Beginning Google Maps Applications with PHP and Ajax

From Novice to Professional

Michael Purvis
Jeffrey Sambells
and Cameron Turner

Apress®

Beginning Google Maps Applications with PHP and Ajax: From Novice to Professional

Copyright © 2006 by Michael Purvis, Jeffrey Sambells, and Cameron Turner

ISBN-13 (pbk): 978-1-59059-707-1

ISBN-10 (pbk): 1-59059-707-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section or at the official book site, http://googlemapsbook.com.

CHAPTER 3

Interacting with the User and the Server

Now that you've created your first map (in Chapter 2) and had a chance to perform some initial experiments using the Google Maps API, it's time to make your map a little more useful and dynamic. Most, if not all, of the best Google Maps mashups rely on interaction with the user in order to customize the information displayed on the map. As you've already learned, it's relatively easy to create a map and display a fixed set of points using static HTML and a bit of JavaScript. Anyone with a few minutes of spare time and some programming knowledge could create a simple map that would, for example, display the markers of all the places he visited on his vacation last year. A static map such as this is nice to look at, but once you've seen it, what would make you return to the page to look at it again? To keep people coming back and to hold their attention for longer than a few seconds, you need a map with added interactivity and a bit of flair.

You can add interactivity to your map mashups in a number of ways. For instance, you might offer some additional detail for each marker using the info window bubbles introduced in Chapter 2, or use something more elaborate such as filtering the markers based on search criteria. Google Maps, Google's public mapping site (`http://maps.google.com/`) is a mashup of business addresses and a map to visually display where the businesses are located. It provides the required interactivity by allowing you to search for specific businesses, and listing other relevant businesses nearby, but then goes even further to offer driving directions to the marked locations. Allowing you to see the location of a business you're looking for is great, but telling you how to get there in your car, now that's interactivity! Without the directions, the map would be an image with a bunch of pretty dots, and you would be left trying to figure out how to get to each dot. Regardless of how it's done, the point is that interacting with the map is always important, but don't go overboard and overwhelm your users with too many options.

In this chapter, we'll explore a few examples of how to provide interactivity in your map using the Google Maps API, and you'll see how you can use the API to save and retrieve information from your server. While building a small web application, you'll learn how to do the following:

- Trigger events on your map and markers to add either new markers or info windows.
- Modify the content of info windows attached to a map or to individual markers.
- Use Google's `GXmlHttp` object to communicate with your server.
- Improve your web application by changing the appearance of the markers.

Going on a Treasure Hunt

To help you learn about some of the interactive features of the Google Maps API, you're going to go on a treasure hunt and create a map of all the treasures you find. The treasures in this case are geocaches, those little plastic boxes of goodies that are hidden all over the earth.

For those of you who are not familiar with *geocaches* (not to be confused with geocoding, which we will discuss in the next chapter), or *geocaching* as the activity is commonly referred to, it is a global "hide-and-seek" game that can be played by anyone with a Global Positioning System (GPS) device (Figure 3-1) and some treasure to hide and seek. People worldwide place small caches of trinkets in plastic containers, and then distribute their GPS locations using the Internet. Other people then follow the latitude and longitude coordinates and attempt to locate the hidden treasures within the cache. Upon finding a cache, they exchange an item in the cache for something of their own.

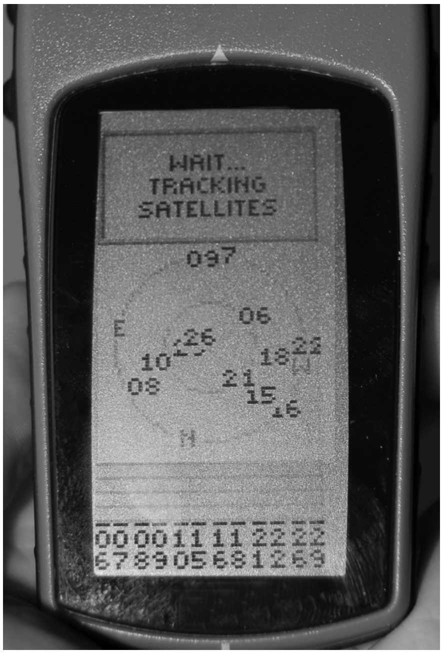


Figure 3-1. *A common handheld GPS device used by geocachers to locate hidden geocaches*

■**Note** For more information about geocaching, check out the official Geocaching website (`http://www.geocaching.com`) or pick up *Geocaching: Hike and Seek with Your GPS*, by Erik Sherman (`http://www.apress.com/book/bookDisplay.html?bID=194`).

As you create your interactive geocache treasure map, you'll learn how to do the following:

- Create a map and add a JavaScript event trigger using the `GEvent.addListener()` method to react to clicks by the users, so that people who visit the map can mark their finds on the map.
- Ask users for additional information about their finds using an info window and an embedded HTML form.
- Save the latitude, longitude, and additional information in the form to your server using the `GXmlHttp` Asynchronous JavaScript and XML (Ajax) object on the client side and PHP on the server.
- Retrieve the existing markers and their additional information from the server using Ajax and PHP.
- Re-create the map upon loading by inserting new markers from a server-side list, each with an info window to display its information.

For this chapter, we're not going to discuss any CSS styling of the map and its contents; we'll leave all that up to you.

Creating the Map and Marking Points

You'll begin the map for this chapter from the same set of files introduced in Chapter 2, which include the following:

- `index.php` to hold the XHTML of the page
- `map_functions.js` to hold the JavaScript functionality
- `map_data.php` to create a JavaScript array and objects representing each location on the map

Additionally, you'll create a file called `storeMarker.php` to save information back to the server and another file called `retrieveMarkers.php` to retrieve XML using Ajax, but we'll get to those later.

Starting the Map

To start, copy the `index.php` file from Listing 2-2 and the `map_functions.js` file from Listing 2-3 into a new directory for this chapter. Also, create an empty `map_data.php` file and empty `storeMarker.php` and `retrieveMarkers.php` files.

While building the map for this chapter and other projects, you'll be adding auxiliary functions to the `map_functions.js` file. You may have noticed in Chapter 2 that you declared the `map` variable outside the `init()` function in Listing 2-2. Declaring `map` outside the `init()` function allows you to reference `map` at any time and from any auxiliary functions you add to the `map_functions.js` file. It will also ensure you're targeting the same `map` object. Also, you may want to add some of the control objects introduced in Chapter 2, such as `GMapTypeControl`. Listing 3-1 highlights the `map` variable and additional controls.

Listing 3-1. *Highlights for map_functions.js*

```
var centerLatitude = 37.4419;
var centerLongitude = -122.1419;
var startZoom = 12;

var map;

function init() {
    if (GBrowserIsCompatible()) {
         map = new GMap2(document.getElementById("map"));
         map.addControl(new GSmallMapControl());
         map.addControl(new GMap2TypeControl());
         map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
    }
}

window.onload = init;
window.onunload = GUnload;
```

Now you have a solid starting point for your web application. When viewed in your web browser, the page will have a simple map with controls centered on Palo Alto, California (Figure 3-2). For this example, the starting `GLatLng` is not important, so feel free to change it to some other location if you wish.

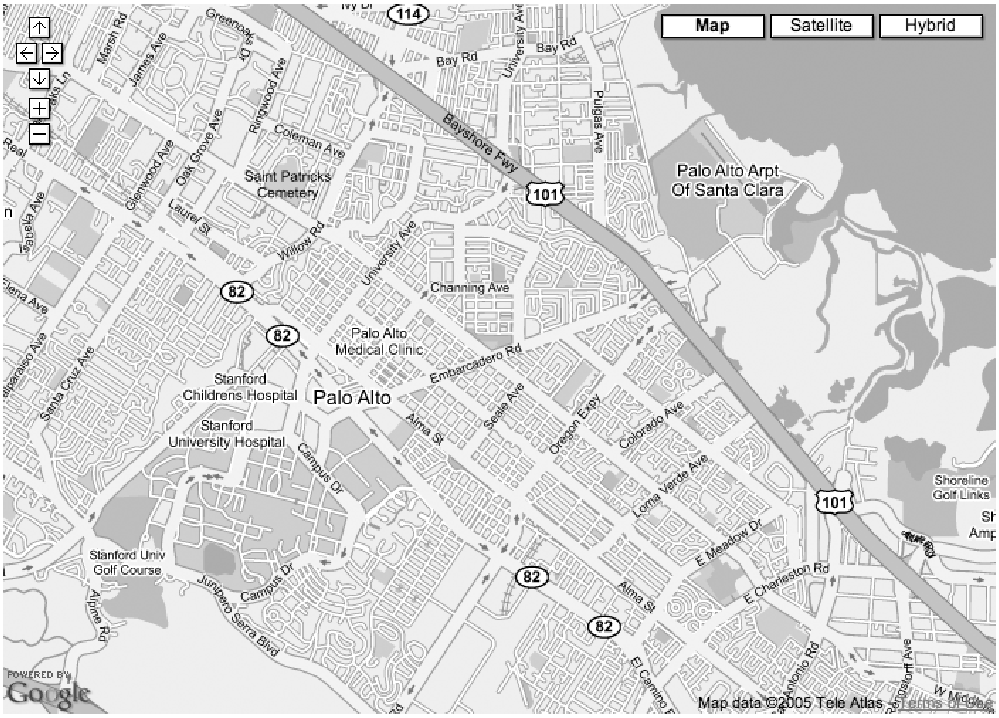

Map
Satellite
Hybrid
Palo Alto
Palo Alto Arpt Of Santa Clara
Bayshore Fwy
Saint Patricks Cemetery
Stanford Childrens Hospital
Stanford University Hospital
Palo Alto Medical Clinic
Stanford Univ Golf Course
Shoreline Golf Links
Map data ©2005 Tele Atlas
POWERED BY Google

Figure 3-2. *Starting map with controls centered on Palo Alto, California*

Listening to User Events

The purpose of your map is to allow visitors to add markers wherever they click. To capture the clicks on the map, you'll need to trigger a JavaScript function to execute whenever the map area is clicked. As you saw in Chapter 2, Google's API allows you to attach these triggers, called *event listeners*, to your map objects through the use of the `GEvent.addListener()` method. You can add event listeners for a variety of events, including `move` and `click`, but in this case, you are interested only in users clicking the map, not moving it or dragging it around.

