Google Maps API, V2

Adding *Where* To Your Applications

Scott Davis

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas

Useful Friday Links

- Source code from this book and other resources.
- Free updates to this PDF
- Errata and suggestions. To report an erratum on a page, click the link in the footer.

To see what we're up to, please visit us at

http://www.pragmaticprogrammer.com

Produced in the United States of America.

Lovingly created by gerbil #32 on 2006-5-18

Contents

Chapter 1

Google Maps

The Google Maps API, version 2 (http://www.google.com/apis/maps/) is a great way to dip your toe into the world of web mapping. You don't have to worry about finding and managing your own data, installing and configuring your own server, or creating your own cross-browser AJAX mapping framework from scratch. It's a programmer's dream—with a little bit of JavaScript and a few latitude/longitude points, you are off to the races.

It is, however, aimed squarely at programmers. You will be neck-deep in JavaScript from the get-go. If all you want is a simple map without having to program it yourself, there is an easier way: Yahoo! Maps (http://developer.yahoo.net/maps/) provides a simple service that allows you to pass in an XML list of points and get a fully rendered map out on the other side. You can give it either lat/long points or plain old street addresses. Yahoo! handles everything for you—creating the points on the map, handling pop-up "bubbles" over your data points, the whole nine yards. You can pass in custom icons, html links, and descriptive text for each item in the XML. You get quite a bit of flexibility for a canned application.

The Google Maps API gives you the same functionality as Yahoo! Maps—pop-up bubbles (called *Info Windows* in Google-speak), custom icons, and so on—but you have to be much more deeply involved in the implementation. After all, Google Maps is an API, not a finished application. You get all of the pieces, but there is "some assembly required."

The upside is that you have much more control of your application. You have an event model that you can tap into—you can react to every click on the map, every drag, every zoom. You have fine-grained control of your custom icons—the image, the shadow, even specifying the tie-points to the map and the Info Window. You have access to terabytes of high-resolution satellite imagery. But the most important thing is that you have a map component that can be seamlessly embedded in your own webpage.

1.1 A Brief History

Ahh, living at the speed of the Internet – ain't it grand?

Google Maps was released in beta on February 7, 2005. The draggable map interface created a sensation. It was a "Wizard of Oz / Technicolor" moment for most web users. Who knew that a web application could be that smooth and responsive? For that matter, who knew that you could even do such a thing in a web browser?

On February 18, 2005, Jesse James Garrett of Adaptive Path published a seminal article that gave a name to this new style of web development: Ajax: A New Approach to Web Applications. Suddenly, Google Maps wasn't simply a revolutionary *mapping* application; it became the poster-child for *all* web applications. Tim O'Reilly (founder of O'Reilly Media, Inc.) coined another phrase, "Web 2.0", that helped further define the difference between how web applications used to behave versus the new "Google Maps" way.

On June 29, 2005, Google released version 1 of their Mapping API. This allowed us to move from being simply consumers of the maps to actual producers. (Presumably, it's the reason you're reading this book right now.)

On April 3, 2006, Google released version 2 of the Mapping API.

While this new version brought many exciting new features to the table (increased zoom levels, additional map controls, and the ability to overlay your own imagery on the map), it also broke compatibility with the previous version. As you read articles on the web or browse code examples, be wary of the date of publication. If you don't specifically see "v2" featured prominently, chances are good that the code simply won't work as advertised. This book has been upgraded to allow you to take full advantage of the latest release, version 2.

Now that you know the history of the API, let's dive right in with some code examples.

1.2 Here's the Game Plan

We'll start with a quick walk-through to satisfy the Attention-Deficit Disorder crowd. Then we'll swing back around cover each component in excruciating detail.

http://www.mapmap.org/googlemaps/examples.html

I encourage you to play with the maps, live, as you read along. Dizzy Gillespie once said that "reading about music is like dancing about architecture"—it loses something in the translation. Google Maps practically *beg* to be played around with. You can also download the code and run the examples on your own server.

http://www.mapmap.org/googlemaps/downloads.html

If you choose to go that route, you need to get a free key from Google at http://www.google.com/apis/maps/signup.html. This key is tied to the public URL of your webserver plus a subdirectory name. For example, the key used in the examples is bound to my server, running at http://www.mapmap.org/googlemaps. If you try to run the code on your server without changing the key, you'll be greeted by a friendly alert box from Google reminding you to apply for your own key.

Before we get too far, we should cover the lawyerly stuff. Google provides the Google Maps API for free, and in return they require that your resulting application is free as well. You can use it for commercial purposes, but you cannot charge your users to view the map. You can password protect access to it only if passwords are free and issued to anyone who asks for one. For more information, see http://www.google.com/apis/maps/faq.html.

Chapter 2

For Those in a Hurry...

Sometimes the quickest way to learn a new API is to see some code in action. In later chapters we'll talk about what this code is actually doing in greater detail. For now, let's just dive in.

2.1 The Simple Map

Let's take a look at the simplest possible Google Map application.

File 1

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="http://maps.google.com/maps?file=api&v=2&key=xyz"
            type="text/javascript"></script>
  </head>
  <body>
    <div id="map" style="width: 500px; height: 400px"></div>
    <script type="text/javascript">
        var map = new GMap2(document.getElementById("map"));

        //zoom levels 0-17+, 0 == world
        map.setCenter(new GLatLng(39.754286, -104.994637), 16);
    </script>
  </body>
</html>
```

Let's examine the interesting parts of the code:

- The <script> tag imports the Google Maps library. This is where you place your key.

- The <div> tag is the placeholder for your map. It can be named

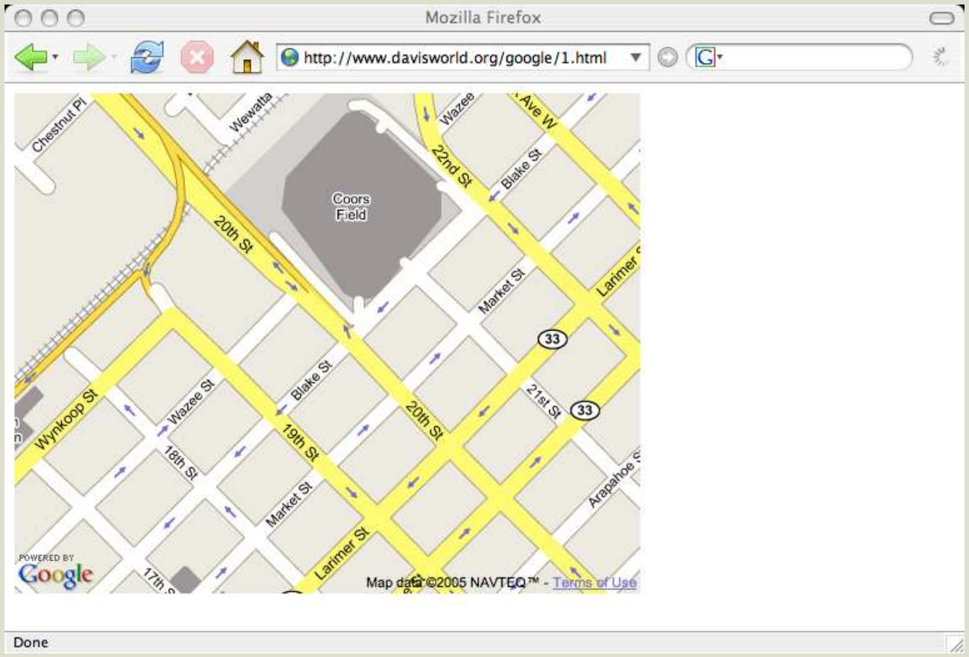


Figure 2.1: The simplest possible Google Maps application.

anything you'd like—we named it "map" (clever, eh?). In the CSS styling, we defined the size of the map in pixels.

- new GMap2() creates the map. In the constructor of the GMap2 object, we pass in the <div> using DOM.
- map.setCenter(), as the name implies, centers the map on a particular latitude/longitude point. It also zooms into the level specified. Google uses a 17 point scale; zoom level 0 will show you the entire world, zoom level 17 is zoomed into the street level.

See Figure 2.1 for the resulting map.

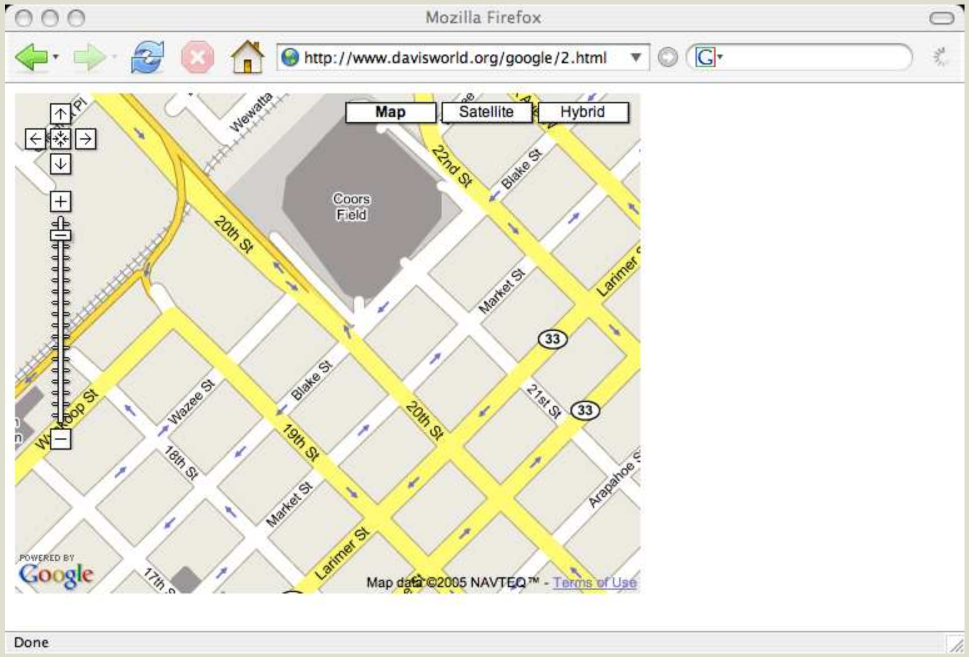

Mozilla Firefox
http://www.davisworld.org/google/2.html
Map
Satellite
Hybrid
Coors Field
Map data ©2005 NAVTEQ™ - Terms of Use
Done

Figure 2.2: Adding map controls to your map.

2.2 Adding navigation components

Now let's add some simple components to the map that will allow us to zoom and change the map type, as shown in Figure 2.2 .

File 2

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="http://maps.google.com/maps?file=api&v=2&key=xyz"
            type="text/javascript"></script>
  </head>
  <body>
    <div id="map" style="width: 500px; height: 400px"></div>
    <script type="text/javascript">
    var map = new GMap2(document.getElementById("map"));
    map.setCenter(new GLatLng(39.754286, -104.994637), 16);

    map.addControl(new GLargeMapControl());
    map.addControl(new GMapTypeControl());
    </script>
  </body>
</html>
```

Let's examine the interesting parts of the code:

- The GLargeMapControl allows the user change the zoom level of the map.
- The GMapTypeControl allows the user to flip between the Map, Satellite, and Hybrid views.

2.3 Setting the initial map type

In addition to letting the user change map types, you can set the initial type programmatically.

File 3

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="http://maps.google.com/maps?file=api&v=2&key=xyz"
            type="text/javascript"></script>
  </head>
  <body>
    <div id="map" style="width: 500px; height: 400px"></div>
    <script type="text/javascript">
    var map = new GMap2(document.getElementById("map"));
    map.setCenter(new GLatLng(39.754286, -104.994637), 16);
    map.addControl(new GLargeMapControl());
    map.addControl(new GMapTypeControl());
```

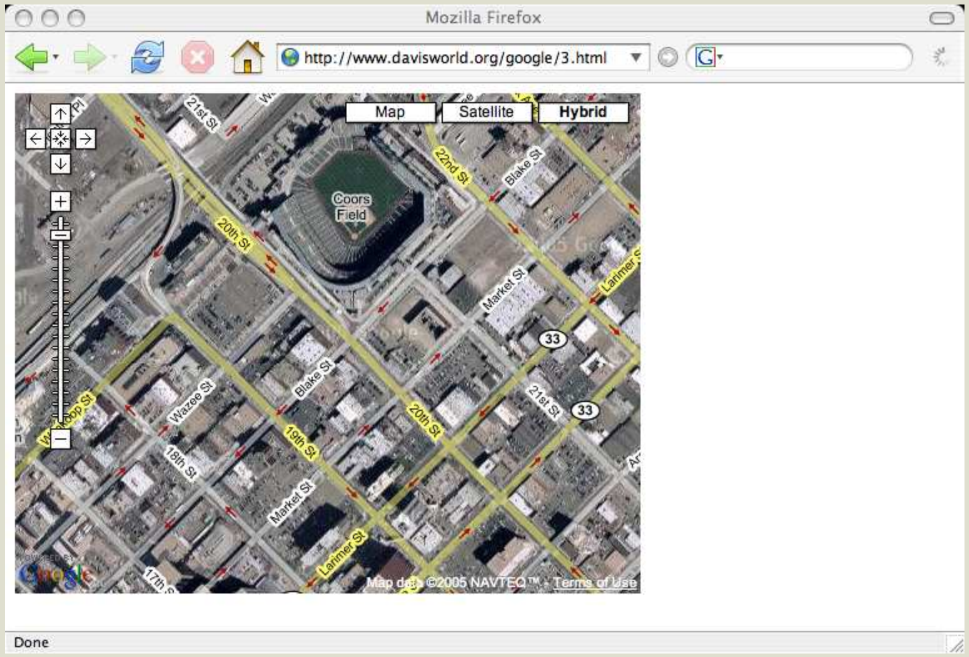

Mozilla Firefox
http://www.davisworld.org/google/3.html
Map
Satellite
Hybrid
Coors Field
Map data ©2005 NAVTEQ™ - Terms of Use
Done

Figure 2.3: Changing the MapType of your map.

