,idCounter));
      break;
    case "polyline":
      overlayArray.push(SerializePolyline(event.overlay,idCounter));
      break;
    case "rectangle":
      overlayArray.push(SerializeRectangle(event.overlay,idCounter));
      break;
    case "circle":
      overlayArray.push(SerializeCircle(event.overlay,idCounter));
      break;
    case "polygon":
      overlayArray.push(SerializePolygon(event.overlay,idCounter));
      break;
  }
});

```

```
function SerializeCircle(circle,id) {
    var object = {};
    object.center = circle.getCenter().toUrlValue();
    object.radius = circle.getRadius();
    object.type = "circle";
    object.id = id;
    object.draggable = circle.getEditable();
    return object;
}
```

Con el método “getCenter” obtenemos el centro del overlay que nos devuelve un LatLng y con “getRadius” obtenemos el radio del overlay circle en metros. A continuación un ejemplo del objeto:



Serializar el mapa

Además de los overlay es conveniente serializar el estado actual del mapa, por ejemplo podemos guardar en que zona estaba el usuario, que nivel de zoom tenia en ese momento y que tipo de mapa tenia seleccionado.

```
function SerializeMap() {
    var object = {};
    object.zoom = map.getZoom();
    object.bounds = map.getBounds().toUrlValue();
    object.mapType = map.getMapTypeId();
    object.type = 'map';
    return object;
}
```

Este es nuestro método para serializar nuestro objeto mapa, no recibe ningún parámetro ya que nuestro mapa esta guardado en la variable global “map”. Como siempre usamos los métodos disponibles del objeto [map](#) para obtener los datos que necesitamos.

“getZoom” para obtener el zoom , “getBounds” para saber en que zona esta el usuario en ese momento, recuerda el bounds del objeto map se refiere al viewport del mapa y “getMapTypeId”

para el tipo de mapa seleccionado.

El mejor lugar para llamar a este método, es un botón que tenemos disponible llamado “save” con el id “save” en un nuestro “index.html”, que creo habrás notado en nuestro ejemplo en el lado izquierdo.

```
$('#save').on('click',function(){
    var data = {};
    data.map = SerializeMap();
    data.overlays = overlayArray;
});
```

La razón por la que hacemos el llamado a este metodo aqui es por que la accion del boton “save” es la de guardar toda la información en la base de datos, por lo tanto solo necesitamos serializar nuestro objeto mapa una sola vez.

Aquí creamos un nuevo objeto llamado “data” para organizar mejor la información antes de enviarla al servidor, nuestro método “SerializeMap” devuelve el objeto mapa serializado, el cual lo guardamos en la propiedad llamada “map” y nuestro arreglo de overlays los guardaremos en una propiedad llamada “overlays”.

Ahora solo nos basta enviar esta información por medio de AJAX al servidor para ser guardada, una maneras de hacer esto es usando el método [POST de JQuery](#).

```
$('#save').on('click',function(){
    var data = {};
    data.map = SerializeMap();
    data.overlays = overlayArray;
    $.post("hereyoururlscriptserverside",data);
});
```

Como guardar los datos en el lado del servidor esta fuera del ámbito de este libro, pero en resumen el script recibirá los datos por medio del método post, de esta manera puedes procesarlos y guardarlos en tu base de datos.

Bueno hemos terminado de serializar todos los overlays y el mapa. Para ver el ejemplo completo debes abrir el archivo de práctica número “27”. Adicional a este ejemplo podrás encontrar otro en la carpeta número “28”, el cual tiene el mismo ejemplo pero con ligeras modificaciones para que cada overlay pueda aceptar un título y una descripción escrita por el usuario, te invito que analises el codigo y como practica puedas mejorarlo un poco mas.

Siguiente paso

Suponiendo que los datos enviados al servidor fueron guardados con éxito en una base de datos, ahora viene la segunda parte de este proceso, recrear todos los overlays y nuestro mapa basados en estos datos, básicamente el proceso será inverso, a esto se le llama : Deserialización.

Los datos almacenados en la base de datos por lo general son obtenidos por medio de un método AJAX o algún servicio web, el cual debe devolver un objeto JSON de preferencia, con todos los datos correspondientes. Para los siguientes ejemplos vamos a suponer que el método AJAX nos ha devuelto el siguiente objeto basado en los ejemplos anteriores.