■ **Tip** If you refer to the Google Maps API documentation in Appendix B, you'll notice a wide variety of events for both the `GMap2` and the `GMarker` objects, as well as a few others. Each of these different events can be used to add varying amounts of interactivity to your map. For example, you could use the `moveend` event for the `GMap2` to trigger an Ajax call and retrieve points for the new area of the map. For the geocaching map example, you could also use the `GMarker`'s `infowindowclose` event to check to see if the information in the form has been saved and if not, ask the user what to do. You can also attach events to Document Object Model (DOM) elements using `GEvent.addDomListener()` and trigger an event using JavaScript with the `GEvent.trigger()` method.

The `GEvent.addListener()` method handles all the necessary code required to watch for and trigger each of the events. All you need to do is tell it which object to watch, which event to listen for, and which function to execute when it's triggered.

```
GEvent.addListener(map, "click", function(overlay, latlng) {
     //your code
});
```

Given the source `map` and the event `click`, this example will trigger the function to run any code you wish to implement.

Take a look at the modification to the `init()` function in Listing 3-2 to see how easy it is to add this event listener to your existing code and use it to create markers the same way you did in Chapter 2. The difference is that in Chapter 2, you used `new GLatLng()` to create the latitude and longitude location for the markers, whereas here, instead of creating a new `GLatLng`, you can use the `latlng` variable passed into the event listener's handler function. The `latlng` variable is a `GLatLng` representation of the latitude and longitude where you clicked on the map. The `overlay` variable is the `overlay` where the clicked location resides if you clicked on a marker or another `overlay` object.

Listing 3-2. *Using the addListener() Method to Create a Marker at the Click Location*

```
function init() {
     if (GBrowserIsCompatible()) {
          map = new GMap2(document.getElementById("map"));
          map.addControl(new GSmallMapControl());
          map.addControl(new GMap2TypeControl());
          map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
```

```
            //allow the user to click the map to create a marker
            GEvent.addListener(map, "click", function(overlay, latlng) {
                var marker = new GMarker(latlng)
                map.addOverlay(marker);
            });
        }
    }
```

Ta-da! Now, with a slight code addition and one simple click, anyone worldwide could visit your map page and add as many markers as they want (Figure 3-3). However, all the markers will disappear as soon as the user leaves the page, never to be seen again. To keep the markers around, you need to collect some information and send it back to the server for storage using the `GXmlHttp` object or the `GDownloadUrl` object, which we'll discuss in the "Using Google's Ajax Object" section later in this chapter.

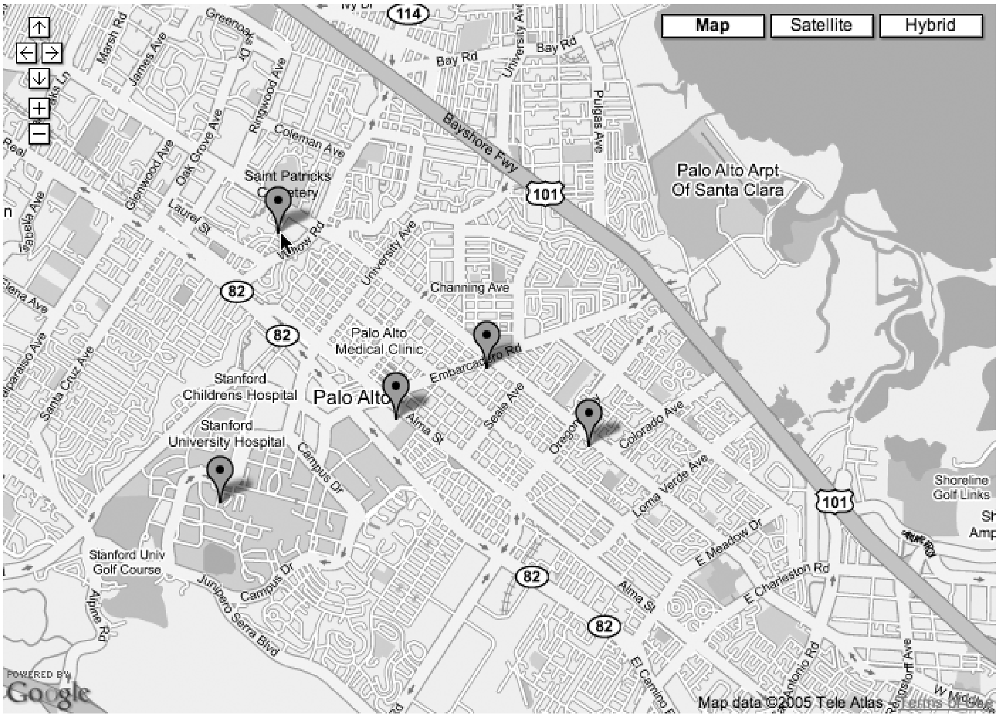


Figure 3-3. *New markers created by clicking on the map*

RETRIEVING THE LATITUDE AND LONGITUDE FROM A MAP CLICK

When you click on a Google map, the `latlng` variable passed into the event listener's handler function is a `GLatLng` object with `lat()` and `lng()` methods. Using the `lat()` and `lng()` methods makes it relatively easy for you to retrieve the latitude and longitude of any point on earth simply by zooming in and clicking on the map. This is particularly useful when you are trying to find the latitude and longitude of places that do not have readily accessible latitude/longitude information for addresses.

In countries where there is excellent latitude and longitude information, such as the United States, Canada, and more recently, France, Italy, Spain and Germany, you can often use an address lookup service to retrieve the latitude and longitude of a street address. But in other locations, such as the United Kingdom, the data is limited or inaccurate. In the case where data can't be readily retrieved by computer, manual human entry of points may be required. For more information about geocoding and using addresses to find latitude and longitude, see Chapter 4.

Additionally, If you want to retrieve the X and Y coordinates of a position on the map in pixels on the screen, you can use the `fromLatLngToDivPixel()` method of the `GMap2` object. By passing in a `GLatLng` object, `GMap2.fromLatLngToDivPixel(latlng)` will return a `GPoint` representation of the X and Y offset relative to the DOM element containing the map.

Asking for More Information with an Info Window

You could simply collect the latitude and longitude of each marker on your map, but just the location of the markers would provide only limited information to the people browsing your map. Remember interactivity is key, so you want to provide a little more than just a marker. For the geocaching map, visitors really want to know what was found at each location. To provide this extra information, let's create a little HTML form. When asking for input of any type in a web browser, you need to use HTML form elements. In this case, let's put the form in an info window indicating where the visitor clicked.

As introduced in Chapter 2, the info window is the cartoon-like bubble that often appears when you click map markers (Figure 3-4). It is used by Google Maps to allow you to enter the To Here or From Here information for driving directions, or to show you a zoomed view of the map at each point in the directions. Info windows do not need to be linked to markers on the map. They can also be created on the map itself to indicate locations where no marker is present.

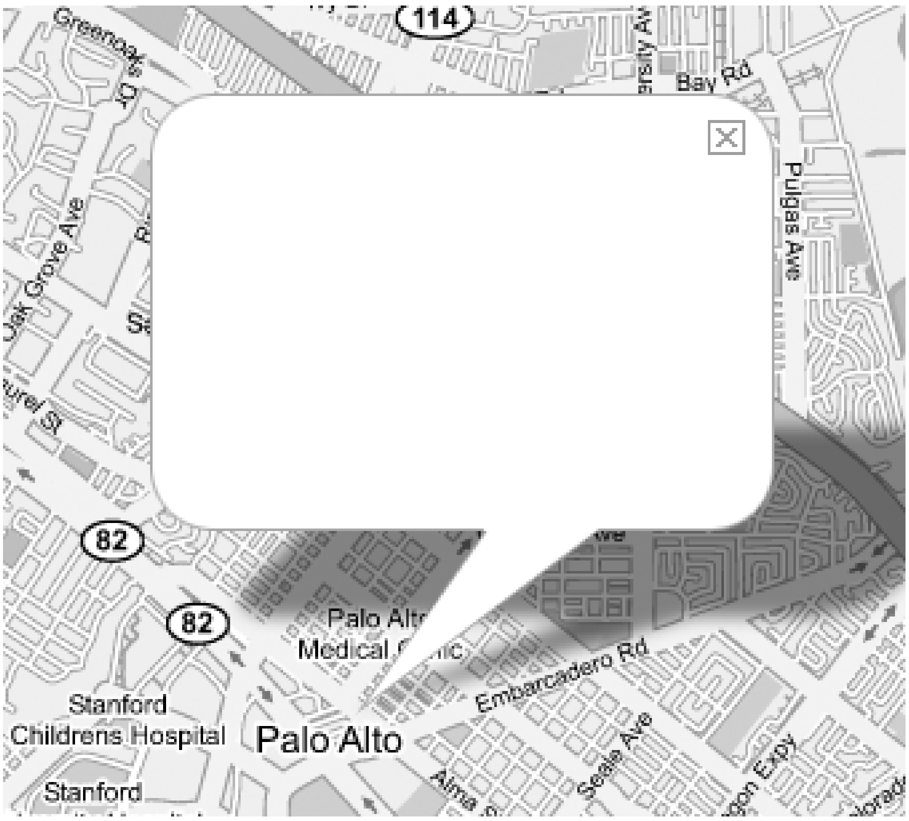


Figure 3-4. *An empty info window*

You're going to use the info window for two purposes:

- It will display the information about each existing marker when the marker is clicked.
- It will hold a little HTML form so that your geocachers can tell you what they've found.

■ **Note** When we introduce the `GXmlHttp` object in the "Using Google's Ajax Object" section later in this chapter, we'll explain how to save the content of the info window to your server.