```
    // map.setMapType(G_NORMAL_MAP);
    // map.setMapType(G_SATELLITE_MAP);
    map.setMapType(G_HYBRID_MAP);
    </script>
  </body>
</html>
```

Let's examine the interesting parts of the code:

- map.setMapType() sets the default view of the map: one of G_NORMAL_MAP, G_SATELLITE_MAP, or G_HYBRID_MAP.

See Figure 2.3 for the resulting map.

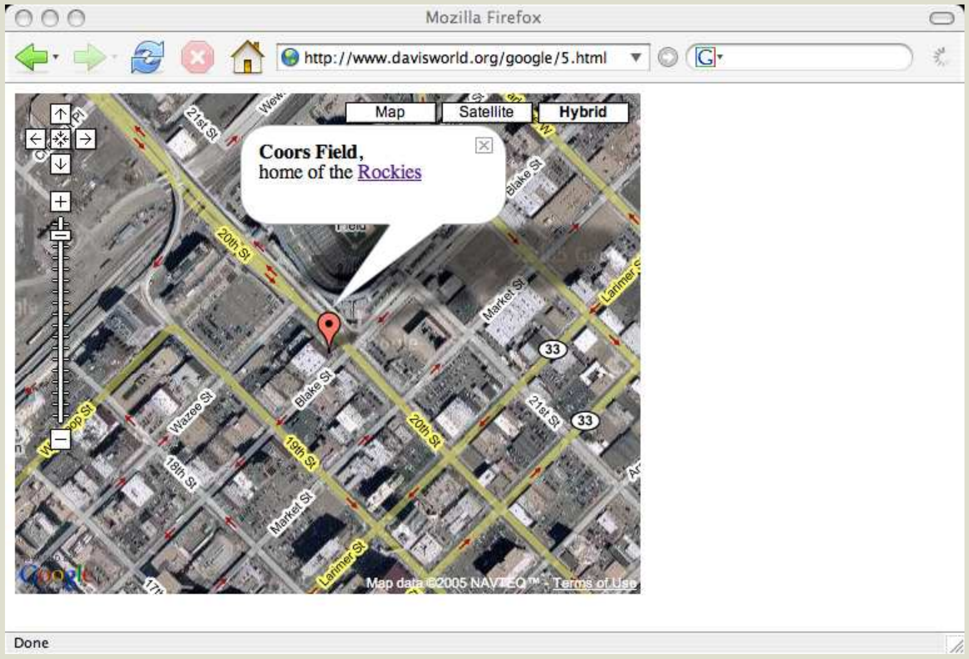

Mozilla Firefox
http://www.davisworld.org/google/5.html
Map
Satellite
Hybrid
Coors Field,
home of the Rockies
Map data ©2005 NAVTEQ™ - Terms of Use
Done

Figure 2.4: Adding user data to your map.

2.4 Creating a Point and an Info Window

In this final example, we'll create a point on the map and display an Info Window (see Figure 2.4).

File 4

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="http://maps.google.com/maps?file=api&v=2&key=xyz"
            type="text/javascript"></script>
  </head>
  <body>
    <div id="map" style="width: 500px; height: 400px"></div>
    <script type="text/javascript">
    var map = new GMap2(document.getElementById("map"));
    var coorsFieldPoint = new GLatLng(39.754286, -104.994637);
    map.setCenter(coorsFieldPoint, 16);
    map.addControl(new GLargeMapControl());
    map.addControl(new GMapTypeControl());
    map.setMapType(G_HYBRID_MAP);

    //create marker, add to map overlay
    var coorsFieldMarker = new GMarker(coorsFieldPoint);
    map.addOverlay(coorsFieldMarker);

    //create Info Window (html)
    var coorsFieldHtml = '<b>Coors Field</b>,<br /> home of the ' +
            '<a href="http://www.coloradorockies.com">Rockies</a>';
    coorsFieldMarker.openInfoWindowHtml(coorsFieldHtml);
    </script>
  </body>
</html>
```

Let's examine the interesting parts of the code:

- We could have passed an anonymous point as the first parameter to map.setCenter(). Instead we created a GLatLng called coorsFieldPoint that can be reused later in the code.
- new GMarker() creates a "pushpin" for coorsFieldPoint.
- map.addOverlay() adds the GMarker to the map.
- We create a variable that holds an arbitrary string of HTML. Then we display an Info Window on the GMarker using the HTML string from the previous line.

Chapter 3

The Excruciating Details

Now that you have a better idea of what can be done with the Google Maps API, let's go back and look at things in greater details. We have only scratched the surface up to this point.

The Google Maps API is made up of 20 basic objects. The objects fall into five categories: Core, Map Controls, User Data, Events, and AJAX.

3.1 Core Objects

The core objects are the basic building blocks of a map.

- GMap2
- GLatLng
- GLatLngBounds

3.2 Map Controls

Map Control objects allow the user to manipulate the map.

- GLargeMapControl
- GSmallMapControl
- GSmallZoomControl
- GMapTypeControl
- GScaleControl
- GOverviewMapControl
- GControlPosition
- GSize

- GMapType

3.3 User Data

User Data objects allow the developer to add custom data to the map.

- GMarker
- GIcon
- GInfoWindowTab
- GPolyline

3.4 Events

Event objects allow the developer to react to user actions, such as drags and clicks.

- GEvent

3.5 AJAX

AJAX objects allow the developer to make asynchronous calls to web services and update the display without having to refresh the entire page.

- GXmlHttp
- GXml
- GDownloadUrl

Chapter 4

Core Objects

GMap2

When Google upgraded their API from v1 to v2, they claimed that is was "designed to be 99% backwards-compatible with the previous version of the API". While that's a nice thought, the reality of the situation is many things changed. *Many* things... (See the 6 page upgrade document for details.)

One of the most significant changes was the move from a GMap object to a GMap2. Although the original GMap is preserved in the new API, all of the cool stuff requires you to use a GMap2. And more than a simple name change, Google changed the constructor as well as many of the key methods hanging off of it.

If you see code examples that use the GMap object, chances are good that they are a bit long in the tooth. Caveat emptor.

The Core objects are the basic building blocks of your map. While you may not use AJAX or work with GEvents in every application, you'd be hard pressed to avoid using these elements.

4.1 GMap2

A *GMap2* object, not surprisingly, is your map. You can have as many GMap2s as you'd like on your page.

The GMap2 constructor has one required argument—container. This is an HTML div. The id attribute of the div is the unique name of the map. By default, the GMap2 uses the size of the div specified in the style attribute to determine the size of the map.

```
<div id="map" style="width: 800px; height: 500px"></div>
<script type="text/javascript">
    var map = new GMap2(document.getElementById("map"));
</script>
```

If you'd like more than one map on your page, simply give the divs unique ids. (You can also add a GOverviewMapControl to achieve the same effect as our example here. We'll look at custom controls in the next chapter.)

```
<div id="overviewMap" style="width: 200px; height: 125px"></div>
<div id="detailMap" style="width: 800px; height: 500px"></div>
<script type="text/javascript">
    var overviewMap = new GMap2(document.getElementById("overviewMap"));
    var detailMap = new GMap2(document.getElementById("detailMap"));
</script>
```

4.2 GLatLng

The maps we've defined up to this point are missing two critical pieces: the center point and the zoom level. Without these two additional pieces of information, the maps cannot be rendered.

GLatLong

A *GLatLong* object is a single Latitude/Longitude point. A common point of confusion (no pun intended) is the proper order of the ordinates. In mathematics, we're used to (x,y) ordering. That is a (longitude, latitude) point, geographically speaking. So GLatLong points are really (y,x) ordered. Later in the book, we talk about GPoints that refer to a specific pixel location on the screen. GPoints use conventional (x,y) ordering. Confused yet? Yeah, me too.

The other order of business we need to take care of is the zoom level of the map. The zoom level is an integer ranging from 0-18, where 18 is zoomed in to the street level. A zoom level of 1 displays the whole world if your map size is set to width: 550px; height: 525px.

Zoom level 0, interestingly enough, zooms you out to the place where you can see several world maps stitched together at the international date line. Why on Earth would you want to see several world maps at once? It boils down to the trickiness of displaying a round world on a flat map.

If you are looking at your home town on a globe, you can spin the globe and eventually get back to the same place. On a flat map, this isn't as easy. To allow the users to "spin" the flat map—continue panning in the same direction and eventually get back to the same place—Google stitched together several duplicate flat maps to approximate the effect. Zoom level 1 shows you the middle map. Zooming out past that shows you the duplicate map on either side.

(Neat trick, if you ask me...)

The map.setCenter() method should be used to initialize your GMap. After the map has been rendered for the first time, you can continue to use map.setCenter() with both arguments, or adjust the properties independently using map.setCenter() with a single GLatLng and map.setZoom() with an integer zoom value. To query the map for the current state of these properties, use map.getCenter() and map.getZoom().

```
<div id="overviewMap" style="width: 200px; height: 125px"></div>
<div id="detailMap" style="width: 800px; height: 500px"></div>
<script type="text/javascript">
       overviewMap = new GMap2(document.getElementById("overviewMap"));
       detailMap = new GMap2(document.getElementById("detailMap"));

       //NOTE: This is the geographic center of the US
       var usCenterPoint = new GLatLng(39.833333, -98.583333);

       overviewMap.setCenter(usCenterPoint, 1);
       detailMap.setCenter(usCenterPoint, 7);
</script>
```

4.3 GLatLngBounds

GLatLngBounds

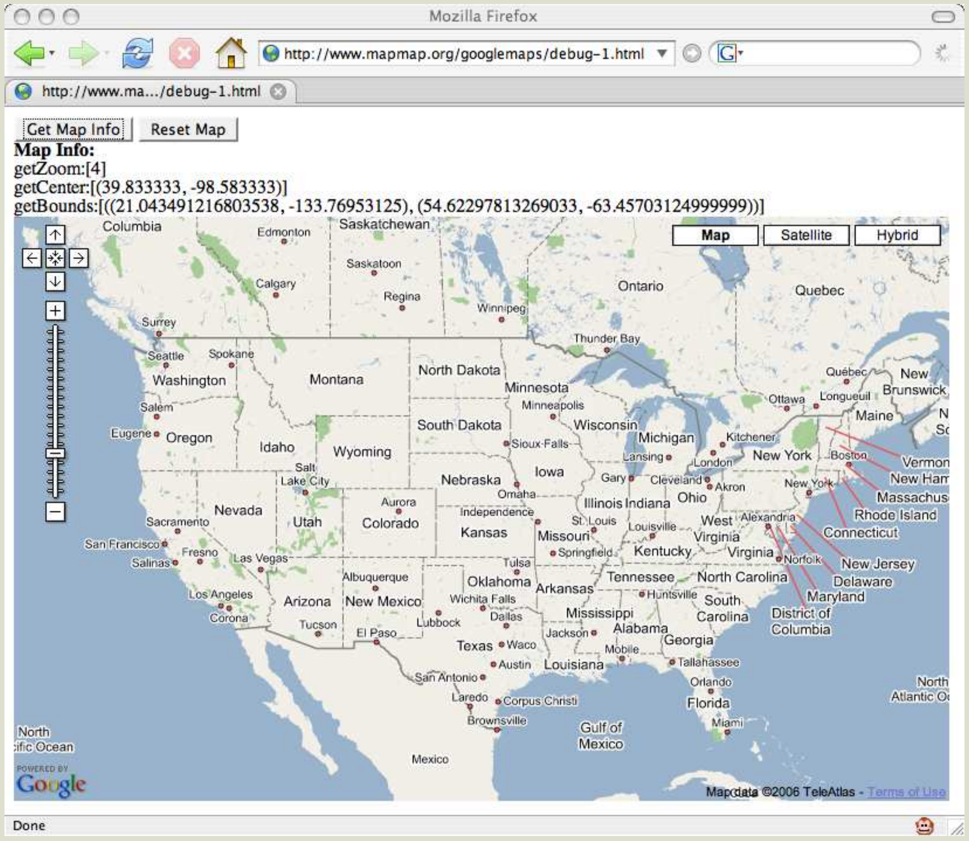
The *GLatLngBounds* object represents the geographic size of our map. ("Bounding Box" is a common cartographic term used to describe the size of a map.) It is a two element array of GLatLngs. The first element is the lower-left corner of the map; the second one is the upper-right corner.

The physical size of the map doesn't change—it is defined by the style attribute of the HTML div. But the geographic bounds of the map are constantly changing. Each time you pan, you are looking at a new bounding box. Even if your center point doesn't change, when you zoom in or out your bounding box changes. Recall that map.getCenter() returns a GLatLng. A complementary method, named map.getBounds(), returns a GLatLngBounds object.