```
{
  "overlays":[{"position":"","type":"","id":"","draggable":"","title":"","description":""}],
  "map":{"zoom":"","bounds":"","mapType":"","type":"map"}
}
```

Los datos de los overlay se encuentran en la propiedad “overlays”, que es un arreglo de objetos y los datos de nuestro mapa se encuentran en la propiedad “map”. Vamos a crear un método llamado “FakeAjax” el cual nos devolverá este objeto simulando un llamado AJAX al servidor, Para los siguientes ejemplos vamos usar como punto de partida el archivo de práctica número “29” el cual contiene creado el método “FakeAjax”.

Normalmente los objetos JSON que son devueltos como respuesta del servidor, vienen en formato cadena por lo tanto tienes que parsear esa cadena para convertirla a un objeto Javascript por ejemplo usando el método [jQuery.parseJSON\(\)](#)

Deserializar Marker

En nuestro archivo de practica numero “29” tenemos el método “FakeAjax”, que nos devuelve un objeto con toda la información que necesitamos recrear. El cual ejecutamos en el evento “idle” de nuestro mapa.

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var data = FakeAjax();
});
```

Ahora necesitamos iterar sobre el arreglo de objetos que contiene la información que necesitamos en la propiedad “overlays”.

```

google.maps.event.addListenerOnce(map, 'idle', function() {
    var data = FakeAjax();
    for (var i = 0 ; i < data.overlays.length ; i++)
    {
        switch(data.overlays[i].type)
        {
            case "marker":
                //marker
                break;
            case "polyline":
                //polyline
                break;
            case "rectangle":
                //rectanlge
                break;
            case "circle":
                //cirlce
                break;
            case "polygon":
                //polygon
                break
        }
    }
});

```

Usando un ciclo FOR iteramos sobre el arreglo de objetos y para cada elemento del arreglo usamos su propiedad “type”, para saber que tipo de overlay se necesita deserializar usando un switch javascript. Ahora vamos a crear un método llamado “DeserializeMaker”.


```

google.maps.event.addListenerOnce(map, 'idle', function() {
    var data = FakeAjax();
    for (var i = 0 ; i < data.overlays.length ; i++)
    {
        var objectDeserialized;
        switch(data.overlays[i].type)
        {
            case "marker":
                objectDeserialized = DeserializeMaker(data.overlays[i]);
                break;
            case "polyline":
                //polyline
                break;
            case "rectangle":
                //rectanlge
                break;
            case "circle":
                //cirlce
                break;
            case "polygon":
                //polygon
                break;
        }
        objectDeserialized.setMap(map);
    }
});

function DeserializeMaker (object) {
    var position = object.position.split(',');
    var latLng = new google.maps.LatLng(position[0],position[1]);
    var marker = new google.maps.Marker({
        position: latLng,
        draggable:object.draggable
    });
    return marker;
}

```

En nuestro método “DeserializeMarker” estamos recibiendo como parámetro el elemento del arreglo actual sobre el cual se está iterando, este objeto tiene las propiedades que previamente habíamos serializado para recrear el marker, en este caso solo es “position” el cual es una cadena de texto con los valores LatLng de la position del marker separados por coma, usando el método [split](#) podemos obtener estos datos como un arreglo. Recuerda que el primer valor es Lat y el segundo es Lng. Este es un ejemplo del objeto que estamos recibiendo:

```
Object {
  position: "-39.198205,-56.162109",
  type: "marker",
  id: 1,
  draggable: true
}
```

Después creamos nuestros objetos [LatLng](#) y [Marker](#) usando las clases de Google Maps, pasando como parámetro los valores obtenidos de la propiedad “position”. Nuestro método devuelve el objeto marker creado, el cual es asignado en la variable “objectDeserialized”, en esta variable asignaremos cada objeto creado por nuestros métodos y al final de cada ciclo FOR usaremos el método “setMap” para asignar el objeto creado al mapa ya que todos los overlays comparten el este metodo.

Deserializar rectangle

Con el overlay rectangle vamos a usar el método “DeserializeRectangle”. A partir de ahora solo haré referencia al código referente al switch para evitar repetir líneas de código continuamente para mostrar los siguientes ejemplos. Ya que el resto de código será el mismo.

```
switch(data.overlays[i].type)
{
  case "marker":
    objectDeserialized = DeserializeMaker(data.overlays[i]);
    break;
  case "polyline":
    //polyline
    break;
  case "rectangle":
    objectDeserialized = DeserializeRectangle(data.overlays[i]);
    break;
  case "circle":
    //circle
    break;
  case "polygon":
    //polygon
    break;
}
```

```
function DeserializeRectangle (object) {
    var bounds = object.bounds.split(',');
    var swLatLng = new google.maps.LatLng(bounds[0],bounds[1]);
    var neLatLng = new google.maps.LatLng(bounds[2],bounds[3]);
    var rectangleBounds = new google.maps.LatLngBounds(swLatLng,neLatLng);
    var rectangle = new google.maps.Rectangle({
        bounds: rectangleBounds,
        editable:object.editable
    });
    return rectangle;
}
```