Creating an Info Window on the Map

In Listing 3-2, you used the event listener to create a marker on your map where it was clicked. Rather than creating markers when you click the map, you'll modify your existing code to create an info window. To create an info window directly on the map object, call the `openInfoWindow()` method of the map:

```
GMap2.openInfoWindow(GLatLng, htmlDomElem, GInfoWindowOptions);
```

`openInfoWindow()` takes a `GLatLng` as the first parameter and an HTML DOM document element as the second parameter. The last parameter, `GInfoWindowOptions`, is optional unless you want to modify the default settings of the window.

For a quick demonstration, modify Listing 3-2 to use the following event listener, which opens an info window when the map is clicked, rather than creating a new marker:

```
GEvent.addListener(map, "click", function(overlay, latlng) {
     map.openInfoWindow (latlng,document.createTextNode("You clicked here!"));
});
```

Now when you click the map, you'll see an info window pop up with its base pointing at the position you just clicked with the content "You clicked here!" (Figure 3-5).

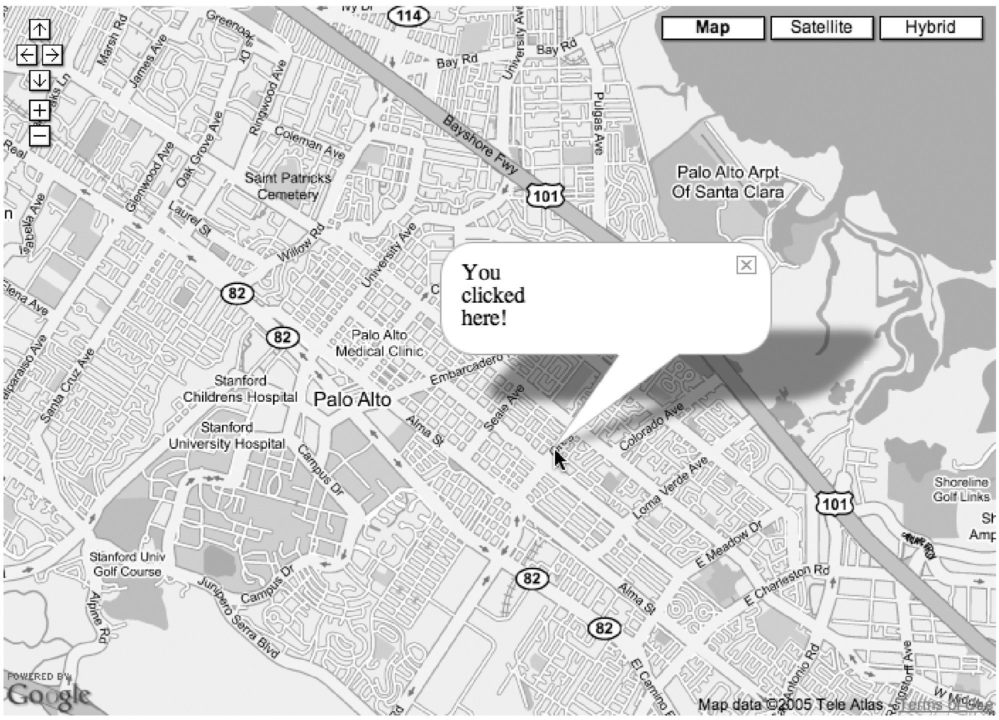

Map
Satellite
Hybrid
You clicked here!
Palo Alto
Palo Alto Arpt Of Santa Clara
Saint Patricks Cemetery
Palo Alto Medical Clinic
Stanford Childrens Hospital
Stanford University Hospital
Stanford Univ Golf Course
Shoreline Golf Links
Map data ©2005 Tele Atlas

Figure 3-5. *An info window created when clicking the map*

Embedding a Form into the Info Window

When geocachers want to create a new marker, you'll first prompt them to enter some information about their treasure. You'll want to know the geocache's location (this will be determined using the point where they clicked the map), what they found at the location, and what they left behind. To accomplish this in your form, you'll need the following:

- A text field for entering information about what they found
- A text field for entering information about what they left behind
- A hidden field for the longitude
- A hidden field for the latitude
- A submit button

The HTML form used for the example is shown in Listing 3-3, but as you can see in Listing 3-4, you are going to use the JavaScript Document Object Model (DOM) object and methods to create the `form` element. You need to use DOM because the `GMarker.openInfoWindow()` method expects an HTML DOM element as the second parameter, not simply a string of HTML.

■ **Tip** If you want to make the form a little more presentable, you could easily add ids and/or classes to the `form` elements and use CSS styles to format them accordingly.

Listing 3-3. *HTML Version of the Form for the Info Window*

```
<form action="" onsubmit="storeMarker(); return false;">
     <fieldset style="width:150px;">
          <legend>New Marker</legend>
          <label for="found">Found</label>
          <input type="text" id="found" style="width:100%;"/>
          <label for="left">Left</label>
          <input type="text" id="left" style="width:100%;"/>
          <input type="submit" value="Save"/>
          <input type="hidden" id="longitude"/>
          <input type="hidden" id="latitude"/>
     </fieldset>
</form>
```

■ **Note** You may notice the form in Listing 3-3 has an `onsubmit` event attribute that calls a `storeMarker()` JavaScript function. The `storeMarker()` function does not yet exist in your script, and if you try to click the *Save* button, you'll get a JavaScript error. Ignore this for now, as you'll create the `storeMarker()` function in the "Saving Data with GXmlHttp" section later in the chapter, when you save the form contents to the server.

Listing 3-4. *Adding the DOM HTML Form to the Info Window*

```
GEvent.addListener(map, "click", function(overlay, latlng) {

     //create an HTML DOM form element
     var inputForm = document.createElement("form");
     inputForm.setAttribute("action","");
     inputForm.onsubmit = function() {storeMarker(); return false;};

     //retrieve the longitude and lattitude of the click point
     var lng = latlng.lng();
     var lat = latlng.lat();

     inputForm.innerHTML = '<fieldset style="width:150px;">'
          + '<legend>New Marker</legend>'
          + '<label for="found">Found</label>'
          + '<input type="text" id="found" style="width:100%;"/>'
          + '<label for="left">Left</label>'
          + '<input type="text" id="left" style="width:100%;"/>'
          + '<input type="submit" value="Save"/>'
          + '<input type="hidden" id="longitude" value="' + lng + '"/>'
          + '<input type="hidden" id="latitude" value="' + lat + '"/>'
          + '</fieldset>';

     map.openInfoWindow (latlng,inputForm);
});
```

Caution When creating the DOM `form` element, you need to use the `setAttribute()` method to define things like `name`, `action`, `target`, and `method`, but once you venture beyond these basic four, you may begin to notice inconsistencies. For example, using `setAttribute()` to define `onsubmit` works fine in Mozilla-based browsers but not in Microsoft Internet Explorer browsers. For cross-browser compatibility, you need to define `onsubmit` using a function, as you did in Listing 3-4. For more detailed information regarding DOM and how to use it, check out the DOM section of the W3Schools website at `http://www.w3schools.com/dom/`.

After you've changed the `GEvent.addListener()` call in Listing 3-2 to the one in Listing 3-4, when you click your map, you'll see an info window containing your form (Figure 3-6).

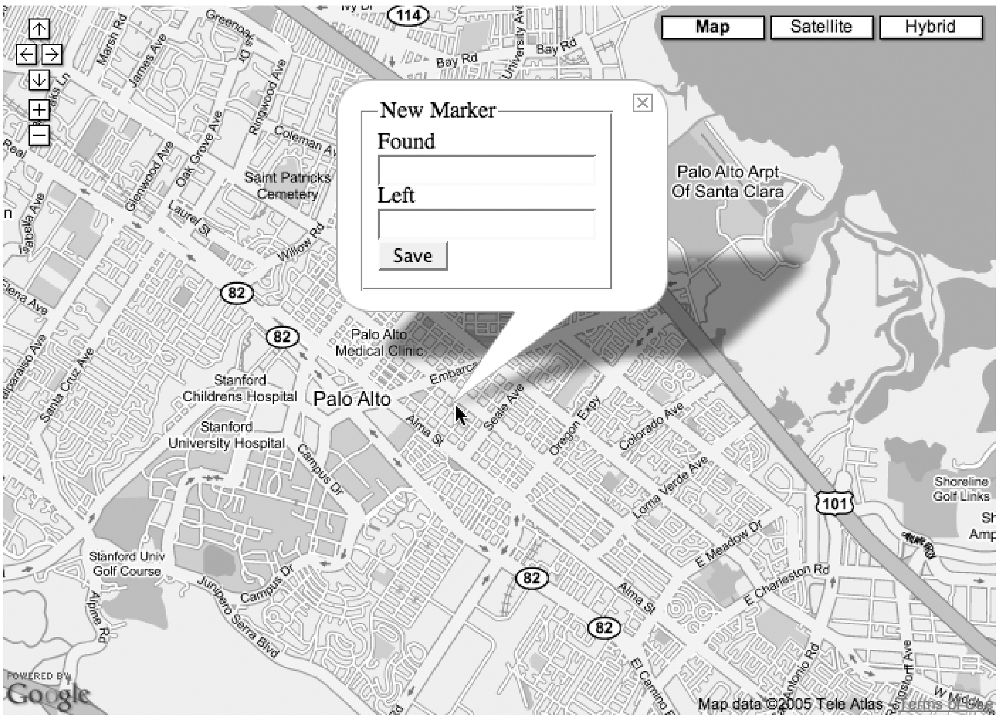


Figure 3-6. *The info window with an embedded form*

In Listing 3-4, the latitude and longitude elements of the form have been pre-populated with the `latlng.lat()` and `latlng.lng()` values from the `GLatLng` object passed in to the event listener. This allows you to later save the latitude and longitude coordinates and re-create the marker in the exact position when you retrieve the data from the server. Also, once the information has been saved for the new location, you can use this latitude and longitude to instantly create a marker at the new location, bypassing the need to refresh the web browser to show the newly saved point.