This example brings everything together that we've discussed up to this point. (See Figure 4.1, on page 19 for the resulting map, or go to http://www.mapmap.org/googlemaps/debug-1.html to play with it live.) As you pan and zoom around, click the *Get Map Info* button to see how the values change. You can click the *Reset Map* button to return it to its initial state.

File 5

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="http://maps.google.com/maps?file=api&v=2&key=xyz"
            type="text/javascript"></script>

    <script type="text/javascript">
        var map = "";
        var usCenterPoint = new GLatLng(39.833333, -98.583333);
        var worldCenterPoint = new GLatLng(0,0);
        var initialCenterPoint = usCenterPoint;
        var initialZoomLevel = 4;
        var initialMapType = G_NORMAL_MAP;
    </script>

  </head>
  <body onLoad="initMap()">

    <input type="button" name="Refresh" value="Get Map Info"
           onClick="getMapInfo()" />
    <input type="button" name="Reset" value="Reset Map"
           onClick="resetMap()" />
    <div id="mapInfo">Map Info:</div>
    <div id="map" style="width: 800px; height: 500px"></div>
    <!-- If you set the initialZoomLevel to 1 and the initialCenterPoint to --
    <!-- worldCenterPoint, this map displays the entire world perfectly: -->
    <!-- <div id="map" style="width: 550px; height: 525px"></div> -->

    <script type="text/javascript">
        function initMap()
        {
           map = new GMap2(document.getElementById("map"));
           map.addControl(new GLargeMapControl());
           map.addControl(new GMapTypeControl());
           resetMap();
        }

        function getMapInfo()
        {
            var output = "<b>Map Info:</b><br />";
            output = output +
                     "getZoom:[" + map.getZoom() + "]<br />";
            output = output +
                     "getCenter:[" + map.getCenter() + "]<br />";
            output = output +
                     "getBounds:[" + map.getBounds() + "]<br />";

            mapInfo = document.getElementById("mapInfo");
            mapInfo.innerHTML = output;
        }

        function resetMap()
        {
           map.setCenter(initialCenterPoint, initialZoomLevel);
           map.setMapType(initialMapType);
        }
    </script>

  </body>
</html>
```



```
Map Info:
getZoom:[4]
getCenter:[(39.833333, -98.583333)]
getBounds:[((21.043491216803538, -133.76953125), (54.62297813269033, -63.45703124999999))]
```

Figure 4.1: Working with Core Objects

Chapter 5

Map Control Objects

Now that we have our basic map (or maps) in place, let's provide the user a way to interact with it.

5.1 Panning

Every GMap2 that you place on the page has dragging enabled by default. You can use map.enableDragging() and map.disableDragging() to control this programmatically. You can also query for the current dragging state using map.draggingEnabled(). This will return true if the map is draggable.

Why would you want to disable arguably the coolest feature of a Google Map? In the Overview/Detail map example, you might want to freeze the overview map in place while allowing the detail map to be moved around freely. Or maybe you are just a control freak. Either way, the choice is yours to decide if your map is draggable or not.

5.2 Zooming

In the quest for simplicity, the developers of Google Maps decided not to make the mouse multi-modal. In other words, dragging with the left mouse button will always pan the map. To give your users the ability to zoom the map, you must add a zoom component to the map.

GLargeMapControl

The API provides three choices for zoom controls. The traditional one used on maps.google.com is a *GLargeMapControl* object. This object shows the full 18 levels of zoom on a slider with plus and minus buttons on the top and bottom. (There is a set of pan buttons above the zoom slider.) The slider, in addition to being clickable, provides a nice, simple visual cue as to what your current zoom level is.

GSmallMapControl

If you prefer a more compact widget, the *GSmallMapControl* object offers the plus/minus zoom buttons without the slider. It also provides a set of pan buttons.

GSmallZoomControl

The smallest possible zoom widget is the *GSmallZoomControl*. This object only displays the plus/minus buttons—no zoom slider, no pan controls.

For the really ambitious, you can subclass GControl and create your own custom map control. For more information and a nice code example, see the Google Documentation.

To add a zoom control widget to your map, use map.addControl(new GLargeMapControl()), substituting GSmallMapControl() or GSmallZoomControl() as appropriate. You can add all three at once, but they overlap each other in both functionality and screen real estate. They appear along upper left side of your map. Bear in mind that the pan controls work even if you have disabled dragging on your map.

GScaleControl

The GLargeMapControl widget gives users a visual cue as to how far they are zoomed in. Traditionally, print maps offer the user a similar indicator in the form of a scale that shows inches or centimeters in relation to miles or kilometers. The *GScaleControl* provides this functionality. While it isn't an interactive widget like the others we've discussed so far, it is added to the map using the same map.addControl().

GControlPosition

map.addControl() accepts an optional *GControlPosition*, allowing you to place the control anywhere on the map you'd like. GControlPosition takes two arguments: an anchor and an offset. The anchor can be one of four constant values:

- G_ANCHOR_BOTTOM_LEFT
- G_ANCHOR_BOTTOM_RIGHT
- G_ANCHOR_TOP_LEFT
- G_ANCHOR_TOP_RIGHT

GSize

The offset is a *GSize*, which is an (x,y) pair that creates an invisible rectangle. In the case of a GSize, x equals the width of the box; y equals the height.

```
map.addControl(new GLargeMapControl());
map.addControl(new GScaleControl(),
               new GControlPosition(G_ANCHOR_BOTTOM_RIGHT,
                                    new GSize(20,20)));
```

5.3 Changing the Map Type

Another way that users can interact with the map is by changing the map type. The default map type, called Map mode, is a simple line drawing. There are two more map types available: Satellite and Hybrid. Satellite mode shows imagery over the area instead of line drawings. Hybrid mode superimposes the Map mode over the Satellite mode, giving you the best of both worlds.

GMapTypeControl

You can adjust the map type using map.setMapType(map_type), where map_type can be G_NORMAL_MAP, G_SATELLITE_MAP, or G_HYBRID_MAP. Or you can display the *GMapTypeControl* widget in the upper right corner using

```
map.addControl(new GMapTypeControl())
```

GMapType

You can use map.getCurrentMapType() to determine the current map. It returns a *GMapType* object. To display the name of the current map type, use mapType.getName().

```
alert( map.getCurrentMapType().getName() );
```

5.4 GOverviewMap

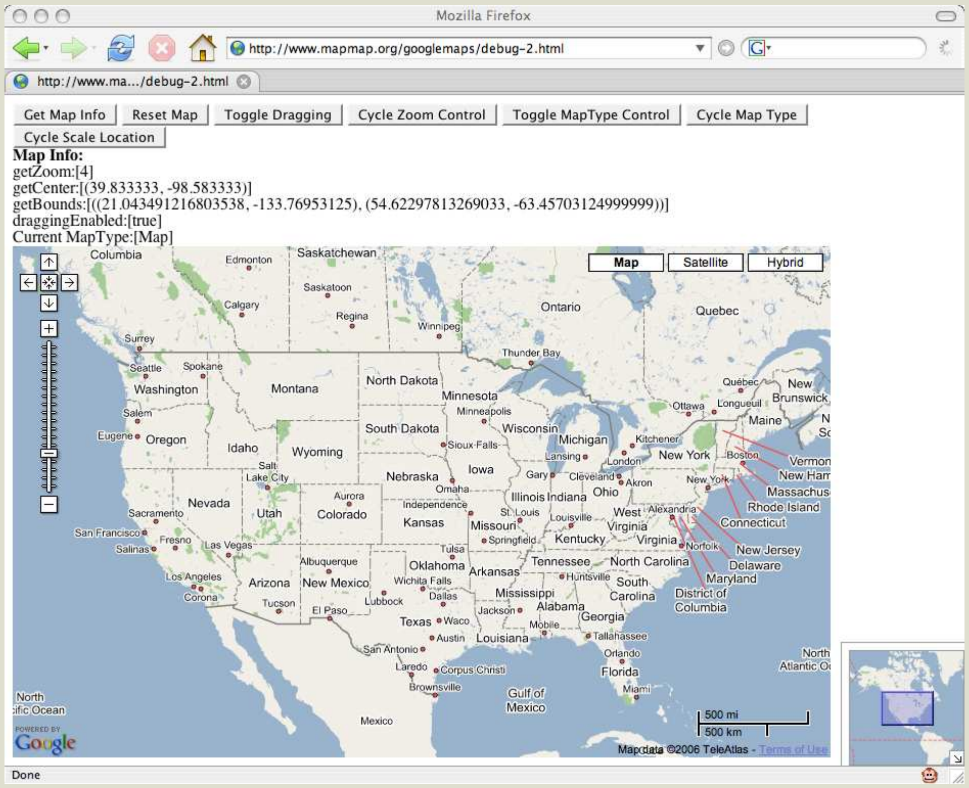
GOverviewMap

The API offers one last type of built-in control: a *GOverviewMap*. As the name suggests, this provides a tiny overview map of your main map. It is fully interactive – if you drag the OverviewMap, it pans the main map as well. To add one to the screen, use map.addControl(new GOverviewMap());.

5.5 Putting it all together

http://www.mapmap.org/googlemaps/debug-2.html

Let's put everything together in another map. (See Figure 5.1, on the next page, for the resulting map, or play with it online.) The getMapInfo() function has been updated to include the dragging state and current map type. There are additional buttons along the top that allow you to exercise the various methods we discussed in this section: Toggle Dragging, Cycle Zoom Control, Toggle MapType Control, Cycle Map Type, and Cycle Scale Location. Choose View/Source in your browser to see the source code.



```
Map Info:
getZoom:[4]
getCenter:[(39.833333, -98.583333)]
getBounds:[((21.043491216803538, -133.76953125), (54.62297813269033, -63.45703124999999))]
draggingEnabled:[true]
Current MapType:[Map]
```

Figure 5.1: Manipulating the Map

Chapter 6

User Data Objects

Now that you feel comfortable working with a “stock” map, let’s look at adding your own custom data. The API allows you to add points and lines (GMarkers and GPolylines, respectively). You can also customize the appearance of your GMarker by creating a GIcon.

6.1 GMarker

GMarker

In the Core Objects section, we introduced the GLatLng. A GLatLng stores a Latitude / Longitude coordinate, but it doesn’t offer you a way to visualize it on a map. A *GMarker* is the way to add GLatLngs to the map for display purposes. The GMarker constructor takes a GLatLng as the only required argument.

Overlays

Once we have the marker, we need to tell the map to display it; map.addOverlay(myMarker) should do the trick. (Objects that you superimpose over the map are called *Overlays*.) You can remove the marker using map.removeOverlay(myMarker). To remove all overlays, use map.clearOverlays().