Nuestro método está recibiendo como parámetro el siguiente objeto :

```
Object {
  bounds: "-31.615966,-43.813477,-27.80021,-38.803711",
  type: "rectangle",
  id: 2,
  editable: true
}
```

La propiedad “bounds” contiene las coordenadas LatLng en formato cadena necesarios para crear el objeto bounds que necesitamos para recrear nuestro overlay, usando el método split convertimos los datos a un arreglo y los usamos para crear los objetos LatLng. Recuerda que estos puntos vienen en el siguiente orden : “lat_lo, lng_lo, lat_hi ,lng_hi” después de haber usado “[toUrlValue](#)” para serializarlo.

Con estos puntos LatLngs creamos nuestro objeto “bounds”, el que es usado para crear nuestro overlay.

Deserializar Polyline

Para el overlay Polyline usaremos el método “DeserializePolyline”

```
switch(data.overlays[i].type)
{
    case "marker":
        objectDeserialized = DeserializeMaker(data.overlays[i]);
        break;
    case "polyline":
        objectDeserialized = DeserializePolyline(data.overlays[i]);
        break;
    case "rectangle":
        objectDeserialized = DeserializeRectangle(data.overlays[i]);
        break;
    case "circle":
        //circle
        break;
    case "polygon":
        //polygon
        break;
}
```

Este método recibirá el siguiente objeto como parámetro :

```
Object {
  path: "-39.605688,-49.042969,-39.707187,.....",
  type: "polyline",
  id: 3,
  draggable: true
}
```

La propiedad “path” contiene todos nuestros puntos separados por comas en formato cadena, por lo tanto debemos de crear a partir de ellos un MVCArray par recrear nuestro path.

El método encargado para de esta tarea será “DeserializeMvcArray” :

```
function DeserializeMvcArray(stringLatLng) {
    var arrayPoints = stringLatLng.split(',');
    var mvcArray = new google.maps.MVCArray();
    for(var i= 0; i < arrayPoints.length; i+=2)
    {
        var latLng = new google.maps.LatLng(arrayPoints[i],arrayPoints[i+1]);
        mvcArray.push(latLng);
    }
    return mvcArray;
}
```

Este método recibe el path en formato cadena, después creamos un arreglo a partir de esta cadena con el método split. Usando la clase [MVCArray](#) creamos un objeto, al cual dentro del ciclo FOR, agregamos cada punto LatLng creado, por medio del método [push](#). También nota que nuestro ciclo for itera cada 2 elementos en el arreglo, esto es debido a que el primer elemento es Lat y el siguiente Lng y así sucesivamente, al terminar el ciclo for regresará el objeto MVCArray Creado. Ahora solo nos resta incluir este método en “DeserializePolyline”.

```
function DeserializePolyline (object) {
    var mvcArray = DeserializeMvcArray(object.path);
    var polyline = new google.maps.Polyline({
        path: mvcArray,
        draggable:object.draggable,
        editable:true
    });
    return polyline;
}
```

En este método sólo nos resta pasar nuestro mvcArray creado a nuestro overlay para crear la polyline.

Deserializar Polygon

Para polygon utilizaremos de nuevo el método “DeserializeMvcArray”, debido a que el path del objeto serializado polygon también es una cadena con los valores separados por comas.

```
Object {
  path: "-36.421282,-50.581055,-37.822802....",
  type: "polygon",
  id: 5,
  draggable: true
}
```

Crearemos un método llamado “DeserializePolygon” :

```
switch(data.overlays[i].type)
{
  case "marker":
    objectDeserialized = DeserializeMaker(data.overlays[i]);
    break;
  case "polyline":
    objectDeserialized = DeserializePolyline(data.overlays[i]);
    break;
  case "rectangle":
    objectDeserialized = DeserializeRectangle(data.overlays[i]);
    break;
  case "circle":
    //circle
    break;
  case "polygon":
    objectDeserialized = DeserializePolygon(data.overlays[i]);
    break;
}

function DeserializePolygon (object) {
  var mvcArray = DeserializeMvcArray(object.path);
  var polygon = new google.maps.Polygon({
    paths: mvcArray,
    draggable:object.draggable,
    editable:true
  });
  return polygon;
}
```

Nuestro método recibe el objeto serializado y crea el MVCArray usando la propiedad path con el método “DeserializeMvcArray”, el que después pasamos como parámetro para crear nuestro overlay.