If you click again elsewhere on the map, you'll also notice your info window disappears and reappears at the location of the new click. As a restriction of the Google Maps API, you can have only one instance of the info window open at any time. When you click elsewhere on the map, the original info window is destroyed and a brand-new one is created. Be aware that it is not simply moved from place to place.

You can demonstrate the destructive effect of creating a new info window yourself by filling in the form (Figure 3-7), and then clicking elsewhere on the map without clicking the *Save* button. You'll notice that the information you entered in the form disappears (Figure 3-8) because the original info window is destroyed and a new one is created.

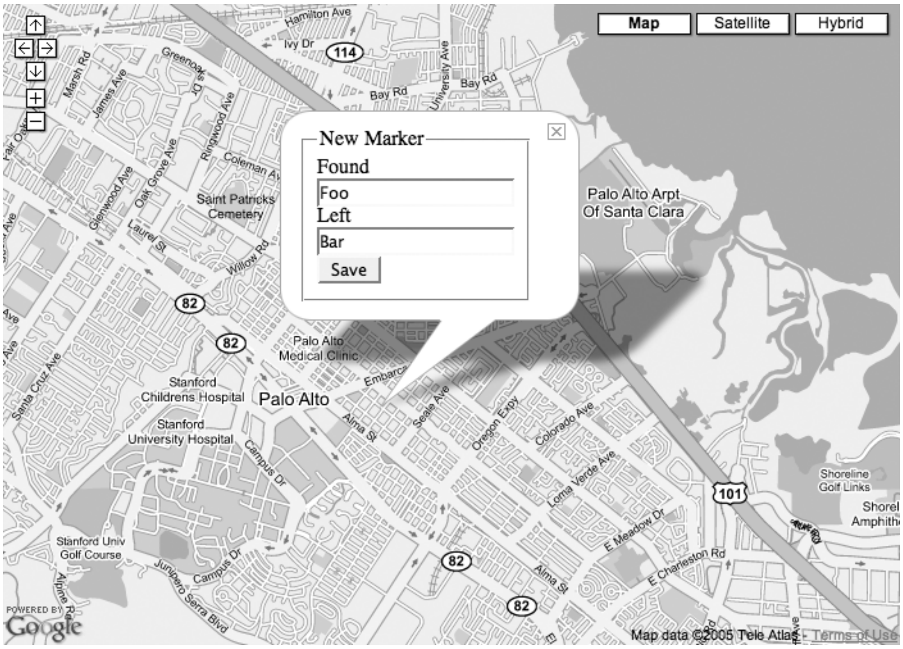


Figure 3-7. *Info window with populated form information*

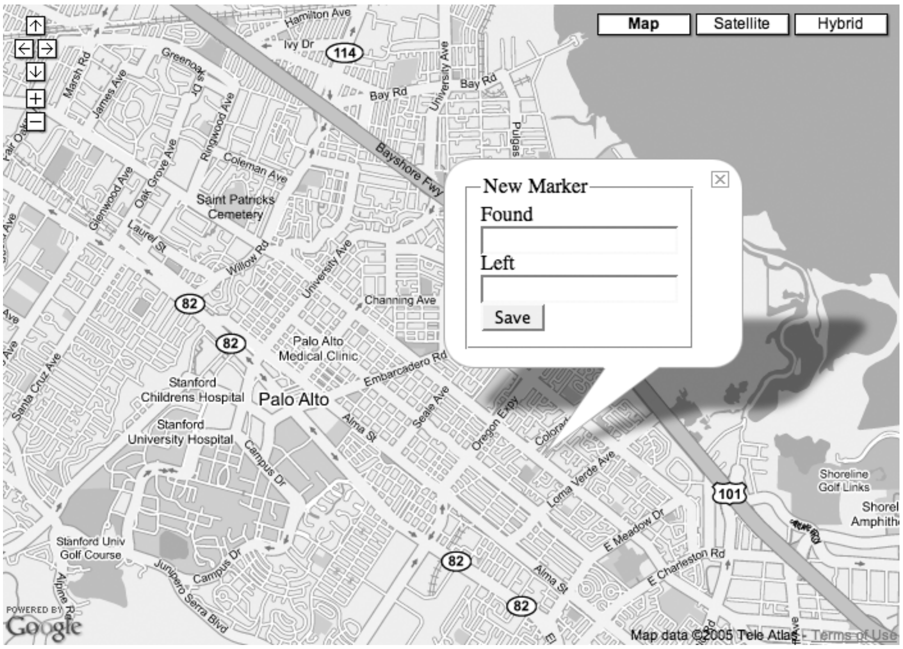


Figure 3-8. *New info window that has lost the previously supplied information*

Earlier, when you created the info window containing "You clicked here!" the same thing happened. Each marker had the same content ("You clicked here!"), so it just *appeared* as though the info window was simply moving around.

■ **Tip** If you've taken some time to review the Google Maps API in Appendix B, you might be wondering why you couldn't use the `GMap2.openInfoWindowHtml()` method to add the form to the info window. After all, it lets you use an HTML string rather than an HTML DOM element. The short answer is you can. In version 1 of the API, `openInfoWindowHtml()` required a marker to exist on the map first, whereas `openInfoWindow()` allowed you to open an info window at a specified point without a marker. We chose to use the `openInfoWindow()` method here so that you would be able to see how the DOM structure and click actions interact with the info window.

Avoiding an Ambiguous State

When creating your web applications, be sure not to create the marker until after you've verified the information and saved it to the server. If you create the marker first and the user then closes the info window using the window's close button (Figure 3-9), there would be a marker on the map that wasn't recorded on the server (Figure 3-10).

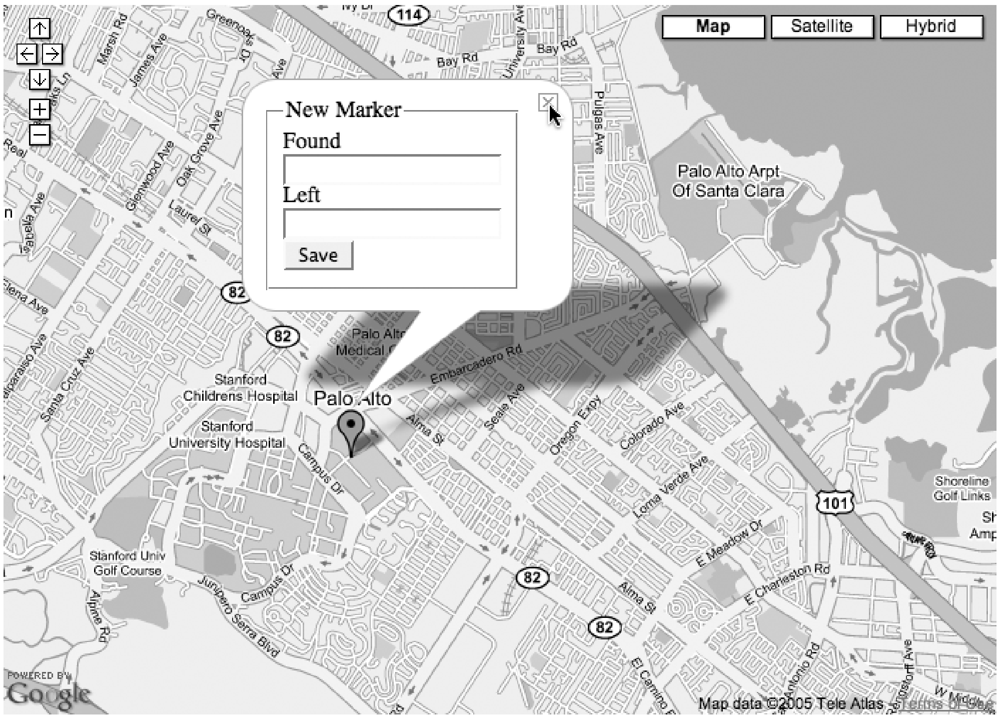

Map
Satellite
Hybrid
New Marker
Found
Left
Save
Palo Alto
Palo Alto Arpt Of Santa Clara
Stanford Childrens Hospital
Stanford University Hospital
Stanford Univ Golf Course
Embarcadero Rd
Shoreline Golf Links
POWERED BY Google
Map data ©2005 Tele Atlas

Figure 3-9. *Using the close (X) button to close the info window*

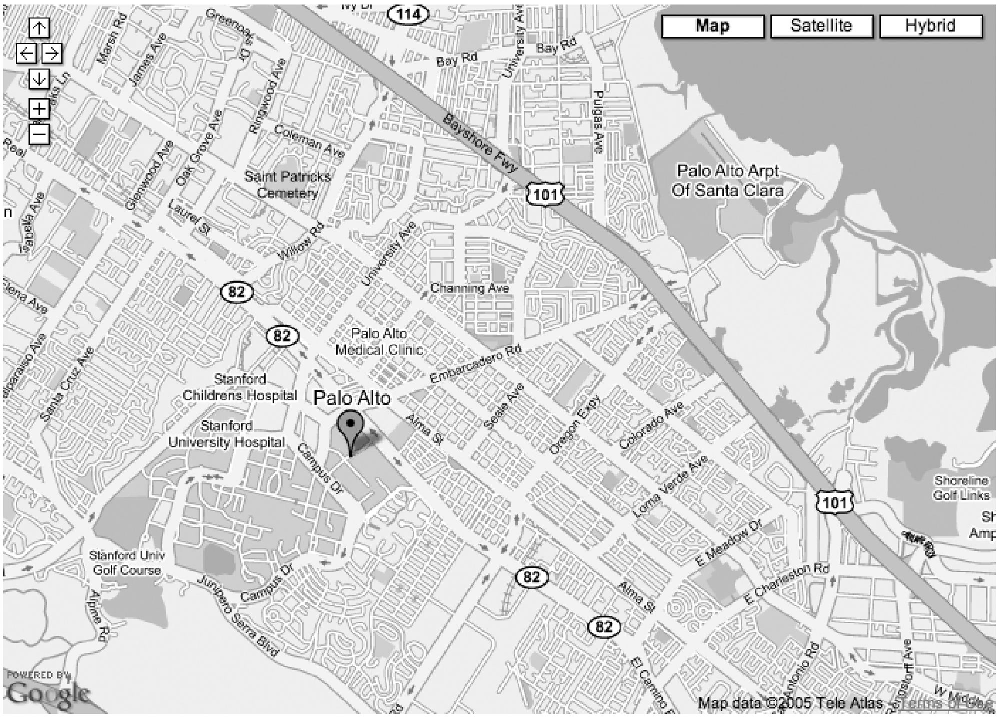

Map
Satellite
Hybrid
Palo Alto
Palo Alto Arpt Of Santa Clara
Stanford University Hospital
Map data ©2005 Tele Atlas