```
var myPoint = new GLatLng(38.898748, -77.037684);
var myMarker = new GMarker(myPoint);
map.addOverlay(myMarker);
```

Theoretically a map can support an unlimited number of markers, but anecdotal evidence suggests that performance starts to slow down significantly after a hundred or so markers. (File under, “Doc, it hurts when I do this.”)

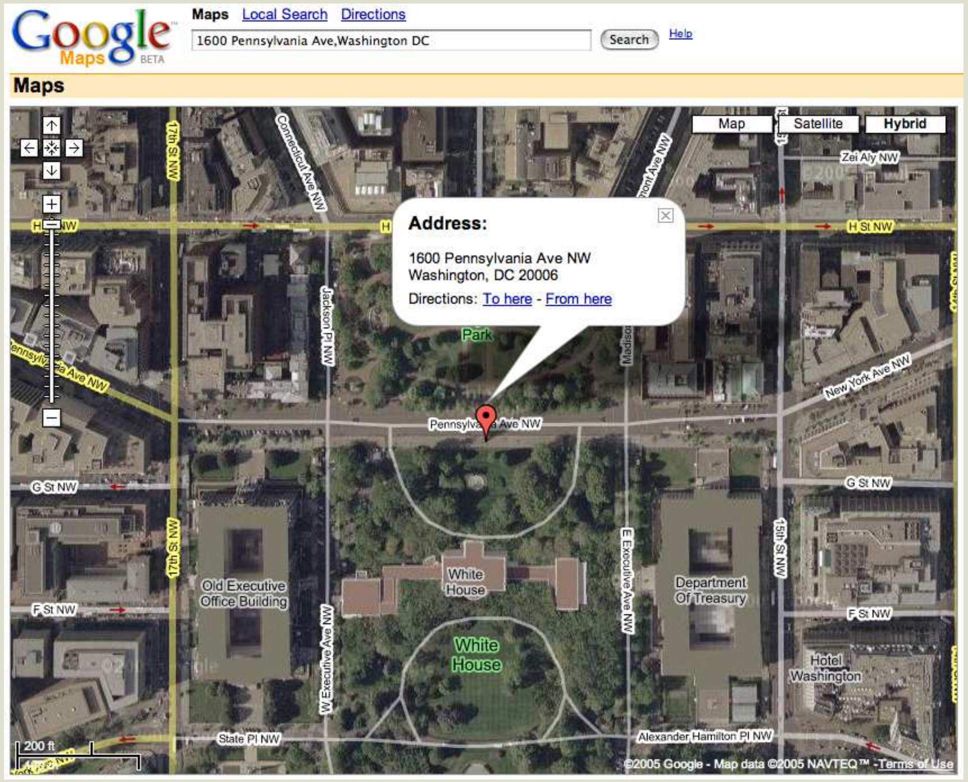

Google Maps BETA
Maps Local Search Directions
1600 Pennsylvania Ave,Washington DC
Search
Help
Maps
Map Satellite Hybrid
Address:
1600 Pennsylvania Ave NW
Washington, DC 20006
Directions: To here - From here
White House
Old Executive Office Building
Department Of Treasury
Hotel Washington
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

Figure 6.1: The default GIcon with an Info Window

6.2 GIcon

The default icon used for a GMarker gives your map a distinct Google Maps look and feel, as Figure 6.1 shows. You can, however, substitute your own GIcon for a more personalized touch. But before we get into the API details of how to specify a custom GIcon, we should talk about how to get a suitable image to use.

Your icon must be a PNG file. Any size will do, but anything more than 20-30 pixels square will begin to look too big in relation to the

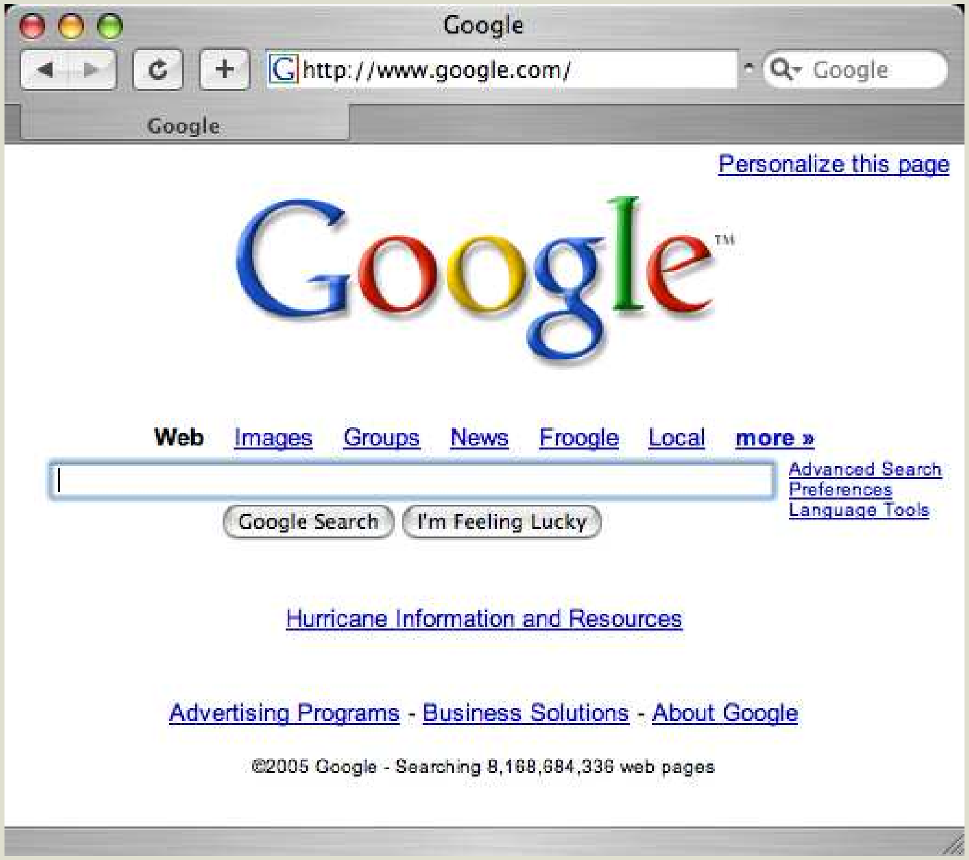

Google
http://www.google.com/
Google
Google
Personalize this page
Google
Web Images Groups News Froogle Local more »
Advanced Search
Preferences
Language Tools
Google Search
I'm Feeling Lucky
Hurricane Information and Resources
Advertising Programs - Business Solutions - About Google
©2005 Google - Searching 8,168,684,336 web pages

Figure 6.2: Notice the Google favicon in the URL bar.

http://www.google.com/mapfiles/marker.png

rest of the map. The default marker is 20x34 pixels.

favicons

I don't know about you, but I became a programmer due to a marked *lack* of artistic skills. The idea of hand-drawing my icons makes me feel positively nauseous. Luckily, there are a wealth of images out there that are just the right size. I'm speaking of *favicons*, those little custom icons that show up in the URL bar and bookmark menu as you go from website to website. (See Figure 6.2 .)

Nearly every commercial website out there has a custom favicon. You can usually download them directly from the root of the website. For example, Google's blue "G" can be found at in their site's top-level directory at http://www.google.com/favicon.ico. If you can't find a site's favicon in the root, do a View/Source and look for a tag in the header such as <link rel="shortcut icon" href="/favicon.ico"/>.

For more information on favicons, see Wikipedia.

I'm not suggesting that you should download website favicons and use them without asking permission—they are copyrighted material, after all—but we can use them as an inspiration for what we are trying to accomplish. Favicons are the perfect size for GIcons: by definition they are 16x16. And since they are so prevalent, there are a number of free utilities and websites that make it easy to create them.

The one that I use most often is http://www.html-kit.com/favicon. It allows you to upload an image of any size, in nearly any file format, and download the resulting 16 x 16 favicon in a zip file. The favicon is nice, but we need PNGs for the Google Map API. Luckily, in addition to the favicon.ico file, you get a 16 x 16 PNG preview image in the extra folder of the zip. Voila! Google Maps-ready icons in a can.

http://www.imagemagick.org

In my quest for quick and easy icons, I've also gotten quite handy with a command-line graphics program called ImageMagick. No, "command-line graphics program" is not an oxymoron. It allows me to do image manipulation from where I feel most comfortable – at the blinking cursor of a black and white terminal window. ImageMagick is open source (so the price is right) and available on all of the major platforms (Mac, Linux, and Windows, among others).

You can use the identify command to get the exact pixel size of any image (identify foo.gif). You can then use the convert command to change formats and optionally scale the image. If you already have a favicon (and permission to use it), convert favicon.ico mylogo.png will get it into the proper format. To resize images as you change formats, use the scale parameter (convert -scale 16x16 bigimage.tiff tinyimage.png).

Now that we have a suitable image, let's incorporate it into our map. First, define a GIcon. Then, the constructor of a GMarker takes an optional GIcon argument after the required GLatLng.

```
var myIcon = new GIcon();
myIcon.image = "http://www.mapmap.org/googlemaps/google.png";
myIcon.iconSize = new GSize(16,16);

var myMarker = new GMarker(myPoint, myIcon);
map.addOverlay(myMarker);
```

The GIcon can be further customized in a number of ways. You can use icon.shadow and icon.shadowSize to add, umm, *shadows* to your icon. The Google Maps graphic designers have an incredibly keen eye for detail when it comes to shadows. Go back and take a look at a Google Map with overlays on it (for example, Figure 6.1, on page 26). Can you see the subtle drop shadow to the right of both the marker and the Info Window?

I used the following ImageMagick command to create the shadow image.

```
convert -shear 45 -charcoal 1 \
        -resize 32x16! -background none \
        google.png google_shadow.png
```

I am generally satisfied with matte custom icons in my maps. (Or should I say lucky to have them at all?) But for the sake of completeness, here are two examples of GIcons with shadows. The first one is the default values for a GIcon. The second is a custom icon, shadows and all. For a really classy shadow, I could play with the Alpha channel to adjust the transparency of the image as well. (I'll leave that as an exercise for the artists out there who are, as we speak, shaking their heads in shame at my pathetic attempt at artistic integrity.)

```
var defaultIcon = new GIcon();
defaultIcon.image = "http://www.google.com/mapfiles/marker.png";
defaultIcon.iconSize = new GSize(20,34);
defaultIcon.shadow = "http://www.google.com/mapfiles/shadow50.png";
defaultIcon.shadowSize = new GSize(37,34);

var myIcon = new GIcon();
myIcon.image = "http://www.mapmap.org/googlemaps/google.png";
myIcon.iconSize = new GSize(16,16);
myIcon.shadow = "http://www.mapmap.org/googlemaps/google_shadow.png";
myIcon.shadowSize = new GSize(32,16);
```

There is one last thing we need to discuss as far as custom icons are concerned – anchors. When it comes to "X marks the spot", we need to give the map a hint as to which part of the GIcon should sit over the GLatLng. The icon.iconAnchor property represents the exact pixel of the icon that should match up with the point on the map. The upper left corner of the icon is (0,0). The default GIcon looks like it hits map right about bottom-center, or (10,34).

If we are going to be attaching Info Windows to our GIcons, we also need to specify the icon.infoWindowAnchor property. The default GIcon attaches the Info Window at top-center, or (10,0). (Again, see Figure 6.1, on page 26.)

```
var defaultIcon = new GIcon();
defaultIcon.image = "http://www.google.com/mapfiles/marker.png";
defaultIcon.iconSize = new GSize(20,34);
defaultIcon.shadow = "http://www.google.com/mapfiles/shadow50.png";
defaultIcon.shadowSize = new GSize(37,34);
defaultIcon.iconAnchor = new GPoint(10,34);
defaultIcon.infoWindowAnchor = new GPoint(10,0);
```

6.3 Info Windows

Now that our GIcon knows where to anchor an Info Window, let's pop one up.

The API treats Info Windows slightly differently than GMarkers in several ways. While you can have as many GMarkers on the map as you like, the API limits you to displaying a single Info Window at a time. If you add a GMarker offscreen, the map doesn't scroll to display it. If you open an Info Window offscreen, the map pans smoothly until the Info Window comes into view.

As with dragging earlier, you can control the ability of the map to display Info Windows using the methods map.enableInfoWindow() and map.disableInfoWindow(). To check the current state, use the method map.infoWindowEnabled().

Info Windows are not first-class objects that you can instantiate and leave laying around. Instead, you must call one of several openInfoWindow() methods on either a map or a GMarker.

An Info Window requires a point and a payload. If openInfoWindow() is called on a GMarker, the point is implied. If it is called on a map, you must specify it. The payload can be either an HTML DOM element or a string containing HTML (map.openInfoWindow() or map.openInfoWindowHtml(), respectively).

The easiest way to get started is to just pop up an Info Window at an arbitrary point on the map. Since the map provides a convenient getCenter() method, we can use that to begin. openInfoWindowHtml() has two required arguments: the point and the HTML string.

```
var center = map.getCenter();
var coords = "Center point: (" + center.lat() + ", " + center.lng() + ")";
```

```
map.openInfoWindowHtml(center, coords);
```

If you're thinking to yourself that the coords variable doesn't look much like HTML, well, you're right. Despite the name, you can pass openInfoWindowHtml() a plain old string and it will get rendered faithfully. We'll play with embedding HTML tags in our string in just a minute.

Theoretically, map.openInfoWindow() looks more generic. In reality, it probably should've been named map.openInfoWindowDom(). For us to accomplish the same thing as above, we have to create a DOM textNode object to hold the string:

```
map.openInfoWindow(somePoint,
                   document.createTextNode("This is my string"));
```

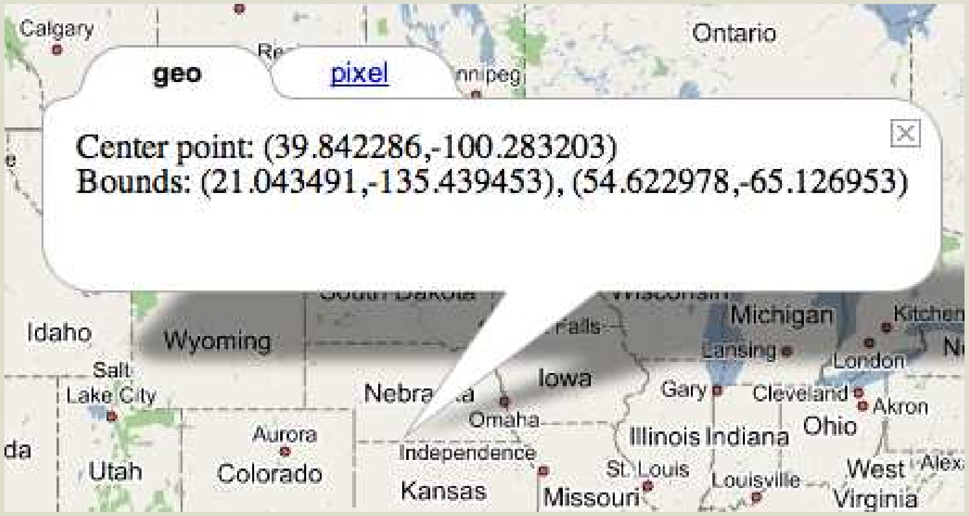
Opening an Info Window directly on a GMarker is pretty straightforward.

```
var myMarker = new GMarker(myPoint, myIcon);
map.addOverlay(myMarker);
myMarker.openInfoWindowHtml("Hey, check this out!");
```

GInfoWindowTab

Starting with v2 of the API, you can pop up tabbed Info Windows using openInfoWindowTabs() and openInfoWindowTabsHtml(). To do so, simply create an array of *GInfoWindowTab*s. (See Figure 6.3, on the following page.)

