CHAPTER 10

Creating Mashups of Several Services

In previous chapters, you learned about the raw ingredients of mashups. This chapter teaches you how to write mashups by walking you through a detailed example of mashing up Flickr photos with Google Maps. This chapter draws upon what you have learned in previous chapters. In Chapter 1, you learned about how geotagging photos started in Flickr and how people such as Rev. Dan Catt and C.K. Yuan built tools—essentially mashups, such as Geobloggers and GMiF—to display those geotagged photos. In Chapter 2, you learned about how such features were baked into Flickr. In Chapter 6, you learned about how to program the Flickr API, while in Chapter 8, you learned the basics of Ajax and how to program Google Maps. We will draw upon all those pieces of knowledge in this chapter.

Given that you can already display Flickr photos on a Yahoo! map, why would you still build any Flickr-map mashup? Well, you might for a number of reasons. You might have a preference for Google Maps over the default maps. Making such a mashup is an instructive process. What better way to learn about mashups than to mash up the two most mashed up services: GMap and Flickr?

What you learn in this chapter will be useful for other mashups. The type of mashup shown here is an extremely common one: getting data from somewhere and putting that data on a map. (Here, we're not screen-scraping that data but rather getting that directly out of an API. There are mashups that require screen-scraping, but that's largely outside the scope of this book.)

You will also learn about the interaction of server-side and client-side programming, another major issue in many mashups. In addition, you will learn about the central process of dealing with impedance matching between APIs. That is, you will find how to make APIs that have different conceptual and implementation details fit together so that data can flow between them. You will learn where to find the common matching points (for example, latitudes and longitudes are common in both the Flickr API and Google Maps) and create interfaces (channel adapters) that bridge the APIs. Finally, there is also the process of taking the work you did and then recasting the same logic into a different environment.

The bulk of this chapter is devoted to writing a simple mashup of Flickr photos with Google Maps using the Google Maps API, but we finish by creating a Flickr/Google Maps mashup using the Google Mapplets API. Since the Mapplets API is similar but not identical to the Google Map API, you will be able to use some of the programming you will do for Google Maps. You'll see how mapplets eliminate the need for server-side programming on your part; the solution we will come up with will be a pure HTML/JavaScript combination.

The goals of this chapter are as follows:

* To enable you to build a significant end-to-end mashup that gives you knowledge about building other mashups
* To cover and reinforce the materials beforehand, which was background material building up to this mashup building

The Design

For both the Google Maps and the Google Mapplets–based mashup, you will want to let your users search for geotagged photos in Flickr and to display them on a Google map. When the user changes the bounding box (that is, the rectangular region of a map often defined by the coordinates of the map's southwest and northeast corners) of the map (by panning and zooming or by changing the zoom level of the map), a new search for geotagged photos is done, and the resulting pictures are displayed on the map.

We will build the mashups in manageable chunks:

* You'll review what you have already learned about geotagging in Flickr and then see how to use the Flickr API to associate locations with photos and how to find geotagged photos.
* You'll study how to access XML web services from the browser using the `XMLHttpRequest` browser object, both natively and wrapped in the Yahoo! UI library.
* You'll study how the security constraints on the browser necessitate a server-side proxy for accessing web services.
* You'll build a server-side proxy to get Flickr geotagged photos.
* You'll work toward building a mashup of the client-side Google Maps API with the Flickr API by first building a simple client-side framework.
* You'll elaborate the client-side framework to translate a search for Flickr geotagged photos into an HTML display of the results.
* You'll transform this framework into a mashup of the Google Maps API and Flickr through a series of steps: setting up a basic map; having the map respond to changes in the viewport of the map; bringing together the Flickr and Google Maps into the same page, first as independent pieces; wiring the bounding box of the Google map to be the source of lat/long coordinates; and finally, making the pictures show up in the map.
* You'll refactor this work into a Flickr/Google mapplet to create a pure client-side solution.
* You'll draw conclusions about what you learned in making these mashups and see how they can be applied to creating other mashups.

Note Chapter 13 provides greater detail on maps and further elaborates on the core examples of this chapter—by mashing up Flickr and Google Earth via KML.

Background: Geotagging in Flickr

As you learned in Chapter 1, geotagging in Flickr started with people using tags (specifically, `geotagged` and `geo:lon`, `geo:lat`) to associate a latitude and longitude with a given photo. This way of geotagging was very popular. Lots of people started creating geotagged photos. Moreover, programs arose to both display geotagged photos (such as GMiF and Geobloggers) and create geotagged photos.