Figure 3-10. *Marker left behind by closing the window*

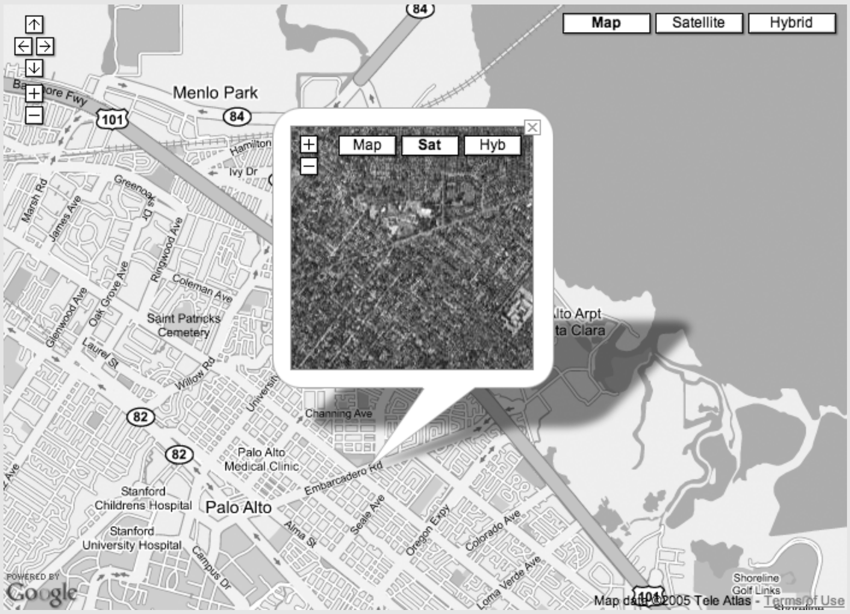
By creating the marker only after the data has been saved, you ensure the integrity of the map and keep the visible markers in sync with the stored markers on the server.

If you want, you can save the marker information in two steps: first send just the latitude and longitude to save the marker's location, and then send a second request to save the additional information, if any. Storing the latitude and longitude immediately may seem like a logical idea, until you realize that users may click the map in the wrong location and inadvertently add a bunch of points that don't really mean anything. For the geocaching map, you want to be sure there is information associated with each point, so you need to save all the information in one request.

Caution Don't confuse the `GMap2.openInfoWindow()` method with the `GMarker.openInfoWindow()` method. The map and marker objects have similar properties and methods; however, their parameters differ. You need to use the `GMap2` methods when creating info windows attached to the map itself, but if you have an existing marker, you could then use the `GMarker` methods to attach an info window to the marker. The `GMarker` methods can't be used to create an info window without a marker.

INFO WINDOWS THAT ZOOM

`GMap2.showMapBlowup()` and `GMarker.showMapBlowup()` are two other methods in the Google Maps API that will let you create info windows. These info windows are special and contain a zoomed-in view of the map. For example, `map.showMapBlowup(new GLatLng(37.4419, -122.1419), 3, G_SATELLITE_TYPE)` will display a small satellite map at zoom level 3 centered on Palo Alto, California. If you create the map blowup in an event listener, you can zoom in on any point you click on your map.



Controlling the Info Window Size

When you add content to the info window, it will automatically expand to encompass the content you've placed in it. The content container will expand in the same way a `<div>` tag expands to its internal content. To provide a bit of control over how it expands, you can add CSS styles to the content of the info window in the same way you would in a regular HTML page.

In Listings 3-3 and 3-4, the `<fieldset>` element was assigned a width of `150px`, forcing the info window's content container to 150 pixels wide (Figure 3-11). Also, the text `<input>` elements were set to a width of `100%` to display a simple clean form. (For more tips and tricks regarding info windows, see Chapter 9.)

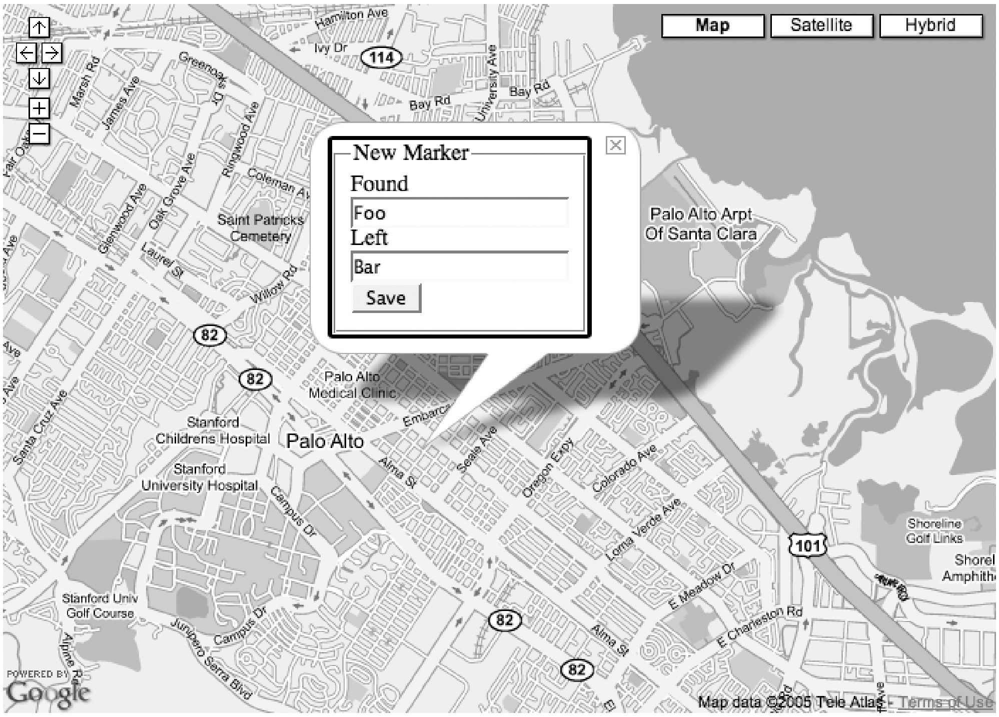
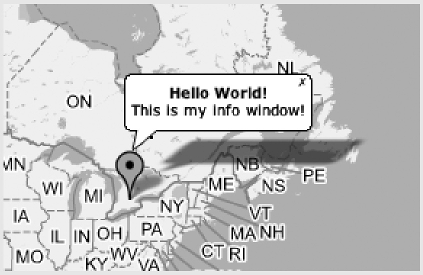


Figure 3-11. *Info window with an inner width of 150 pixels*

HOW TO CHANGE THE STYLE OF THE INFO WINDOW

Getting tired of the cartoon bubble and want to create something fancier with the info window API? Sorry, you're out of luck—well, sort of.

Currently, the Google Maps API doesn't allow you to change the style of the info window, but you could create your own using the `GOverlay` class. If you're interested, check out Chapter 9, where you'll learn how to create your own info window, as in the following example.



Using Google's Ajax Object

To save the markers entered by your geocaching visitors, you're going to upgrade to "Web 2.0" and use only Ajax to communicate with your server. Ajax relies completely on JavaScript running in your visitor's web browser; without JavaScript running, Ajax simply won't work. You can argue that using a strictly JavaScript-based Ajax interface might not be a good idea. You read everywhere that in order to be good coders and offer compliant services, you should always have an alternative solution to JavaScript-based user interfaces, and that's completely true, but the Google Maps API itself doesn't offer an alternative for JavaScript-disabled browsers. Therefore, if geocachers are visiting your page without the ability to use JavaScript, they're not going to see the map! Feel free to build alternative solutions for all your other web tools, and we strongly suggest that you do, but when dealing strictly with the Google Maps API, there isn't really much point in a non-JavaScript solution, since without JavaScript, the map itself is useless.

To communicate with your server, Google has provided you access to its integrated Ajax object called `GXmlHttp`. If you want to spend the time, you could roll your own Ajax code. If you're a fan of one of the many free libraries such as Prototype (`http://prototype.conio.net`), you could also use one of those. For this example, we'll stick to the Google Maps API and the `GXmlHttp` object, as it's already loaded for you and doesn't require you to include anything else.

■ **Caution** The Google `GXmlHttp` object, and any other Ajax script based on the `XmlHttpRequest` object, allows you to query only within the domain where the map is served. For example, if your map were at `http://example.com/webapp/`, then the `GXmlHttp.request()` method can retrieve data only from scripts located in the `http://example.com` domain. You can't retrieve data from another domain such as `http://jeffreysambells.com`, as the request would break the web browser's "Same Origin" security policy (`http://www.mozilla.org/projects/security/components/same-origin.html`). Using a little JavaScript trickery to dynamically add `<script>` tags to the page does allow you to get around this policy but requires you to do special things on the server side as well. For an example of how to do this, check out the `XssHttpRequest` object at `http://jeffreysambells.com/posts/2006/03/06/centralized_ajax_services/`.

To implement the `GXmlHttp` object, a few things need to happen when users click the Save button:

- The information in your form needs to be sent to the server and verified for integrity.
- The information needs to be stored as necessary.
- Your server-side script needs to respond back to the client-side JavaScript to let the client know that everything was successful and send back any necessary information.
- The client-side JavaScript needs to indicate to the user that there was either an error or a successful response.