```
var geoCenter = map.getCenter(); //returns a GLatLng
var geoBounds = map.getBounds(); //returns a GLatLngBounds
var geoTabText = "Center point: (" + geoCenter.toUrlValue() + ")";
geoTabText += "<br />";
geoTabText += "Bounds: (" + geoBounds.getSouthWest().toUrlValue() + ")";
geoTabText += ", (" + geoBounds.getNorthEast().toUrlValue() + ")";

var pixelCenter = map.fromLatLngToDivPixel(geoCenter); // returns a GPoint
```



Center point: (39.842286,-100.283203)
Bounds: (21.043491,-135.439453), (54.622978,-65.126953)

Figure 6.3: New to v2: tabbed Info Windows.

```
var pixelBounds = map.getSize(); //returns a GSize
var pixelTabText = "Center point: " + pixelCenter.toString();
pixelTabText += "<br />";
pixelTabText += "Bounds: " + pixelBounds.toString();

var tabs = [
  new GInfoWindowTab("geo", geoTabText),
  new GInfoWindowTab("pixel", pixelTabText)
];

map.openInfoWindowTabsHtml(geoCenter, tabs);
```

GInfoWindowOptions

By default, the first tab in the array is displayed. To override this, pass in a *GInfoWindowOptions* object to openInfoWindowTabsHtml() as the last argument. The selectedTab property is the index of the tab in array notation (the first element is 0, the second is 1, etc.).

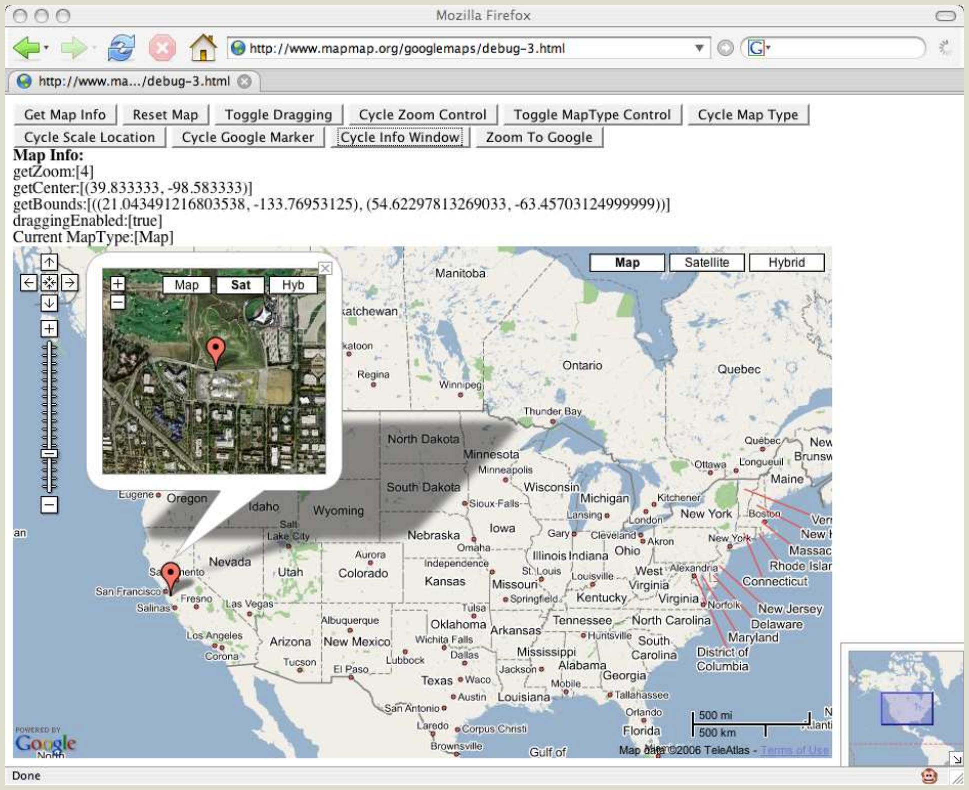
```
var tabs = [
  new GInfoWindowTab("geo", geoTabText),
  new GInfoWindowTab("pixel", pixelTabText)
];

var opts = new Object();
opts.selectedTab = 1;

map.openInfoWindowTabsHtml(geoCenter, tabs, opts);
```

In addition to text-based Info Windows, you can display a graphical Info Window that shows a zoomed-in mini map over the point. (Think of it as a digital magnifying glass.) showMapBlowup() is available on both maps and markers. By default, it uses a zoomLevel of 17 and the current mapType. You can override these values by passing in a GInfoWindowOptions.

There is no GInfoWindowOptions object, really. You cannot instantiate it. (Notice in the code example that we create a plain old new Object().) It is given a proper name for documentation purposes only. Technically, you don't even have to bother with creating a named object; just pass in properties on an anonymous object in the function call.

```
map.addOverlay(googleMarker);

var opts = new Object();
opts.zoomLevel = 14;
opts.mapType = G_SATELLITE_MAP;

googleMarker.showMapBlowup(opts);

//NOTE: rather than creating a separate opts object,
//      you can pass in an anonymous object
googleMarker.showMapBlowup({zoomLevel:14, mapType:G_SATELLITE_MAP});
```

Take a look at http://www.mapmap.org/googlemaps/debug-3.html to see this in action. The *Cycle Google Marker* button shows you a default and custom GIcon over Google's headquarters. The *Cycle Info Windows* button shows you an Info Windows on the map, an Info Window on a GMarker, and a Map Blowup. (See Figure 6.4, on the next page.)



```
Map Info:
getZoom:[4]
getCenter:[(39.833333, -98.583333)]
getBounds:[((21.043491216803538, -133.76953125), (54.62297813269033, -63.45703124999999))]
draggingEnabled:[true]
Current MapType:[Map]
```

Figure 6.4: A GIcon showing a Map Blowup Info Window.

6.4 GPolyline

GPolyline

To wrap up this chapter, lets talk about *GPolyline*s. These are the lines used by maps.google.com to display your turn-by-turn driving directions. Even though it may sound a bit strange to the untrained ear initially, "polyline" is technically the correct term. A *line* is a straight line with two end points. A *polyline* has multiple vertices, allowing our map element to zig-zag as expected.

To build a GPolyline, simply pass it an array of GLatLngs in the http://www.hypersolutions.org/pages/rgbhex.html

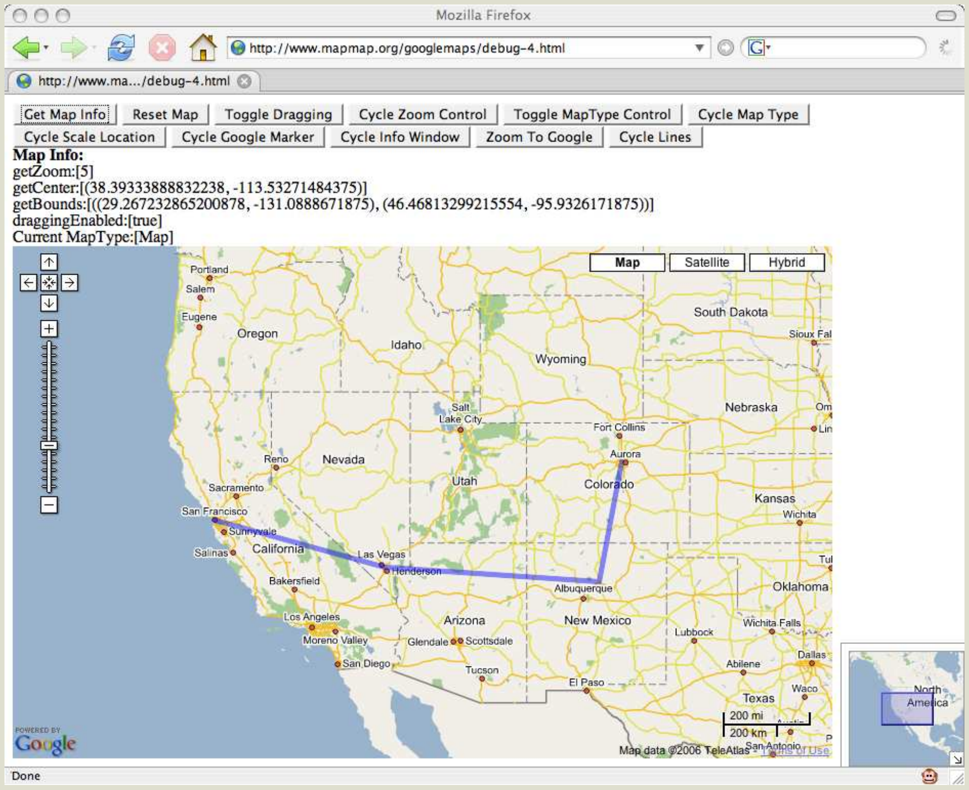
constructor. If you would like to tweak the appearance, you can pass in three optional arguments: color, weight, and opacity. Color is the same RGB hex string you use in traditional web development. (There are many nice webpages that list hex RGB values, such as this one.) Weight is an integer value; the higher the number, the thicker the line. Opacity is a decimal value that specifies the transparency of the line. 1 is fully opaque. 0 is fully transparent. For 50% opacity, use 0.5.

```
var myPoints = new Array(denver, santaFe, lasVegas, sanFrancisco);
var myLine = new GPolyline(myPoints);
map.addOverlay(myLine);

var myFancyPoints = new Array(denver, saltLakeCity, portland, seattle);
var myFancyLine = new GPolyline(myFancyPoints, "#FF0000", 10, 0.5);
map.addOverlay(myFancyLine);
```

Go to http://www.mapmap.org/googlemaps/debug-4.html to see GPolylines in action. The "Cycle Lines" button allows you to see a simple polyline and a styled example. (See Figure 6.5, on the following page.)

In most browsers, polylines are rendered on the Google servers and passed down to the browser as an overlay. If you are using Microsoft Internet Explorer, you can have the browser render polylines client-side. Doing so will improve your performance. Include the following snippet of code to have IE use native VML rendering:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:v="urn:schemas-microsoft-com:vml">
  <head>
    <style type="text/css"> v\:* { behavior:url(#default#VML); } </style>
   </head>
```



```
Map Info:
getZoom:[5]
getCenter:[(38.39333888832238, -113.53271484375)]
getBounds:[((29.267232865200878, -131.0888671875), (46.46813299215554, -95.9326171875))]
draggingEnabled:[true]
Current MapType:[Map]
```

Figure 6.5: GPolylines on a map.

Even if you get past all of the geo-geek jargon we've thrown at you up to this point, the GPolyline might still end up breaking your heart. People who want to display a polyline that follows the underlying roads like, "Plot my driving route from Denver to Las Vegas using Interstate 70" will be disappointed. This level of functionality is simply not provided by the current API. You are given the mechanics to draw lines on your map, but not the underlying intelligence agent.

Maps.google.com allows you to perform simple "Point A to Point B" queries and display a route. While this is quite useful, it would be great if the API allowed us to leverage this base functionality to do more complicated actions. The website MapsOnUs lets you create a driving plan that goes from Point A to Point B with intermediate stopping points along the way. Even if the Google Maps API didn't provide this level of functionality initially, making routes first class objects like GMarkers would allow us to instantiate several of them and display them simultaneously on the map. (Perhaps in the next release of the API, eh?)

http://www.mapsonus.com/

Starting with v2 of the API, Google has given us the capabilities to begin this process. Just like the GControl object that allows you to create your own custom map control, v2 introduces the new *GOverlay* object. Creating your own overlay (such as a routing overlay) is entirely possible now, but it's not for the faint of heart. For more information and some source code, see the Google Documentation.

GOverlay

Chapter 7

Events

At this point, we now have all of the building blocks in place. You can create a map. You can put up controls that allow the user to change the basic characteristics of the map. You can add custom objects like points, lines, and Info Windows. But these last two sections – Events and AJAX – are what really (excuse the pun) put Google on the map of mapping websites.

When Google Maps was first released, it wasn't the first mapping site on the web. MapQuest, Yahoo Maps, and MSN Maps had all been around for years. The model was well understood – type in your address, click submit, and then print out your static map. Panning was a "click, then wait for the screen to refresh" affair.

What Google Maps brings to the table was an amazing level of interactivity. You are inexorably drawn to mess around with a Google map – drag it around, flip on satellite imagery, zoom in and out. With the stock JavaScript event model available to you, coupled with the custom events on the map and map objects, you can continue that level of deep interactivity in a customized way.

7.1 GEvent

The event model allows you to react to the user – do something customized when they click on the map, add points to the map from an HTML form, etc. The map object is just one of many objects on your page. Using standard JavaScript coupled with the Google Maps API, you can bridge your custom HTML and your custom map by having a table click place something on the map, or a map click insert something into a table.

If you've done JavaScript programming in the past, you may be familiar with the "old school" way of event handling:

```
<select onchange='doSomethingElse()'>
<button onclick='doSomething()'>
```

The on*Whatever* attributes assign a single event listener to specific element events. The Google Maps API uses the more modern method of programmatically adding event listeners to elements. Moving from on*Whatever* attributes to event listeners, in addition to making your code more loosely coupled, allows you to associate more than one listener to the same event.