This approach (what I refer here as *old-style geotagging*), as cool as it was, was a hack. Flickr moved to institutionalize geotagging, into what I refer to as *new-style geotagging*. First, Flickr created the framework of *machine tags* to clean up the clutter. Clearly, there was a desire for developers (spurred on by serving users) to add extra metadata to Flickr photos. The result was that data meant for machine consumption was pushed into tags, which were geared more for people manually sticking in descriptions. Flickr decided to take tags of the following form and make them into machine tags:

```
namespace:predicate=value
```

For example, the `geo:lat=` and `geo:lon=` tags have become machine tags. This means they are not displayed by default in the UI. Rather, a user needs to click the "Show machine tags" link to see these machine tags. (The thinking is that machine tags weren't really for human consumption—so why display them?)

Let's consider a geotagged photo that we already looked at in Chapter 1 ("Campanile in the Fog"):

```
http://flickr.com/photos/raymondyee/18389540/
```

You can see the relevant geotags under Tags by clicking "Show machine tags" to reveal this:

```
geo:lon=-122.257704
geo:lat=37.8721
```

You can use the Flickr API to get at these regular and machine tags. Remember that Flickr geotagging was based originally on the `geotagged` tag and tags of the form `geo:lon=[LONGITUDE]` and `geo:lat=[LATITUDE]` that became machine tags. For example, to use the Flickr API to look up the tags for the photo whose ID is 18389540, you issue the following HTTP `GET` request:

```
http://api.flickr.com/services/rest/?method=flickr.tags.getListPhoto~CCC
&api_key={api_key}&photo_id=18389540
```

whose response is as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
  <photo id="18389540">
    <tags>
      <tag id="29475-18389540-11787" author="48600101146@N01"
           authorname="Raymond Yee" raw="campanile"
machine_tag="0">campanile</tag>
      <tag id="29475-18389540-1700" author="48600101146@N01"
           authorname="Raymond Yee" raw="geotagged"
machine_tag="0">geotagged</tag>
      <tag id="29475-18389540-10860922" author="48600101146@N01"
           authorname="Raymond Yee" raw="geo:lon=-122.257704"
```

```
            machine_tag="1">geo:lon=122257704</tag>
        <tag id="29475-18389540-10860930" author="48600101146@N01"
            authorname="Raymond Yee" raw="geo:lat=37.8721"
            machine_tag="1">geo:lat=378721</tag>
        <tag id="29475-18389540-88988" author="48600101146@N01"
            authorname="Raymond Yee" raw="UC Berkeley"
            machine_tag="0">ucberkeley</tag>
        <tag id="29475-18389540-9233381" author="48600101146@N01"
            authorname="Raymond Yee" raw="mashupguide"
            machine_tag="0">mashupguide</tag>
    </tags>
  </photo>
</rsp>
```

Note You might wonder why you get machine tags for latitude and longitude since using `geo:lat` and `geo:lon` has been superceded. I'm showing this technique for historic interest and also because it's still used by older pieces of software (such as the Google Maps in Flickr Greasemonkey script that uses old-style geotagging).

With new-style geotagging, support for geotagging was built into the core of Flickr (geo-information became a first-class citizen of the Flickr data world). Each photo can optionally be associated with a location (that is, a latitude and longitude) and permissions about who can see this location.

There are some major advantages of the new-style geotagging:

* You can search for photos in a given bounding box. There was no way to do so with regular tags unless you crawled a whole bunch of geotagged photos and built your own database of those photos and their locations and built geosearching on top of that database. Flickr does that for you.
* You can control the visibility of the location independently of that photo (that is, the photo can be visible but not the location). In the old-style geotagging, if the photo is visible, then its tags are also visible, thus rendering any `geo:lat`/`geo:lon` visible.
* The new style is the official way to do geotagging, whereas the old style never had official support. Along with it being the official way comes a lot of supporting features: the Flickr map, a link to a map for any georeferenced photo, and so on.

By setting a location, you give a photo a latitude, longitude, and accuracy (1–16): world level equals 1, country equals approximately 3, and street equals approximately 16. The default accuracy is 16. Permissions are the values for four parameters: `is_public`, `is_contact`, `is_friend`, and `is_family` (0 or 1). (See Chapter 2 for a discussion of the permission system in Flickr.) There are five methods under `flickr.photos.geo`: getting, setting, deleting the location of a given photo (`flickr.photos.geo.getLocation`, `flickr.photos.geo.setLocation`, and `flickr.photos.geo.removeLocation`), and getting and setting the permission (`flickr.photos.geo.getPerms` and `flickr.photos.geo.getPerms`).

You'll notice that for the following, in addition to using the old-style geotagging in the example photo, I am also using the new-style geotagging:

```
http://flickr.com/photos/raymondyee/18389540/
```

Since this photo is public, anyone can use `flickr.photos.geo.getLocation` to access the photo's latitude and longitude. (All the other `geo.*` methods require authorization.) Let's use the API to get the location. Issue an HTTP `GET` request on this:

```
http://api.flickr.com/services/rest/?method=flickr.photos.geo.getLocation
&api_key={api_key}&photo_id=18389540
```

You will get the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
  <photo id="18389540">
    <location latitude="37.8721" longitude="-122.257704" accuracy="16">
      <locality>Oakland</locality>
      <county>Alameda</county>
      <region>California</region>
      <country>United States</country>
    </location>
  </photo>
</rsp>
```

For the other methods, it's easier to demonstrate using a Flickr API kit that helps you with the Flickr authentication process (which is covered in detail in Chapter 6). I'll now display some code to show how to use Python to manipulate a photo's location and geopermission. Here, `flickr.client` is an authenticated instance of the Flickr client using Beej's Python Flickr API (`http://flickrapi.sourceforge.net/`).

Let's retrieve the location of the photo:

```
>>> rsp = flickr.client.photos_geo_getLocation(photo_id=18389540)
```

Now let's remove the location of the photo:

```
>>> rsp = flickr.client.photos_geo_removeLocation(photo_id=18389540)
```

Let's write the location back to the photo:

```
>>> rsp = flickr.client.photos_geo_setLocation(photo_id=18389540,lat=37.8721,
lon=-122.257704,accuracy=16)
```

In addition to reading and writing the location and geopermissions of an individual photo, you can use the Flickr API to search for photos that have an associated location. You do so by using the `flickr.photos.search` method (the one to which you were introduced in Chapter 6), documented here:

```
http://www.flickr.com/services/api/flickr.photos.search.html
```

To do a search for geotagged photos, you add the search parameters of the following form:

```
bbox=lon0,lat0,lon1,lat1
```

Here `lon0,lat0` and `lon1,lat1` are the longitude and latitude of the southwest and northeast corners of the bounding box, respectively. Note that you can also use the

accuracy parameter to specify the minimum accuracy level you demand of the specified locations.

Let's consider the example of searching for photos around Berkeley in a bounding box with the following parameters:

```
SW:  37.81778516606761, -122.34374999999999
NE:  37.92619056937629, -122.17208862304686
```

The following will get the first page of all the publicly available geotagged photos in Flickr, including photos of all accuracies (with this call, you can get at the total number of such photos):

```
http://api.flickr.com/services/rest/?api_key={api_key}&method=flickr.photos.sear
ch~CCC
&bbox=-180%2C-90%2C180%2C90&min_upload_date=820483200&accuracy=1
```

You can get the first page of photos with a bounding box around the UC Berkeley campus:

```
http://api.flickr.com/services/rest/?api_key={api_key}&method=flickr.photos.sear
ch~CCC
&bbox=-122.34374999999999%2C+37.81778516606761%2C+-122.17208862304686~CCC
%2C+37.92619056937629&min_upload_date=820483200&accuracy=1
```

The Flickr API doesn't like unqualified searches for geotagged photos. That is, you can't just, say, search for photos in a certain bounding box—you need to use at least one other search parameter to reduce the strain on the Flickr database caused by unqualified searches. Here I'm using the min_upload_date parameter to convince Flickr to give some results.

Background: XMLHttpRequest and Containing Libraries

In the previous chapters, especially Chapters 6 and 7, I concentrated on showing you how to make web service requests using server-side languages such as PHP and Python. In this section, I will show you how to make HTTP requests from JavaScript in the browser. The key piece of technology is the XMLHttpRequest (XHR) object (or XHR-like objects in Internet Explorer). I will outline the basics of XHR, covering briefly how to use XHR in the raw and then in the form of a library (specifically the YUI Connection Manager) that abstracts the details of XHR for you.

Using XMLHttpRequest Directly

The XHR object is an API for JavaScript for transferring XML and other textual data between the (client-side) browser and a server. There are differences in naming the object between Internet Explorer and the other browsers. Moreover, there are subtle issues that are easiest to handle by using a good wrapper around XHR, such as the Yahoo! Connection Manager.