To accomplish this, let's send the information back to the server and store it in a flat XML file. Then, when responding that everything is okay, let's create a new marker on the map with the new information to confirm to the user that the data was successfully saved.

Saving Data with GXmlHttp

To send information to the server using the `GXmlHttp` object, first you need to retrieve the information from the form in the info window you created. Referring back to Listings 3-3 and 3-4, you'll notice that each of the form elements has a unique id associated with it. Since you're using the Ajax method to send data, the form will not actually submit to the server using the traditional `POST` method. To submit the data, you retrieve the values of the form by using the JavaScript `document.getElementById()` method and concatenate each of the values onto the `GET` string of the `GXmlHttp` request object. Then using the `onreadystatechange()` method of the `GXmlHttp` object, you can process the request when it is complete.

Listing 3-5 shows the `storeMarker()` and `createMarker()` functions to add to your `map_functions.js` file. Also, if you haven't already done so, create the `storeMarker.php` file in the same directory as your HTML document and create an empty `data.xml` file to store your marker data. Be sure to give the `data.xml` file the appropriate write permissions for your server environment.

■Tip For more information about the `XmlHttpRequest` object and using it to send data via the `POST` method, see the W3Schools page at `http://www.w3schools.com/xml/xml_http.asp`.

Listing 3-5. *Sending Data to the Server Using GXmlHttp*

```
function storeMarker(){
    var lng = document.getElementById("longitude").value;
    var lat = document.getElementById("latitude").value;

    var getVars =  "?found=" + document.getElementById("found").value
        + "&left=" + document.getElementById("left").value
        + "&lng=" + lng
        + "&lat=" + lat ;

    var request = GXmlHttp.create();

    //open the request to storeMarker.php on your server
    request.open('GET', 'storeMarker.php' + getVars, true);
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            //the request is complete

            var xmlDoc = request.responseXML;
            //retrieve the root document element (response)
            var responseNode = xmlDoc.documentElement;

            //retrieve the type attribute of the node
            var type = responseNode.getAttribute("type");

            //retrieve the content of the responseNode
            var content = responseNode.firstChild.nodeValue;

            //check to see if it was an error or success
            if(type!='success') {
                alert(content);
            } else {
                //create a new marker and add its info window
                var latlng = new GLatLng(parseFloat(lat),parseFloat(lng));
                var marker = createMarker(latlng, content);
                map.addOverlay(marker);
                map.closeInfoWindow();
            }
        }
    }
    request.send(null);
    return false;
}

function createMarker(latlng, html) {
     var marker = new GMarker(latlng);
     GEvent.addListener(marker, 'click', function() {
          var markerHTML = html;
          marker.openInfoWindowHtml(markerHTML);
    });
    return marker;
}
```

The `storeMarker()` function you just added is responsible for sending the marker information to the server through Ajax. It retrieves the information from the form and sends it to the `storeMarker.php` script in Listing 3-6 using the `GXmlHttp` object. You can also see that the `createMarker()` function is used to create the `GMarker` object and populate the info window. By creating the `GMarker` in another function, you can reuse the same function later when retrieving markers from the server (in Listing 3-8, later in the chapter).

Listing 3-6. *storeMarker.php Server-Side Script Used to Store the Marker Information in XML Format*

```
<?php

header('Content-Type: text/xml');

$lat = (float)$_GET['lat'];
$lng = (float)$_GET['lng'];
$found = $_GET['found'];
$left = $_GET['left'];


//create an XML node
$marker = <<<MARKER
<marker lat="$lat" lng="$lng" found="$found" left="$left"/>\n
MARKER;

//open the data.xml file for appending
$f=@fopen('data.xml', 'a+');
if(!$f) die('<?xml version="1.0"?>
<response type="error"><![CDATA[Could not open data.xml file]]></response>
');

//add the node
$w=@fwrite($f, $marker);
if(!$w) die('<?xml version="1.0"?>
<response type="error"><![CDATA[Could not write to data.xml file]]></response>');

@fclose($f);

//return a response
$newMarkerContent = "<div><b>found </b>$found</div><div><b>left </b>$left</div>";
echo <<<XML
<?xml version="1.0"?>
<response type="success" icon="$icon"><![CDATA[$newMarkerContent]]></response>
XML;

?>
```

For simplicity in the example, we use a flat file on the server in Listing 3-6 to store the data. This file (called `data.xml`) is simply a list of all the points saved to the server and resembles the following:

```
<marker lat="37.441" lng="-122.141" found="Keychain" left="Book"/>
<marker lat="37.322" lng="-121.213" found="Water Bottle" left="Necklace"/>
```

Note there is no surrounding root node, so the file is not actually valid XML. When you retrieve the XML later in the chapter, you'll be retrieving all the XML at once and wrapping it in a parent `<markers>` node, so you'll end up with a valid XML result:

```
<?xml version="1.0"?>
<markers>
    <marker lat="37.441" lng="-122.141" found="Keychain" left="Book"/>
    <marker lat="37.322" lng="-121.213" found="Water Bottle" left="Necklace"/>
</markers>
```

Since you're going to retrieve all the XML without any matching or searching to determine which bits to retrieve, it makes sense to store the data in one file in the format you want. In a real-world web application, you would probably want to store the information in a SQL database and retrieve only a smaller subset of points based on some search criteria. Once you've mastered sending and retrieving data, you could easily extend this example with a searchable SQL database, and then retrieve only the points in the latitude and longitude bounds of the viewable map area.

Checking When the Request Is Completed

When you click the Save button on the info window, the information in the form is sent back to the server using the `GXmlHttp` object in Listing 3-5 and awaits a response back in XML format from the PHP script in Listing 3-6. During the request, the `readyState` property of the request object will contain one of five possible incrementing values:

- `0`, for uninitialized
- `1`, for loading
- `2`, for loaded
- `3`, for interactive
- `4`, for completed

The changes to the `readyState` property are monitored using the `GXmlHttp.onreadystatechange()` event listener (Figure 3-12). At each increment, the function you've defined for the `onreadystatechange()` method will be triggered to allow you to execute any additional JavaScript code you would like. For the example, you need to deal with only the completed state of the request, so your function checks to see when `readyState==4`, and then parses the XML document as necessary.

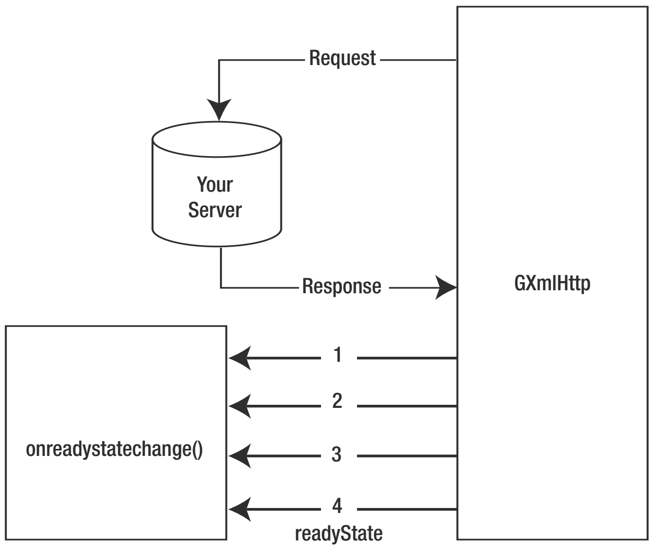


Figure 3-12. *GXmlHttp request response diagram*

■ **Tip** This Ajax implementation is not actually checking to see if the request completed in a valid state. For example, if the page you requested was not found, the `readyState` will still be `4` at the end of the request, but no XML would have been returned. To check the state of the `GXmlHttp` request, you need to check the `GXmlHttp.status` property. If `status` is equal to `200`, then the page was successfully loaded. A status of `404` indicates the page was not found. `GXmlHttp.status` could contain any valid HTTP request codes returned by the server.

Testing the Asynchronous State

Don't forget that the `GXmlHttp` request is asynchronous, meaning that the JavaScript continues to run while your `GXmlHttp` request is awaiting a response. While the PHP script is busy saving the file to the server, any code you've added after `request.send(null);` may execute before the response is returned from the server.

You can observe the asynchronous state of the request and response by adding a JavaScript `alert()` call right after you send the request:

```
request.send(null);
alert('Continue');
return false;
```

And another alert in the `onreadystatechange()` method of the request:

```
request.onreadystatechange = function() {
     if (request.readyState == 4) {
            alert('Process Response');
        }
}
```

If you run the script over and over, sometimes the alert boxes will appear in the order Process Response then Continue, but more likely, you'll get Continue then Process Request. Just remember that if you want something to occur after the response, you must use the `onreadystatechange()` method when the `readyState` is `4`.

■ **Tip** To further enhance your web application, you could use the various states of the request object to add loading, completed, and error states. For example, when initiating a request, you could show an animated loading image to indicate to the user that the information is loading. Providing feedback at each stage of the request makes it easier for the user to know what's happening and how to react. If no loading state is displayed, users may assume they have not actually clicked the button or will become frustrated and think nothing is happening.

Using GDownloadUrl for Simple Ajax Requests

If your web application doesn't require a high level of control over the Ajax request, you can use an alternative object called `GDownloadUrl`. You can use `GDownloadUrl` to send and retrieve content the same way you do with `GXmlHttp`; however the API is much simpler. Rather than checking response states and all that other stuff, you just supply a URL with any appropriate `GET` variables and a function to execute when the response in returned. This simplifies the request to the following:

```
GDownloadUrl('storeMarker.php' + getVars,  function(data,responseCode)) {
    //Do something with the data
});
```

But note that this approach doesn't give you as much control over the different states of the request.