GEvent

You can tap into the Google event model using static methods on the *GEvent* object:

```
var clickListener = GEvent.addListener(map, "click", doSomething());
GEvent.removeListener(clickListener);
```

The GEvent object does something very nice for us—it normalizes the JavaScript event model across browsers. Browsers like Firefox and Opera use the standard DOM event model. Internet Explorer uses a proprietary, incompatible model. It is common practice for web developers to bridge the gap between disparate browsers by writing their own normalization methods:

```
function registerListener(domElement, event, listener, captureEvent)
{
    //Normalize event registration across all browsers
    //1. Mozilla/Opera, 2. IE, 3. Other

    if(domElement.addEventListener){
      domElement.addEventListener(event, listener, captureEvent);
```

```
            return true;
        }
        else if(domElement.attachEvent){
            return domElement.attachEvent('on'+ event, listener);
        }
        else{
            domElement['on'+ event] = listener;
            return true;
        }
}
```

Notice that this code doesn't check on the browser—it checks on the availability of the function call. The code snippet

```
if (domElement.addEventListener)
```

says, "If you (the current browser I'm running in right now) understand this call, then go ahead and do something interesting." Similarly, detaching an event listener can be accomplished in a cross browser fashion using the following code:

```
function unregisterListener(domElement, event, listener, captureEvent)
{
        //Normalize event registration across all browsers
        //1. Mozilla/Opera, 2. IE, 3. Other

        if(domElement.removeEventListener){
            domElement.removeEventListener(event, listener, captureEvent);
            return true;
        }
        else if(domElement.detachEvent){
            return domElement.detachEvent('on'+ event, listener);
        }
        else{
            domElement['on'+ event] = "";
            return true;
        }
```

I should point out that there are plenty of open source JavaScript frameworks out there that take care of these mundane details for you out of the box. Prototype is one popular example. http://www.ajaxian.com is a great resource for keeping up with this fast-moving market segment. I purposely chose to limit our focus to the Google Maps JavaScript framework in this book – learning one framework at a time is enough, wouldn't you say?

```
}
```

The GEvent object encapsulates this type of behavior so that we, the developers, don't have to account for which browser we're running in by using additional code. Until all vendors decide to become conformant to the specification, it's nice that the API can compensate for us.

7.2 GBrowserIsCompatible

On a similar note, the Google Maps API offers us a method to check whether the browser is compatible:

```
if (GBrowserIsCompatible()) {
  var map = new GMap2(document.getElementById("map"));
  map.setCenter(myPoint, 1);
}
```

We haven't been wrapping our code in this call for readability reasons (and because all modern browsers are supported), but it's nice to know that this level of introspection is available to us by both native JavaScript and the Google Maps API.

7.3 GMap Events

Some of the events on the map object are simple events—they don't return any values, they just fire a notification.

drag()
: Sent continuously while the user is dragging the map.

dragstart()
: Sent when the user begins dragging the map.

dragend()
: Sent when the user stops dragging the map.

infowindowopen()
: Sent when an Info Window is diplayed.

infowindowclose()
: Sent when an Info Window is closed.

maptypechanged()
: Sent when the map is switched between Map, Satellite, and Hybrid modes.

Other map events return additional information. The `zoomend` event, for example, returns the previous and current zoom levels.

click(overlay, point)
: Sent when the map is clicked on. If the user clicks on a GMarker or other overlay object, that object is returned. If an open area of the map is clicked, the lat/long point of the click is returned.

zoomend(previousZoomLevel, currentZoomLevel)
: Sent when the user changes the zoom level of the map.

addoverlay(overlay)
: Sent when a new overlay is added to the map. The added overlay is returned.

removeoverlay(overlay)
: Sent when an overlay is removed from the map. The removed overlay is returned.

clearoverlays()
: Sent when all overlays are removed from the map.

mousemove(latlng)
: Sent continuously when the mouse is in motion. The lat/long of the cursor is returned.

mouseout(latlng)
: Sent when the mouse is moved off of the map. The lat/long of the cursor is returned.

mouseover(latlng)
: Sent when the mouse is moved on to the map. The lat/long of the cursor is returned.

7.4 Event Handlers

Events can either be passed to functions or handled by function closures. A function closure is similar to an anonymous inner class in Java. If the event handling code isn't more than a couple of lines and doesn't need to be reused, a closure is a quick and dirty way to code the behavior.

```
var dragListener = GEvent.addListener(map, "drag", function() {
    document.getElementById("output").innerHTML =
                                   map.getCenter().toUrlValue();
} );

var clickListener = GEvent.addListener(map,
                                      "click",
                                       function(overlay, point) {
    handleMapClick(overlay, point);
} );
```

7.5 GMarker Events

GMarkers also have events that you can listen for. Earlier, you saw that you could pop up an Info Window by either adding it to a map or to a GMarker. If you add it to the map, you have to provide the coordinates; if you add it to a marker, the point is implicit. The same idea applies to events: if you register a click listener on the map, the listener passes back the overlay or the point in the event. If you register a click listener on a point, the event doesn't have any associated arguments because you know explicitly which overlay was clicked.

click()
: Sent when the marker is single-clicked.

dblclick()
: Sent when the marker is double-clicked.

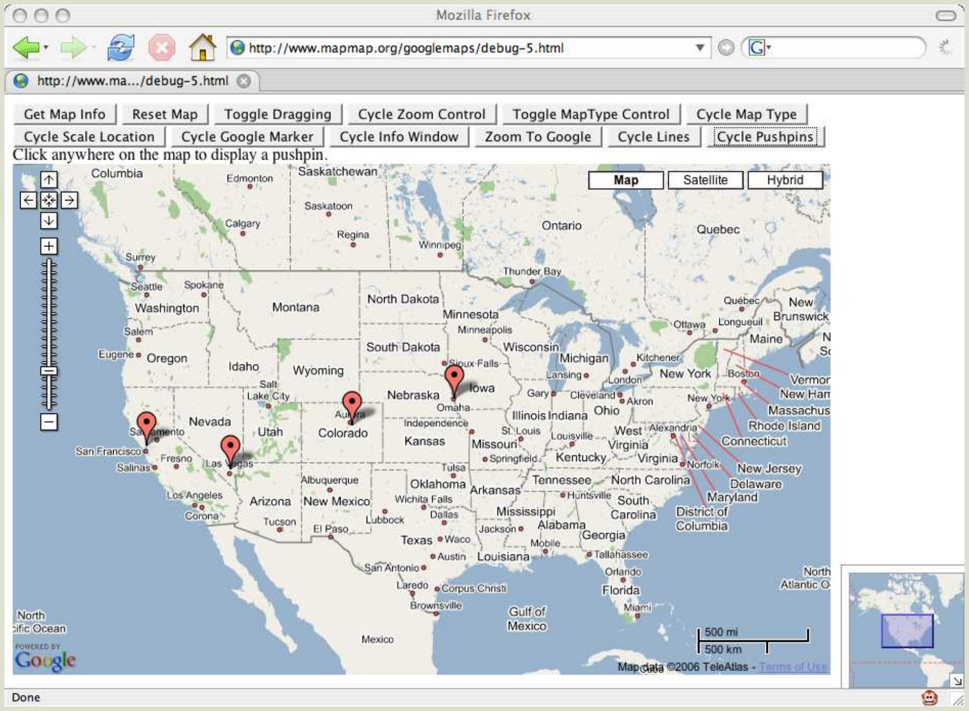
mouseover()
: Sent when the mouse moves over the marker.

mouseout()
: Sent when the mouse moves off of the marker.

infowindowopen()
: Sent when the corresponding Info Window is diplayed.

infowindowclose()
: Sent when the corresponding Info Window is closed.

remove()
: Sent when the marker is removed from the map.



Click anywhere on the map to display a pushpin.

Figure 7.1: Adding pushpins to your Google Map.

7.6 Simple Examples

See http://www.mapmap.org/googlemaps/debug-5.html for an example of map events in action (it looks a bit like Figure 7.1).

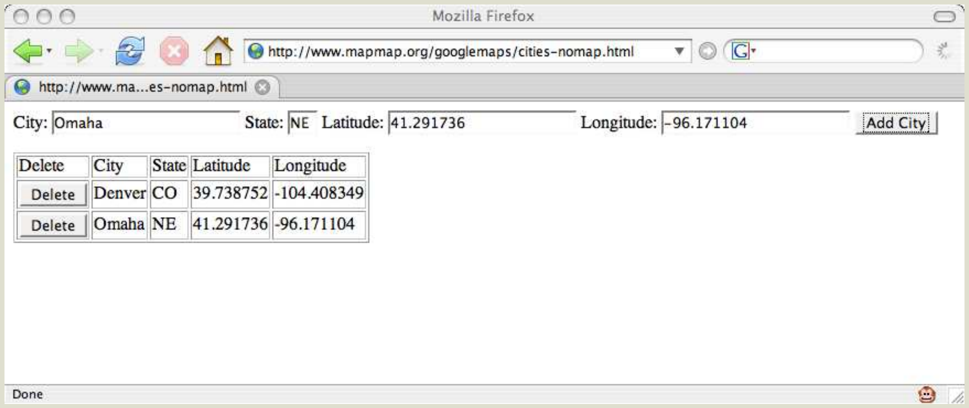
Notice that we added a "Cycle Pushpins" button. If you press the button once, it allows you to add a new GMarker by clicking on the map. Pressing the button a second time adds a second listener that displays the coordinates of the click at the top of the screen. Pressing the button a third time turns off both listeners and clears all of the markers.

```
//events
var pushpinsCycle = 0;
var clickListener;
var clickListener2;

function cyclePushpins()
{
    switch(pushpinsCycle)
    {
        //turn on event listener
        case 0:
            pushpinsCycle++;
            clickListener = GEvent.addListener(map, "click",
                                   function(overlay, point) {
                                      pushpin(overlay, point);
                                   } );
            document.getElementById("mapInfo").innerHTML =
              "Click anywhere on the map to display a pushpin.";
            break;

        //turn on another event listener
        case 1:
            pushpinsCycle++;
            clickListener2 = GEvent.addListener(map, "click",
                                     function(overlay, point) {
                                        showCoords(overlay, point);
                                     } );
            document.getElementById("mapInfo").innerHTML =
              "Click anywhere on the map to display a pushpin "
              + "<b><i>with coordinates</i></b>.";
            break;

        //turn off event listener
        case 2:
            pushpinsCycle = 0;
            GEvent.removeListener(clickListener);
            GEvent.removeListener(clickListener2);
            map.clearOverlays();
            document.getElementById("mapInfo").innerHTML =
              "All click listeners have been removed.";
            break;
    }
}

function pushpin(overlay, point)
{
   if (overlay) { map.removeOverlay(overlay); }
   else if (point) { map.addOverlay(new GMarker(point)); }
}

function showCoords(overlay, point)
{
    var output = "";
    if(point){
      output += "(" + point.toUrlValue() + ")";
    }
    document.getElementById("mapInfo").innerHTML = output;
}
```

7.7 A Real-World Example

Let's look at a more robust application. The beginning of it is online at http://www.mapmap.org/googlemaps/cities-nomap.html. (See Figure 7.2, on the following page.) There are two basic elements on the page: a form for data entry and an HTML table to display the results. Type values in the form and press the Add City button – the data should appear in the table. If you press the delete button on the table row, the row should disappear.

Now before you get too upset about having to type in Lat/Long points by hand, just settle down. Once we add a little AJAX to our


Mozilla Firefox

http://www.mapmap.org/googlemaps/cities-nomap.html

City: Omaha State: NE Latitude: 41.291736 Longitude: -96.171104 Add City

Delete	City	State	Latitude	Longitude
Delete	Denver	CO	39.738752	-104.408349
Delete	Omaha	NE	41.291736	-96.171104

Done

Figure 7.2: Using Javascript to add form data to the table

application, you'll be able to type in a city and have a web service fill in the coordinates for you. For that matter, once we add a map to our application, you'll be able to click on the map and have it fill in the coordinates for you. For now, we just want to exercise the plain old HTML and JavaScript.

Let's walk through the application. The first thing we do is register the addCity() function as an event listener for the *Add City* button. The pageInit() function is essentially our JavaScript constructor for the page—we wait until the entire HTML page is loaded, and then run the pageInit() function.

Waiting for the entire page to load helps avoid strange lifecycle issues across browsers—some browsers try to run the event listener code before the HTML elements exist, causing the application to fail silently.

```
<script type="text/javascript">
  &ldots;

  function pageInit()
  {
    registerListener(document.getElementById("addCityButton"),
                      "click", addCity, false);
  }

  registerListener(window, "load", pageInit, false);
</script>
```

Next, let's examine the addCity() function. When the user clicks the *Add City* button, we pull the data out of the HTML form via getCityBean() and pass it to the addToTable() and addToArray() functions.

```
function addCity()
{
  var cityBean = getCityBean();

  addToTable(cityBean);
  addToArray(cityBean);
}
```

The getCityBean() method takes advantage of a little JavaScript magic. We create a new generic Object and then add attributes to it on the fly. Each document.getElementById() method grabs input data from the HTML form.

```
<div id="formSection">
  <form id="cityForm" >
    City: <input type="text" name="city" id="city"/>
    State: <input type="text" name="state" id="state" size="2" maxlength="2"/>
    Latitude: <input type="text" name="latitude" id="latitude">
    Longitude: <input type="text" name="longitude" id="longitude"/>
    <input type="button" name="AddCity" value="Add City" id="addCityButton"/>
  </form>
</div>
```

```
function getCityBean()
{
  var cityBean = new Object();
  cityBean.city = document.getElementById("city").value;
  cityBean.state = document.getElementById("state").value;
  cityBean.latitude = document.getElementById("latitude").value;
  cityBean.longitude = document.getElementById("longitude").value;
  cityId++;
  cityBean.id = "city" + cityId;
  return cityBean;
}
```

Once our data is gathered together in a convenient bucket, we pass it to addToTable() to update the table display.