Deserializar Circle

Para circle usaremos el método llamado “DeserializeCircle” y este es el objeto que recibirá este método :

```
Object {
  center: "-29.688053,-56.513672",
  radius: 388514.2326,
  type: "circle",
  id: 4,
  draggable: true
}
```

Y en nuestro switch

```
switch(data.overlays[i].type)
{
  case "marker":
    objectDeserialized = DeserializeMaker(data.overlays[i]);
    break;
  case "polyline":
    objectDeserialized = DeserializePolyline(data.overlays[i]);
    break;
  case "rectangle":
    objectDeserialized = DeserializeRectangle(data.overlays[i]);
    break;
  case "circle":
    objectDeserialized = DeserializeCircle(data.overlays[i]);
    break;
  case "polygon":
    objectDeserialized = DeserializePolygon(data.overlays[i]);
    break;
}
```

```
function DeserializeCircle (object) {
    var center = object.center.split(',');
    var centerLatLng = new google.maps.LatLng(center[0],center[1]);
    var circle = new google.maps.Circle({
        center: centerLatLng,
        radius: object.radius,
        draggable:object.draggable,
        editable:true
    });
    return circle;
}
```

La propiedad “center” es la que contiene nuestro punto LatLng en formato cadena, la cual convertimos a un objeto LatLng Google Maps la cual usamos en las opciones junto con “radius”.

Actualizar Map

Hasta ahora hemos deserializado todos los tipos de overlays, pero todavía nos queda actualizar el mapa con el nivel de zoom , tipo de mapa y el lugar exacto donde estaba el usuario cuando los datos fueron guardados. Usaremos el método llamado “UpdateMap”.

```
google.maps.event.addListenerOnce(map, 'idle', function() {
    var data = FakeAjax();
    for (var i=0;i < data.overlays.length;i++)
    {
        var objectDeserialized;
        switch(data.overlays[i].type)
        {
            case "marker":
                objectDeserialized = DeserializeMaker(data.overlays[i]);
                break;
            case "polyline":
                objectDeserialized = DeserializePolyline(data.overlays[i]);
                break;
            case "rectangle":
                objectDeserialized = DeserializeRectangle(data.overlays[i]);
                break;
            case "circle":
                objectDeserialized = DeserializeCircle(data.overlays[i]);
                break;
            case "polygon":
                objectDeserialized = DeserializePolygon(data.overlays[i]);
                break;
        }
    }
})
```



```

        objectDeserialized.setMap(map);
    }
    UpdateMap(data.map);
});

```

“UpdateMap” es llamado fuera del ciclo FOR debido a que los datos que necesitamos no se encuentran en un arreglo, sino en un objeto simple en la propiedad “map” con las siguientes propiedades:

```

Object {
  zoom: 5,
  bounds: "-41.261381,-63.325123,-24.786843,-33.793873",
  mapType: "hybrid",
  type: "map"
}

```

Y nuestro método es el siguiente:

```

function UpdateMap (object){
    var bounds = object.bounds.split(',');
    var swLatLng = new google.maps.LatLng(bounds[0],bounds[1]);
    var neLatLng = new google.maps.LatLng(bounds[2],bounds[3]);
    var mapBounds = new google.maps.LatLngBounds(swLatLng,neLatLng);
    map.fitBounds(mapBounds);
    map.setZoom(object.zoom);
    map.setMapTypeId(google.maps.MapTypeId[object.mapType.toUpperCase()]);
}

```

Como lo hemos hecho anteriormente primero creamos nuestro objeto bounds a partir de la propiedad “bounds” en formato cadena, la cual asignamos con el método “fitBounds” del objeto “[map](#)”, seguido asignamos el zoom y el tipo del mapa.

Al asignar el tipo de mapa estamos usando las [constantes de mapa](#) de la API que comúnmente son de esta manera :

```

google.maps.MapType.HYBRID
google.maps.MapType.ROADMAP
google.maps.MapType.SATELLITE
google.maps.MapType.TERRAIN

```

En nuestro código es un tanto diferente ya que nos basamos en que los objetos en javascript son arreglos asociativos, por lo tanto en nuestra línea de código podemos acceder al valor por medio de su “keyname” y usando el método “[toUpperCase](#)” de javascript, nos aseguramos de que sean en letras mayúsculas. Normalmente esto podría hacerse por medio de un switch pero

creo que de esta manera nos ahorramos unos lineas de código.