Parsing the XML Document Using DOM Methods

When the `readyState` reaches `4` and your `onreadystatechange()` function is triggered, you need to parse the response from the server to determine if the PHP script replied with an execution error or a successful save. Referring back to Listing 3-6, the `storeMarker.php` source, you can see that in the event of a successful save, the `type` attribute of the XML response node is `success`:

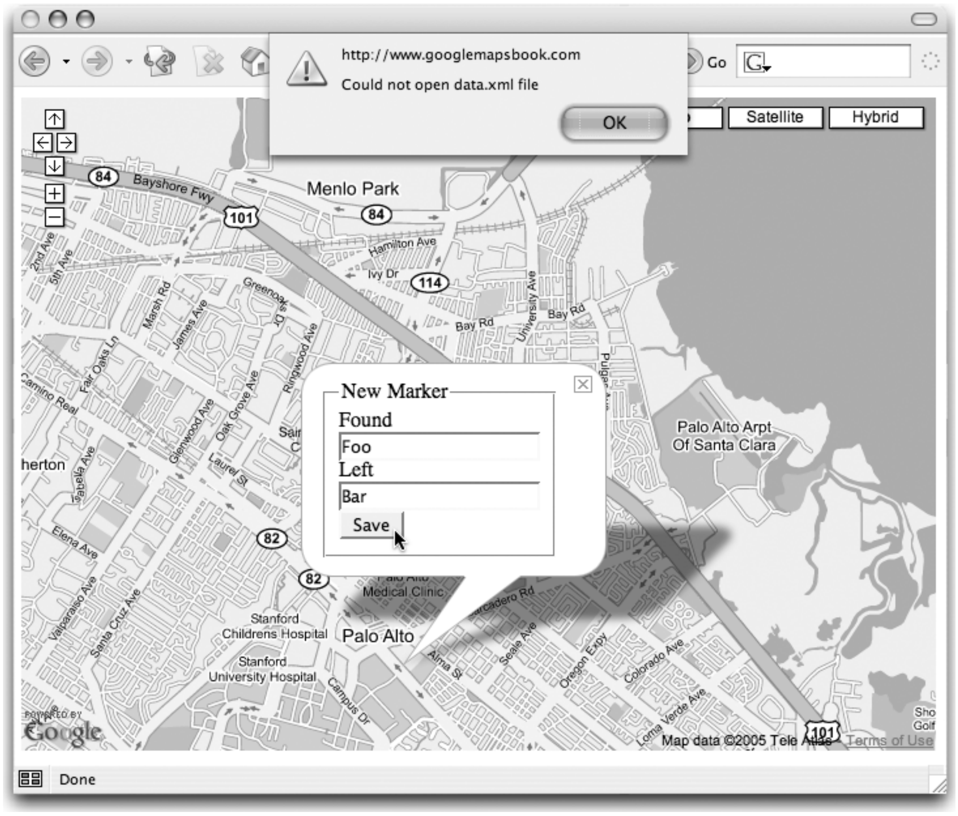
```
<?xml version="1.0"?>
<response type="success">
    <![CDATA[<div><b>Found</b> foo</div><div><b>Left</b> bar </div>]]>
</response>
```

In the event of an error, such as the script not having permission to write to the file, the value of `type` is `error`:

```
<?xml version="1.0"?>
<response type="error">
   <![CDATA[Could not open data.xml file.]]>
</response>
```

When the web browser receives the XML from your `request` object, it is contained in the `responseXML` property. You can now search the XML using the JavaScript DOM methods and properties, such as `xmlDoc.documentElement` to give you the root node (in this case, the `<response>` node) and the `getAttribute()` method to retrieve the value of the `type` attribute of the `<response>` node.

In the event of an error in Listing 3-5, you simply need to call a JavaScript `alert()` with the content of the `<response>` tag to alert the user (Figure 3-13).


http://www.googlemapsbook.com
Could not open data.xml file

New Marker
Found
Foo
Left
Bar

Figure 3-13. *An error in the response*

With a successful execution, you create a new marker at the latitude and longitude of the click and attach an event listener to the marker itself in order to create a new info window with the content of the response. The new marker now indicates the newly created location on the map, and when clicked, displays the information about the marker (Figure 3-14).

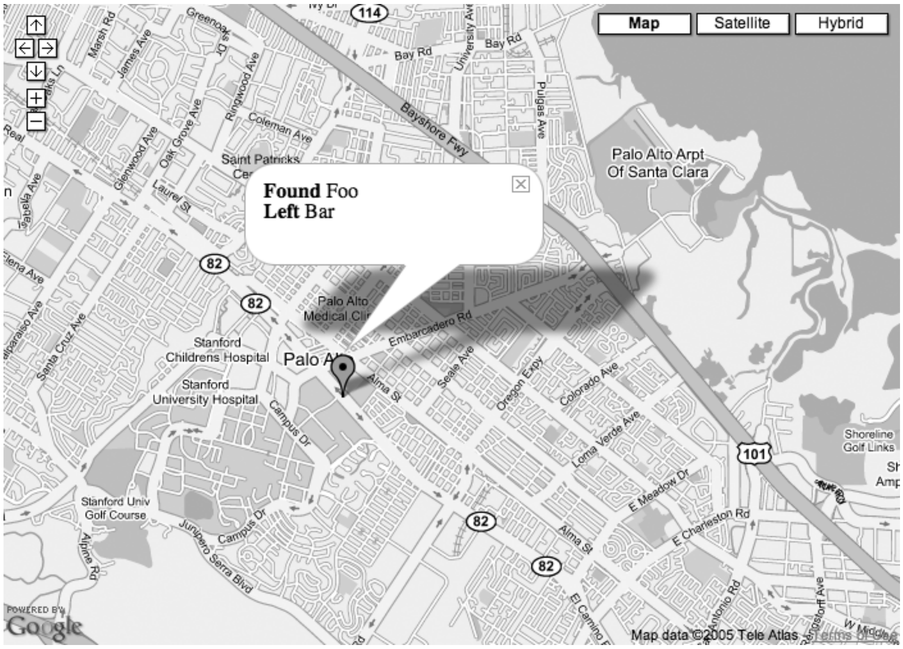


Figure 3-14. *A successful request and response*

You've probably noticed that in Listing 3-5, you've used the `marker.openInfoWindowHTML()` method rather than the `map.openInfoWindow()` method. Since you now have your marker on the map, you can apply the info window directly to it and pass in an HTML string rather than an HTML DOM element.

■ **Caution** When accepting input from the users of your website, it is always good practice to assume the data is evil and the user is trying to take advantage of your system. Always filter input to ensure it's in the format you are expecting. Numbers should be numbers, strings should be strings, and unless desired, nothing should contain HTML or JavaScript code. Listing 3-6 could easily be compromised through cross-site scripting (XSS) if you don't filter out JavaScript in the user-submitted data. For more information see `http://owasp.org`.

Retrieving Markers from the Server

Your geocaching map is almost finished. So far, you've used event listeners to add marks on the map, displayed info windows to ask for more input, and saved the input back to the server using Ajax. You now want to take all the markers you've been collecting and show them on the page when users first visit.

The information to display the markers resides on the server in the `data.xml` file created by the `storeMarkers.php` script in Listing 3-5. In Chapter 2, you loaded the map data from `map_data.php` into the head of the `index.php` document using a `<script>` tag. You could easily do the same thing here, but for this chapter, we're going to mix things up a bit and show you a more controlled way. To gain more interactive control, you'll retrieve the data from the server using the `GXmlHttp` object. Using the `GXmlHttp` object will allow you to retrieve points at any time, not just when the page loads. For example, when you've finished this example, you could extend it further by tracking the movements of the map and retrieve the points based on the map's viewable area, as you'll see later in Chapter 7.

To start, remove the reference to `map_data.php` from the head of your `index.php` file and create the `retrieveMarkers.php` file in Listing 3-7 on your server in the same directory as your HTML document.

Listing 3-7. *retrieveMarkers.php Script Used to Format and Retrieve the data.xml File*

```
<?php
header('Content-Type:text/xml');
$markers = file_get_contents('data.xml');
echo <<<XML
<markers>
$markers
</markers>
XML;
?>
```

Also, copy the `retrieveMarkers()` function from Listing 3-8 into the `map_functions.js` file. In Listing 3-8, notice the marker is created by the same `createMarker()` function you used in Listing 3-5. This allows you to maintain the proper scope of the data passed into the info window. If you create each marker in the `retrieveMarkers()` function, each marker's info window will have the `html` value of the last marker created in the loop. The value of `html` will be identical for each marker because the info window is not actually created until you click the marker and `html` is retrieved from the scope of the JavaScript at that time. By moving the creation into another function, you've given each instance of the function its own namespace.

Listing 3-8. *Ajax retrieveMarkers() Function*

```
function retrieveMarkers() {
    var request = GXmlHttp.create();

    //tell the request where to retrieve data from.
    request.open('GET', 'retrieveMarkers.php', true);
    //tell the request what to do when the state changes.
    request.onreadystatechange = function() {
         if (request.readyState == 4) {
              var xmlDoc = request.responseXML;

              var markers = xmlDoc.documentElement.getElementsByTagName("marker");
              for (var i = 0; i < markers.length; i++) {
                   var lng = markers[i].getAttribute("lng");
                   var lat = markers[i].getAttribute("lat");
                   //check for lng and lat so MSIE does not error
                   //on parseFloat of a null value
                   if(lng && lat) {
                       var latlng = new GLatLng(parseFloat(lat),parseFloat(lng));

                       var html = '<div><b>Found</b> '
                            + markers[i].getAttribute("found")
                            + '</div><div><b>Left</b> '
                            + markers[i].getAttribute("left")
                            + '</div>';

                       var marker = createMarker(latlng, html);
                       map.addOverlay(marker);
                   }
              } //for
         } //if
    } //function

    request.send(null);
}
```

Once you've created the `retrieveMarkers.php` file and copied the `retrieveMarkers()` function into the `map_functions.js` file, you can load the markers into your map by calling the `retrieveMarkers()` function. For example, to load the markers when the page loads, you'll need to call the `retrieveMarkers()` function from the `init()` function after you create the map.

```
function init() {
     ... cut ...
     map = new GMap2(document.getElementById("map"));
     retrieveMarkers();
     ... cut ...
}
```

When the `retrieveMarkers()`function is executed, the server-side PHP script, `retrieveMarkers.php` (Listing 3-7), will return an XML file containing the latitude and longitude for each marker you previously saved.

```
<markers>
    <marker lat="37.441" lng="-122.141" found="Keychain" left="Book"/>
    <marker lat="37.322" lng="-121.213" found="Water Bottle" left="Necklace"/>
    ... etc ...
</markers>
```

The XML also contains the additional information you requested for each marker so that you'll be able to include it in the info window. You can search this file using the JavaScript DOM methods in the same way you did for the `storeMarker()` function in Listing 3-5, but because you have a list of markers, you'll need to loop through the object list from `xmlDoc.documentElement.getElementsByTagName("marker")` and create each marker individually.