Notice that the table has an empty <*tbody*> element. While rarely used when hand-editing tables in HTML, each table technically has an implicit <*thead*>, <*tbody*>, and <*tfoot*> section. If you plan to use JavaScript to programmatically add and remove data from your HTML tables, it is best to explicitly create them and give them a unique id. When adding a row directly to a table element, most browsers do the right thing and add it to the <*tbody*>. Safari, on the other hand, does exactly what you ask it to do and adds the row to the table—above the <*thead*> section.

```
<div id="tableSection">
  <table border="1">
    <tr id="cityHeader">
        <td>Delete</td>
        <td>City</td>
        <td>State</td>
        <td>Latitude</td>
        <td>Longitude</td>
    </tr>

    <!-- Note: if tbody is not used, Safari adds new rows above the header -->
    <tbody id="cityTable"></tbody>
  </table>
</div>

function addToTable(cityBean)
{
  var tbl = document.getElementById("cityTable");
  var newRow = tbl.insertRow(tbl.rows.length);
  newRow.id = cityBean.id;


  var deleteCell = newRow.insertCell(0);
  var deleteButton = document.createElement("input");
```

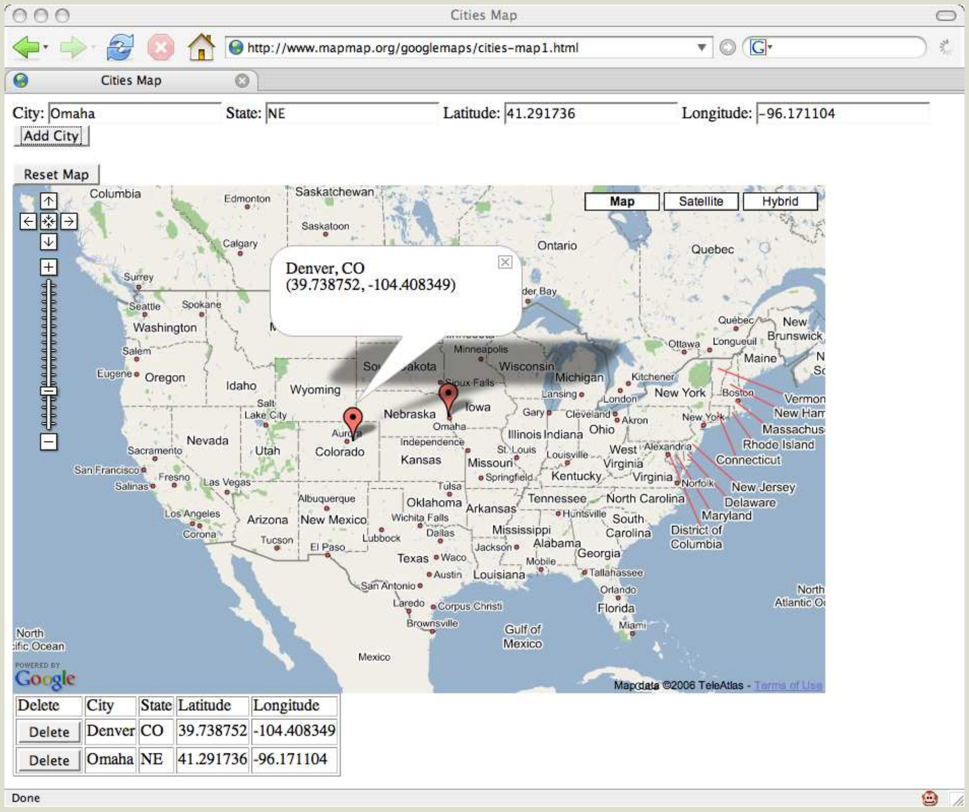
```
                              "click", addCity, false);
  }

  registerListener(window, "load", pageInit, false);
</script>
```

Next, let's examine the addCity() function. When the user clicks the *Add City* button, we pull the data out of the HTML form via getCityBean() and pass it to the addToTable() and addToArray() functions.

```
function addCity()
{
  var cityBean = getCityBean();

  addToTable(cityBean);
  addToArray(cityBean);
}
```

The getCityBean() method takes advantage of a little JavaScript magic. We create a new generic Object and then add attributes to it on the fly. Each document.getElementById() method grabs input data from the HTML form.

```
<div id="formSection">
  <form id="cityForm" >
    City: <input type="text" name="city" id="city"/>
    State: <input type="text" name="state" id="state" size="2" maxlength="2"/>
    Latitude: <input type="text" name="latitude" id="latitude">
    Longitude: <input type="text" name="longitude" id="longitude"/>
    <input type="button" name="AddCity" value="Add City" id="addCityButton"/>
  </form>
</div>

function getCityBean()
{
  var cityBean = new Object();
```


Cities Map
http://www.mapmap.org/googlemaps/cities-map1.html
City: Omaha State: NE Latitude: 41.291736 Longitude: -96.171104
Add City
Reset Map
Map Satellite Hybrid
Denver, CO
(39.738752, -104.408349)
Done

Delete	City	State	Latitude	Longitude
Delete	Denver	CO	39.738752	-104.408349
Delete	Omaha	NE	41.291736	-96.171104

Figure 7.3: Adding cities to the map

method—we get the cityId from the row, and call deleteFromTable(cityId) and deleteFromArray(cityId).

OK, now let's add in the map. (See Figure 7.3 , or http://www.mapmap.org/google
Most of the code should be familiar to you from previous examples. All we're doing is taking our lat/long point from the HTML form and constructing a GMarker out of it.

Our addToCity() function now has an addToMap(cityBean) function in addition to addToTable(cityBean) and addToArray(cityBean).

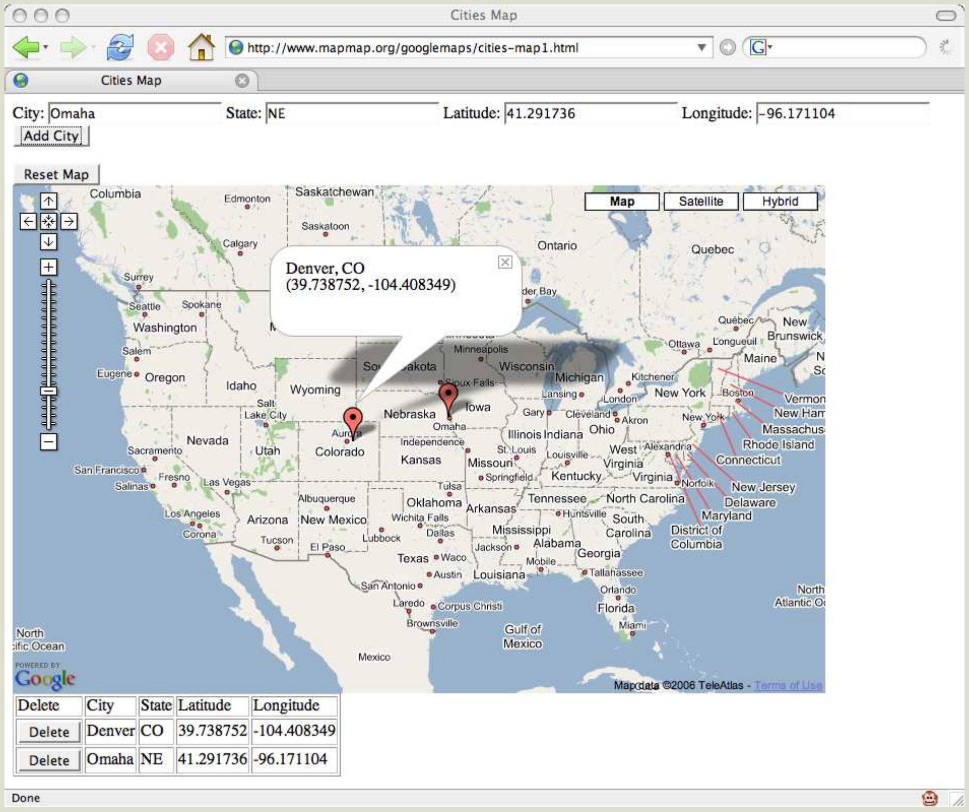
```
function addToMap(cityBean)
{
  var point = new GLatLng(cityBean.latitude, cityBean.longitude);
  cityBean.marker = new GMarker(point);
  map.addOverlay(cityBean.marker);
}
```

When we delete a city, we also call deleteFromMap().

```
function deleteFromMap(cityId)
{
  var position = findById(cityId);

  if(position > -1) {
    var cityBean = cityList[position];
    map.removeOverlay(cityBean.marker);
    map.closeInfoWindow();
  }
}
```

Just to show off, we add an event listener to each cell in the table that opens an Info Window on the map when the user moves their mouse over the row. We tweak the addToTable() function to register the event listeners.

```
function addToTable(cityBean)
{
  var tbl = document.getElementById("cityTable");
  var newRow = tbl.insertRow(tbl.rows.length);
  newRow.id = cityBean.id;


  var deleteCell = newRow.insertCell(0);
  var deleteButton = document.createElement("input");
  deleteButton.type = "button";
  deleteButton.name = "Delete";
  deleteButton.value = "Delete";
```



City: Omaha State: NE Latitude: 41.291736 Longitude: -96.171104

Denver, CO
(39.738752, -104.408349)

Delete	City	State	Latitude	Longitude
Delete	Denver	CO	39.738752	-104.408349
Delete	Omaha	NE	41.291736	-96.171104

Figure 7.3: Adding cities to the map

method—we get the cityId from the row, and call deleteFromTable(cityId) and deleteFromArray(cityId).

OK, now let's add in the map. (See Figure 7.3 , or http://www.mapmap.org/google

Most of the code should be familiar to you from previous examples. All we're doing is taking our lat/long point from the HTML form and constructing a GMarker out of it.

Our addToCity() function now has an addToMap(cityBean) function in addition to addToTable(cityBean) and addToArray(cityBean).

```
  deleteButton.type = "button";
  deleteButton.name = "Delete";
  deleteButton.value = "Delete";
  registerListener(deleteButton, "click", deleteCity, false);
  deleteCell.appendChild(deleteButton);

  var cityCell = newRow.insertCell(1);

  cityCell.appendChild(document.createTextNode(cityBean.city));


  var stateCell = newRow.insertCell(2);

  stateCell.appendChild(document.createTextNode(cityBean.state));

  var latitudeCell = newRow.insertCell(3);

  latitudeCell.appendChild(document.createTextNode(cityBean.latitude));

  var longitudeCell = newRow.insertCell(4);

  longitudeCell.appendChild(document.createTextNode(cityBean.longitude));


}
```

Adding the new cityBean to an array is painfully simple compared to the previous addToTable() code. Since JavaScript arrays grow dynamically, our method is a one-liner.

```
function addToArray(cityBean)
{
  cityList[cityList.length] = cityBean;
}
```

The deleteCity() function can be called from any row in the table that has a *Delete* button. The process is the reverse of the addCity()

```
                break;

            //turn off event listener
            case 1:
                pushpinsCycle = 0;
                GEvent.removeListener(clickListener);
                button.value = buttonOff;
                break;
        }
}
```

When the map listener is engaged, the pushpin() function gets called each time the user clicks on the map. Notice that we do nothing if an overlay is clicked on. Since we're presumably trying to add new cities to the map, clicking on an existing city is simply ignored. If an empty area of the map is clicked, a point is returned by the event. We capture the lat/long from the point, construct a new cityBean, and fill in the HTML form with our new data. From there, the user can fill in the City and State information and click on the addCity button to commit their changes.

```
function pushpin(overlay, point)
{
   if(point){
     var cityBean = new Object();
     cityBean.city = "";
     cityBean.state = "";
     cityBean.latitude = point.lat();
     cityBean.longitude = point.lng();
     populateForm(cityBean);
   }
}
```

In a production application, I'd probably hide all of that gory lat/long stuff from my delicate, fragile end users. For the current audience, I made the assumption that if you have made it this far into the book, you're tough enough to handle some exposed internals.

We'll revisit this application one more time to add the lookup service. But in order do that, we need to add the final aspect of the Google Maps API to our toolkit—AJAX.

Chapter 8

AJAX

What makes Google Maps so impressive from a technology perspective is how cleverly it takes advantage of the full constellation of client-side JavaScript tricks. These techniques have been around since the late 1990s, but up until 2005 no one had assembled all of them into a single web application. Google Maps is the "perfect storm" of client-side web technologies.

8.1 DHTML and AJAX

DHTML

DHTML (short for Dynamic HTML) is nothing more than an umbrella term for combining the JavaScript event model with CSS positioning. It is the secret sauce that allows you to drag objects around in the middle of the web page. DHTML had been little more than a "stupid browser trick"—a solution looking for a problem—until the folks at Google decided that it was the most natural way to interact with a map. Before that, DHTML had been mostly relegated to pull-down menus—visually interesting, but hardly core functionality.