Hasta aquí ha sido todo en este capítulo si quieres ver el ejemplo completo puedes abrir el archivo de práctica en la carpeta número “30”. Adicional a este tambien deberias ver el archivo de ejemplo número “31” el cual contiene una versión modificada con algunos campos adicionales y evento click en la lista de la izquierda para ubicar el overlay.

Ejemplos Adicionales

Obtener puntos Latlng contenidos dentro de un bounds

Aunque este libro no cubre lenguajes del lado del servidor, considero que el siguiente caso es muy común entre el desarrollo de aplicaciones de mapas, por tal razón he decidido incluirlo.

Habrán algunas ocasiones donde solo queremos mostrar overlays o markers que solo se encuentran dentro del actual viewport del mapa. Para lograr esto solo tienes que obtener el [bounds](#) del mapa el cual contiene un método llamado :

`bounds.contains(latLng:LatLng)` : El cual devuelve "true" si el Latlng que has pasado como parámetro se encuentra dentro de él.

Usando la API de Google Maps es realmente fácil saber esto, pero esto quiere decir que tienes que cargar todos los puntos Latlng antes en algún arreglo e iterar en cada uno. Lo conveniente sería que solo obtuvieras desde tu base de datos los puntos latlng que se encuentran dentro de este bounds.

Por ejemplo imagina que tienes en tu base de datos una tabla llamada "markers" y esta tabla contiene 2 campos llamados "Latitude" y "longitud", donde previamente han sido guardados estos valores para 1000 registros, en lugar de cargar estos mil registros de una sola vez en tu mapa, solo queremos obtener los que se encuentran dentro del bounds del mapa actual.

Para realizar esto tienes que enviar al servidor tu coordenadas del bounds actual del mapa, esto lo puedes hacer por medio del método :

`bounds.toUrlValue(precision?:number)` : que devuelve una cadena con los puntos en el siguiente orden : "lat_lo,lng_lo,lat_hi,lng_hi" .

Una vez que obtengas estos datos del lado del servidor solo tienes que ejecutar la siguiente consulta SQL, que sólo devolverá los registros con los Latlng que se encuentran dentro de bounds indicado.

```
SELECT * FROM markers
WHERE
Latitude < "lat_hi" AND
Latitude > "lat_lo" AND
Longitude < "lng_hi" AND
Longitude > "lng_lo";
```

Cálculo de áreas y distancias

Google Maps pone a nuestra disposición la librería “[Geometry Library](#)” con la cual podremos realizar cálculos de datos geométricos sobre la superficie de la Tierra. Similar a otras librerías con las que hemos trabajado anteriormente, tenemos que incluirla en nuestro proyecto de la siguiente manera:

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?sensor=false&libraries=geometry"></script>
```

El uso de los métodos de esta librería es tan sencillo como solamente llamar al método estático deseado para realizar la operación. He preparado 2 ejemplos de cómo usar esta librería para calcular áreas y distancias entre puntos en el mapa. Animo al lector a revisar el código para su comprender cómo funciona.

En el ejemplo de práctica en la carpeta número “32”, puedes encontrar un ejemplo de cómo se calcula la distancia entre 2 puntos. Al mover los markers sobre el mapa podrás ver como en el panel izquierdo se actualiza la distancia entre estos puntos, la cual es devuelta en metros.

Y en el ejemplo de práctica en la carpeta número “33”, podrás encontrar el código de ejemplo de como calcular el área de las overlays en Google Maps, las cuales son expresadas en metros cuadrados. Si dibujas una overlay, podrás ver como se actualiza el panel izquierdo con la información del área. Para las polyline se esta calculando la distancia total en metros.

Este fue el último capítulo del libro, espero que lo hayas disfrutado y encontrado útil para tus futuros proyectos web. Mi meta con este libro fue intentar proveer sólidos fundamentos sobre la API de Google Maps, para que cualquiera pueda comenzar con sus propias técnicas en el desarrollo de mapas.

¿ Con esto es suficiente ? Claro que no, todavía existe mucha información sobre Google Maps en internet, creo que un solo libro no sería suficiente, espero haber entregado los conocimientos y confianza para que puedas seguir explorando más sobre esta API. Pase un buen momento escribiendo este libro y espero en un futuro ver tus proyectos web en internet.

Como ultima recomendacion solo tengo que decir que te mantengas al dia con los cambios en la API, ya que google está en constante cambios y mejoras en la API. Si quieres estar al tanto de mis futuros proyectos y las cosas que ando haciendo por ahi puedes visitar mi blog : <http://webmasternoob.blogspot.com/>