You don't necessarily have to return XML to the `GXmlHttp` object. You can also return HTML, text, or the same JSON format introduced in Chapter 2. If you return something other than XML, you need to use the `response.responseText` property and parse it accordingly.

■ **Tip** For more information about using Ajax, read *Beginning Ajax with PHP: From Novice to Professional*, by Lee Babin (`http://www.apress.com/book/bookDisplay.html?bID=10117`).

Adding Some Flair

You now have a fun little interactive web application that anyone with Internet access can use. Geocachers can come and see what kinds of things have been found and let others know what they've found. You could be finished with the map, but let's use the Google Maps API to add just a bit more flair.

All the red markers on the map don't really mean anything when you look at them as a whole. Without clicking on each marker to reveal the info window, there's no way to tell anything about what's there other than the location.

One of the keys to a successful web application is to provide your users the information they want both easily and quickly. For instance, if you come back to the map frequently, you would prefer to quickly pick out the points you haven't seen before, rather than hunt and examine each marker to see the information. To give the map more visual information, let's let the geocachers add a custom icon for their finds. This will make the map more visually interesting and provide quick and easy information to the viewers.

By default, Google uses an inverted teardrop pin for marking points on a map, and up until now, this is what you've been using as well. Now, using Google's `GIcon` object, you can rid your map of the little red dots and customize them to use whatever image you like. Rather than looking at a red marker, you can add a small icon of the find (Figure 3-15).

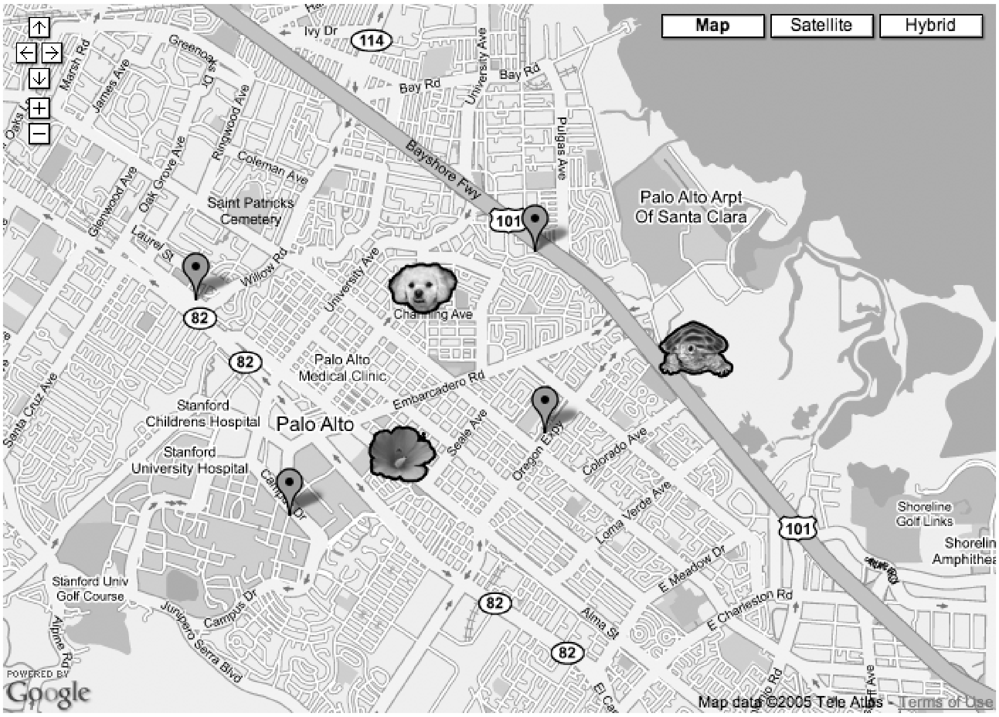

Map
Satellite
Hybrid
Palo Alto Arpt Of Santa Clara
Palo Alto
Stanford Childrens Hospital
Stanford University Hospital
Palo Alto Medical Clinic
Saint Patricks Cemetery
Stanford Univ Golf Course
Shoreline Golf Links
POWERED BY Google
Map data ©2005 Tele Atlas - Terms of Use

Figure 3-15. *Different marker icons on a map*

To use the GIcon object, you are required to set a minimum of three properties:

- GIcon.image: URL of the image
- GIcon.iconSize: Size of the image in pixels
- GIcon.iconAnchor: Location of the anchor point

Also, because you're currently making use of the info window for each of your markers, you must specify the infoWindowAnchor property of the icon.

To get the URL for the GIcon.image property, you'll need to ask the geocaching users where their icon is by adding another element to the info window's form, and then pass it through the GET parameters of your GXmlHttp.request. First, in the click event for the map from Listing 3-4, add the following two highlighted lines:

```
inputForm.innerHTML = '<fieldset style="width:150px;">'
          + '<legend>New Marker</legend>'
          + '<label for="found">Found</label>'
          + '<input type="text" id="found" style="width:100%"/>'
          + '<label for="left">Left</label>'
          + '<input type="text" id="left" style="width:100%"/>'
          + '<label for="left">Icon URL</label>'
          + '<input type="text" id="icon" style="width:100%"/>'
          + '<input type="submit" value="Save"/>'
          + '<input type="hidden" id="longitude" value="' + lng + '"/>'
          + '<input type="hidden" id="latitude" value="' + lat + '"/>'
          + '</fieldset>';
```

Tip For a complete working example of the following changes, see the final example for Chapter 3 in the book's accompanying code or online at `http://googlemapsbook.com/chapter3/final`.

Second, in the `storeMarker()` function from Listing 3-5, add the following highlighted parameter to the request:

```
var getVars =  "?found=" + document.getElementById("found").value
          + "&left=" + document.getElementById("left").value
          + "&icon=" + document.getElementById("icon").value
          + "&lng=" + lng
          + "&lat=" + lat ;
```

Now the icon's URL can be entered and passed to the server. In order to save the information in the `data.xml` file, add the following highlighted lines to the `storeMarkers.php` file in Listing 3-6:

```
$icon = $_GET['icon'];
$marker = <<<MARKER
<marker lat="$lat" lng="$lng" found="$found" left="$left" icon="$icon"/>
MARKER;
```

When the XML is retrieved from the server, it will automatically include the new icon information, so you do not need to modify the `retrieveMarkers.php` file in Listing 3-7. To show the new icons, you'll need to create a new `GIcon` object with the appropriate properties when you retrieve the markers from the server and when you create the new marker upon a successful save.

The `GIcon` objects are created as independent objects and passed in as the second parameter when creating a new `GMarker` object. The `GIcon` objects are reusable, so you do not need to create a new `GIcon` object for each new `GMarker` object, unless you are using a different icon for each marker, as you are doing in this example. To use the icons while retrieving the saved pins in Listing 3-8, add the icon URL as a third parameter to the `createMarker()` call:

```
var marker = createMarker(latlng, html, markers[i].getAttribute("icon"));
```

Then create your `GIcon` object in the `createMarker()` function and assign it to the marker with the following changes:

```
function createMarker(latlng, html, iconImage) {
     if(iconImage!='') {
          var icon = new GIcon();
          icon.image = iconImage;
          icon.iconSize = new GSize(25, 25);
          icon.iconAnchor = new GPoint(14, 25);
          icon.infoWindowAnchor = new GPoint(14, 14);
```

```
        var marker = new GMarker(latlng,icon);
    } else {
         var marker = new GMarker(latlng);
    }
    GEvent.addListener(marker, 'click', function() {
         var markerHTML = html;
         marker.openInfoWindowHtml(markerHTML);
    });
    return marker;
}
```

Additionally, when you create the new `GIcon` object in the `storeMarker()` and `retrieveMarkers()` functions, you'll need to retrieve the icon from the XML and pass the icon image into the `createMarker` call. In `storeMarker()`, add the following:

```
var iconImage = responseNode.getAttribute("icon");
var marker = createMarker(latlng, content, iconImage);
```

In `retrieveMarkers()`, add this:

```
var iconImage =  markers[i].getAttribute("icon");
var marker = createMarker(latlng, html, iconImage);
```

Now when you regenerate the map and create new points, the icon from the URL will be used rather than the default red marker. The size you pick for your `GIcon` objects is based on a width and height in pixels. The preceding changes use an arbitrary `GSize` of 25 by 25 pixels for all of the icons. If the image in the URL is larger than 25 by 25 pixels, it will be squished down to fit.

Summary

Now that you have your first interactive Google Maps web application, grab a GPS and start looking for geocaches to add to your map! Get your friends involved, too, and show off what you've learned.

The ideas and techniques covered in this chapter can be applied to many different web applications, and the same basic interface can be used to mark any geographical information on a map. Want to chart the world's volcanoes? Just click away on the map and mark them down.

You may also want to build on the example here and incorporate some of the other features of the Google Maps API. For example, try retrieving only a specified list of markers, or maybe markers within a certain distance of a selected point. You could also improve the interface by adding listener events to trigger when you open and close an info window, or improve the server-side script by downloading and automatically resizing the desired icons. Later in the book, we'll discuss a variety of other ways to improve your maps.

In the next chapter, we'll show you how you can use publicly available services to automatically plot markers on your map based not just on clicks, but also on postal and street addresses.