AJAX

And while the rich UI experience is what initially grabs most people, how the data gets to the browser behind the scenes is given little thought. *AJAX* (Asynchronous JavaScript and XML) is arguably the unsung hero of Google Maps. It is what breaks the us out of the coarse-grained "click on the map, wait for the entire page to refresh" experience. It is quite literally multithreading for the World Wide Web. It allows us to make a series of micro requests instead of a single, large macro request.

In Google Maps, every time you click on a GMarker to pop up an

Info Window you are making an AJAX request back to the server. Notice that the whole screen doesn't refresh—the information just pops up seamlessly. In our previous Info Window example, the data lives locally with the page. Using AJAX, we can access much more information than what would be reasonable to download all at once. We can send down little pieces of data as the user request it, all behind the scenes.

So DHTML is an umbrella term for the combination of JavaScript events and CSS postioning. AJAX is the umbrella term for the combination of JavaScript events and the *XMLHttpRequest* object. (For a short explanation of AJAX, see Dynamic HTML and XML. For a more detailed treatment of the subject, see Pragmatic Ajax.)

XMLHttpRequest

http://developer.apple.com/internet/webcontent/xmlhttpreq.html

http://www.pragmaticprogrammer.com/titles/ajax/index.html

8.2 GXmlHttp

Even though the XMLHttpRequest object is present in all major browsers, it is implemented in slightly incompatible ways. Just like the JavaScript event model earlier, it is usually up to the web developer to normalize the AJAX model across browsers:

```
//1. Mozilla, Safari, et al
//2. IE

var req;

if (window.XMLHttpRequest) {
    req = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
    req = new ActiveXObject("Microsoft.XMLHTTP");
}
```

GXmlHttp

The *GXmlHttp* object normalizes the AJAX event model across browsers for us:

```
var req = GXmlHttp.create();
```

8.3 Geocoder Web Services

Even though the GXmlHttp object comes to us via the Google Maps API, there is nothing intrinsically “mappy” about it. You can use the object to make any AJAX call you would like.

We’re going to use it to look up the lat/long point for a given city. We could download the entire database of cities and their corresponding points each time the page is loaded, but that isn’t exactly the most optimal solution. A better solution is to leave this list on a server somewhere and only pull down the data we need via a web services request.

A complete list of worldwide cities and their coordinates runs about 137MB. A copy can be downloaded and used for free from http://www.maxmind.com/download/worldcities.

geocoders

We could implement a web service ourselves, but why reinvent the wheel? There are plenty of services out there just waiting to be used. Queries that return lat/long points are called *geocoders*. The Google Maps API doesn’t expose their geocoder to us, but the GXmlHttp object allows us to make a call to the geocoder of our choice.

http://geocoder.us

If you are trying to geocode a full street address, you could use geocoder.us. A simple RESTful web service request like this

```
http://geocoder.us/service/rest?address=1600+Pennsylvania+
  Ave%2C+Washington+DC
```

returns a well-formed XML document like this (a newline was added to make the description tag fit the page):

```
<rdf:RDF
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
    xmlsn:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <geo:Point rdf:nodeID="aid59834990">
    <dc:description>1600 Pennsylvania Ave NW,
                    Washington DC 20502</dc:description>
    <geo:long>-77.037684</geo:long>
    <geo:lat>38.898748</geo:lat>
  </geo:Point>
</rdf:RDF>
```

The geocoder.us service is free. It is based on free data provided by the US Census Bureau. In addition to the RESTful interface demonstrated here, geocoder.us offers SOAP, XML-RPC, and even CSV (comma-separated value) interfaces. The limitation, of course, is that the *US* Census Bureau doesn't provide data for addresses outside of the *US*. The geocoder.us interface also doesn't allow you to query based on just the city—you must provide a full street address.

http://brainoff.com/worldkit/geocoder/rest/

Don't worry. Another free geocoder is available at brainoff.com/worldkits. As the URL indicates, the brainoff.com geocoder isn't limited to just domestic US requests. Making a simple query like this

http://brainoff.com/geocoder/rest?city=London,UK

returns a well-formed XML document like this:

```
<rdf:RDF
    xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
    xmlsn:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <geo:Point>
    <geo:long>-0.1166667</geo:long>
    <geo:lat>51.5</geo:lat>
  </geo:Point>
</rdf:RDF>
```

http://developer.yahoo.com/maps/rest/V1/geocode.html

Yahoo also offers a free geocoder service. You need to sign up for a free application ID just as we did with Google. After that, you are off to the races. The query

```
http://api.local.yahoo.com/MapsService/V1/geocode?
                  appid=mapmap.org&amp;city=Denver&amp;state=CO
```

gives you this:

```
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="urn:yahoo:maps"
         xsi:schemaLocation="urn:yahoo:maps
            http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
  <Result precision="city">
    <Latitude>39.739109</Latitude>
    <Longitude>-104.983917</Longitude>
    <Address></Address>
    <City>DENVER</City>
    <State>CO</State>
    <Zip></Zip>
    <Country>US</Country>
  </Result>
</ResultSet>
```

http://en.wikipedia.org/wiki/Representational_State_Transfer

http://www-128.ibm.com/developerworks/webservices/library/ws-wsajax/

As you might have guessed by now, a RESTful web service accepts requests in the form of simple name/value pairs and returns well-formed XML. (See Wikipedia's article on REST for more information.) You can certainly make SOAP requests via AJAX as well, but you have jump through many more hoops than doing simple string concatenation. (There's an example here.) And why would you subject yourself to that much extra work if you didn't need to? (There's a reason why people say that SOAP is the EJB of the XML world, but that's a discussion for another day....)

But even with the simplicity of RESTful web services, there's a catch. For security reasons, AJAX requests must be made to the same domain as the domain of the parent page. Since our page came from www.mapmap.org, all AJAX requests must go back to mapmap.org as well. So how do we get around this limitation? Simple: we just proxy the request on our server. This could be done in the server-side language of your choice—here is how I implemented it using a single JSP and the <*c:import*> tag from the JSTL.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<% response.setContentType("text/xml"); %>

<c:import url="http://api.local.yahoo.com/MapsService/V1/geocode">
  <c:param name="appid" value="mapmap.org" />
  <c:param name="city" value="${param.city}" />
  <c:param name="state" value="${param.state}" />
</c:import>
```

If I make this request:

```
http://www.mapmap.org/googlemaps/city-ws-proxy.jsp?
                                          city=Denver&amp;state=CO
```

it will simply peel off the city and state parameters, make the request to the real web service on my behalf, and return the XML document.

8.4 Revisiting the Real-World Example

So how do we string all of this together? Here's the user story: we would like the user to be able to type in a city and state in the HTML form and have the corresponding lat/long points magically appear. To accomplish this, we'll provide a *Lookup Coordinates for City* button that extracts the field data from the form and makes an AJAX call to our proxied web service behind the scenes. We'll parse the lat and long elements from the returned XML document and populate the appropriate fields in the HTML form.

From there, it's no different than if the user had typed in the coordinates by hand. They can click on the *Add City* button to commit the new data to the map, or simply ignore the returned data. If a city isn't found, the coordinates are left blank.

We are able to accomplish all of this in a single function:

```
function lookupCity()
{
  var cityBean = getCityBean();
  var query = "city-ws-proxy.jsp";
  query += "?city=" + cityBean.city;
  query += "&state=" + cityBean.state;

  var request = GXmlHttp.create();
  request.open("GET", query, true);
  request.onreadystatechange = function() {
    if(request.readyState == 0) { errorOut("readyState 0 = uninitialized"); }
    else if(request.readyState == 1) { errorOut("readyState 1 = loading"); }
    else if(request.readyState == 2) { errorOut("readyState 2 = loaded"); }
    else if(request.readyState == 3) { errorOut("readyState 3 = interactive");
    else if(request.readyState == 4) {
      errorOut("readyState 4 = complete, status=" + request.status +
               " (" + request.statusText + ")");
      if(request.status == 200)
      {
          var responseDoc = request.responseXML;
          var root = responseDoc.documentElement;

          var latitudeNode = root.getElementsByTagName("Latitude")[0];
          cityBean.latitude = GXml.value(latitudeNode);
          var longitudeNode = root.getElementsByTagName("Longitude")[0];
          cityBean.longitude = GXml.value(longitudeNode);

          populateForm(cityBean);
      }
    }
  }
  request.send(null);
}
```

The first three lines build the string for the RESTful request. The next lines create the GXmlHttp and use a closure for event handling. (We could have just as easily created a separate function to handle the AJAX lifecycle.) The very last line of the function sends the request.

Looking now at the actual event handler, we see that an AJAX request can be in one of five possible states. The state that we're most interested in is 4, which means the request is complete and we have a response. An AJAX request is a full-fledged HTTP request, right down to the status code it returns. We can use request.status to see the numeric value (200 is OK, 404 is File Not Found, etc.), and request.statusText for the description.

But, more importantly, the request object gives us two ways to get at the payload. request.responseText returns the document as a simple string. Since we know that the response we're going to get from the web service is well-formed XML, we can use request.responseXML to get it back as a DOM object. We've been using DOM all along to manipulate our HTML document; we can use the same syntax to parse out the XML document.

Since the elements in our XML data do not have id attributes, we cannot use document.getElementById() as we have been in previous examples. We use the more generic document.getElementsByTagName(), which returns an array of elements. (Notice that method name for ids is *singular*; the method name for tags is *plural*.) Since we know that the first value returned is the one we're interested in, we can hardcode [0] into the function call.

GXml

Now that we have the XML node in hand, we need to crack the nut to get at the sweet chewy data on the inside. We could use DOM syntax to get at it, but instead we take advantage of the *GXml* object included in the Google Maps API. GXml.value() is a convenience method that returns the value of the child textNode. Now that our cityBean has values for the coordinates (or empty strings if the query was unsuccessful), we call populateForm() to add the values to the HTML form.

GDownloadUrl

Make sense? Good. Because in v2 of the API, there is an even simpler way to accomplish this. *GDownloadUrl* takes care of all of the lifecycle logic for us.

```
function lookupCityShortcut()
{
  var cityBean = getCityBean();
  var query = "city-ws-proxy.jsp";
  query += "?city=" + cityBean.city;
  query += "&state=" + cityBean.state;

  GDownloadUrl(query, function(data, status) {
      if(status == 200)
      {
          var responseDoc = GXml.parse(data);
          var root = responseDoc.documentElement;

          var latitudeNode = root.getElementsByTagName("Latitude")[0];
          cityBean.latitude = GXml.value(latitudeNode);
          var longitudeNode = root.getElementsByTagName("Longitude")[0];
          cityBean.longitude = GXml.value(longitudeNode);

          populateForm(cityBean);
      }
      else
```

```
        {
          errorOut("Bad things happened. status=" + status);
        }
      });

    request.send(null);
  }
```

Chapter 9

Where do we go from here?

Now that you have the basics of Google Maps under your belt, all you have to do now is do something creative with your new-found skills. The combination of mapping and freely available data via web services can yield all sorts of interesting applications. These are being called “Google Maps Mashups” by the cool kids on the ’net.

You could put your skills to civic-minded use. For example, the site http://www.chicagocrime.org/ allows you to see all of the crimes in Chicago on a map.

You could put your skills to more practical use. The application at http://www.housingmaps.com/ takes a list of homes for sale from http://www.craigslist.org and puts them on a map.

Or you could be completely frivolous with your new-found skills. http://hotmaps.frozenbear.com/ takes pictures of pretty men or women from http://www.hotornot.com and shows you who lives in your neighborhood.

As you can see, the possibilities are limitless. You can go quite a long way with nothing but web services and the free Google Maps API.

If, however, you hit the limit of what the free API can offer, you still have options. Coming in early 2006, the Pragmatic Press will publish *Pragmatic GIS*. It is an exploration of setting up your own, in-house Google Maps. We discuss setting up a complete GIS infrastructure using nothing but free and open source software and data. From client-side libraries to Java APIs. From spatial databases to geographically enable web servers. Until then, enjoy your time with Google Maps.

Reship Google Maps API, V2

Pragmatic Home

NEW BOOKS NOTICES

Your e-mail
Subscribe *more info*

Titles
The Pragmatic Programmer
Pragmatic Starter Kit Series
Programming Ruby
All Books

Sections
Books
Articles
Workshops
Blogs/Lists
FAQ
About Us

Working with us
For Authors
For Reviewers
For Booksellers
Press Releases

Shopping
Catalog
View Cart

Search

Google Search
[✓ Search pragmaticprogrammer.com] Search WWW

Visit our bookstore:

If you've ordered a PDF from us in the past, use this form to request a latest version of that PDF. The process uses our regular fulfillment engine. Depending on load, it can take as little as a few minutes, or as long as a day to generate your PDF. Our system will send you an e-mail when the PDF is available for collection.

You need to supply **two of the following three pieces** of information for us to be able to find your order. The information you give here must be the same as the information you used when placing the original order.

Help me find my details...

Because it's easy to type an address slightly differently between the order and this form, we recommend using just the order number and your e-mail only when matching your order.

If you're regenerating a PDF for a second edition of a book, be sure to use the appropriate order number. If you received a courtesy upgrade from a first edition PDF, your second edition PDF will have a different order number to the first. Click the button above for information on finding the order number of your PDF.

Order number:	
E-Mail address:	
1st line of billing address:	
	REGENERATE MY PDF

Home All Books Privacy Contact us *Phone Orders:* **800-699-7764**