Even though we will be using the Yahoo! Connection Manager to access XHR, it's still useful to look at how to use XHR before relying on a library. Drawing from Peter-Paul Koch's description of XHR at http://www.quirksmode.org/js/xmlhttp.html and

noting that the following proxies an RSS feed of weather in the 94720 ZIP code (see the discussion after the code for an explanation of the script) . . .

```
http://examples.mashupguide.net/ch10/weather.php?p=94720
```

then I present the following, which shows a typical usage of XHR to read the RSS feed:

```
http://examples.mashupguide.net/ch10/xhr.html
```

This extracts and displays an HTML excerpt from the feed:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>xhr.html</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" >
    <script type="text/javascript">
    //<![CDATA[

  // based on http://www.quirksmode.org/js/xmlhttp.html

  var XMLHttpFactories = [
    function () {
        xhr = new XMLHttpRequest(); xhr.overrideMimeType('text/xml'); return
xhr;
    },
    function () {return new ActiveXObject("Msxml2.XMLHTTP")},
    function () {return new ActiveXObject("Msxml3.XMLHTTP")},
    function () {return new ActiveXObject("Microsoft.XMLHTTP")}
  ];

  function getXmlHttpRequest() {
    var xmlhttp = false;
    for (var i=0;i<XMLHttpFactories.length;i++) {
      try {
        xmlhttp = XMLHttpFactories[i]();
      }
      catch (e) {
        continue;
      }
      break;
    }
    return xmlhttp;
  }

    function writeResults() {

      if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        resultsDiv = document.getElementById('results');
        //alert(xmlhttp.responseText);
        var response = xmlhttp.responseXML;
        resultsDiv.innerHTML =

response.getElementsByTagName('description')[1].firstChild.nodeValue;
      }
```

```
    }

    function load() {

      // http://examples.mashupguide.net/ch10/weather.php?p=94720
      xmlhttp = getXmlHttpRequest();
      if (xmlhttp) {
        zip = "94720";
        url = "weather.php?p=" + zip;
        xmlhttp.open('GET', url, true);
        xmlhttp.onreadystatechange = writeResults;
        xmlhttp.send(null);
      }

    }

    //]]>
    </script>
  </head>
  <body onload="load()" >
  <!-- retrieve -->
  <div id="results"></div>
  </body>
</html>
```

Note the following:

* The code attempts to instantiate XHR by trying various ways to do so until it succeeds—or finally fails if none of the methods works.
* Through the use of the following:

```
xmlhttp.onreadystatechange = writeResults;
```

the `writeResults()` method is the callback for the HTTP `GET` request. That is, XHR feeds `writeResults` with its state (`xmlhttp.readyState`). A typical usage pattern is for the callback routine to wait until the call is complete (`xmlhttp.readyState == 4`) and for the return of an HTTP response code of 200 (to indicate a successful call).

* `xmlhttp.responseXML` returns the body of the HTTP response in the form of an XML DOM.

Using the YUI Connection Manager

The main goal of this section is to again use JavaScript to call the Flickr API to get photos from a given bounding box. In the previous section, you learned how to use XHR directly; here, I show you how to use a library that wraps XHR: the Yahoo! UI (YUI) Library's Connection Manager, which is documented here:

```
http://developer.yahoo.com/yui/connection/
```

The official examples page for the Connection Manager is here:

```
http://developer.yahoo.com/yui/examples/connection/index.html
```

Let's look at the weather example provided by the YUI:

```
http://developer.yahoo.com/yui/examples/connection/weather.html
```

Our ultimate goal is to use the Connection Manager to hook up the Flickr API. Instead of jumping directly to that goal, I'll first explain the weather example. The server-side part is relatively easy to understand, thus letting you concentrate on the XHR part of the example. The example is built in with the YUI download, and therefore you can immediately see an example of a client-side JavaScript invocation of the Yahoo! weather web service.

Enter a ZIP code, and hit Get Weather RSS. The web page uses XHR (wrapped by the Connection Manager) to retrieve an RSS 2.0 feed for the ZIP code, parses the weather information, and displays it on the page. Note that this happens without a page reload—remember that is what XHR (and Ajax) can do for you.

One thing to notice about `weather.html` is that its JavaScript code invokes `assets/weather.php` running from the same server. That is, if you have a version of the YUI example loaded on `examples.mashupguide.net`:

```
http://examples.mashupguide.net/lib/yui/examples/connection/weather.html
```

you'll see that it calls the following:

```
http://examples.mashupguide.net/lib/yui/examples/connection/assets/weather.php
```

What does `weather.php` do?

A quick study shows that `weather.php` takes the ZIP code (that is, 94720), does an HTTP `GET` request on the Yahoo! Weather API (`http://developer.yahoo.com/weather/`), and echoes the feed back.

For example, suppose you make the following request:

```
http://examples.mashupguide.net/lib/yui/examples/connection/assets/weather.php?
~CCC
p=94720
```

The script echoes back the following:

```
http://xml.weather.yahoo.com/forecastrss?p=94720
```

This will be something of the following form:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
  <channel>
    <title>Yahoo! Weather - Berkeley, CA</title>
    <link>http://us.rd.yahoo.com/dailynews/rss/weather/Berkeley__CA/~CCC
*http://weather.yahoo.com/forecast/94720_f.html</link>
    <description>Yahoo! Weather for Berkeley, CA</description>
    <language>en-us</language>
    <lastBuildDate>Mon, 05 Nov 2007 12:53 pm PST</lastBuildDate>
    <ttl>60</ttl>
    <yweather:location city="Berkeley" region="CA" country="US"/>
    <yweather:units temperature="F" distance="mi" pressure="in" speed="mph"/>
    <yweather:wind chill="62" direction="300" speed="10"/>
    <yweather:atmosphere humidity="65" visibility="1287" pressure="30.03"
                         rising="2"/>
    <yweather:astronomy sunrise="6:39 am" sunset="5:06 pm"/>
```

```
    <image>
      <title>Yahoo! Weather</title>
      <width>142</width>
      <height>18</height>
      <link>http://weather.yahoo.com/</link>
      <url>http://l.yimg.com/us.yimg.com/i/us/nws/th/main_142b.gif</url>
    </image>
    <item>
      <title>Conditions for Berkeley, CA at 12:53 pm PST</title>
      <geo:lat>37.87</geo:lat>
      <geo:long>-122.3</geo:long>
      <link>http://us.rd.yahoo.com/dailynews/rss/weather/Berkeley__CA/~CCC
*http://weather.yahoo.com/forecast/94720_f.html</link>
      <pubDate>Mon, 05 Nov 2007 12:53 pm PST</pubDate>
      <yweather:condition text="Fair" code="34" temp="62"
                          date="Mon, 05 Nov 2007 12:53 pm PST"/>
      <description><![CDATA[
<img src="http://l.yimg.com/us.yimg.com/i/us/we/52/34.gif" /><br />
 <b>Current Conditions:</b><br />
 Fair, 62 F<BR /><BR />
 <b>Forecast:</b><BR />
  Mon - Sunny. High: 69 Low: 46<br />
  Tue - Partly Cloudy. High: 70 Low: 47<br />
 <br />
<a href="http://us.rd.yahoo.com/dailynews/rss/weather/Berkeley__CA/~CCC
*http://weather.yahoo.com/forecast/94720_f.html">
Full Forecast at Yahoo! Weather</a><BR/>
 (provided by The Weather Channel)<br/>
 ]]></description>
      <yweather:forecast day="Mon" date="05 Nov 2007" low="46" high="69"
                         text="Sunny" code="32"/>
      <yweather:forecast day="Tue" date="06 Nov 2007" low="47" high="70"
                         text="Partly Cloudy" code="30"/>
      <guid isPermaLink="false">94720_2007_11_05_12_53_PST</guid>
    </item>
  </channel>
</rss>
```

Building a Server-Side Proxy

In the previous section, you learned how to use XHR to talk to a local `weather.php` file that in turn calls the Yahoo! Weather API. You might wonder why XHR doesn't go directly to the Yahoo! Weather API. It turns out that because of cross-domain security issues in the browser, you can't use the XHR object to make a request to a server that is different from the originating server of the JavaScript code. That would apply to the Flickr API as it does to the Yahoo! Weather API. To get around this issue, you will need a little help from a server-side proxy in the form of a PHP script whose job it is to take a tag and bounding box as input, call the Flickr API to get photos, and return that in XML or JSON to the calling script.

I'll show you how to write a server-side proxy to the Flickr API to get geotagged photos, but first I'll prove to you that you can't use XHR to go directly to the Yahoo! Weather API.

What Happens with XHR and Direct API Calls?

Let's see why `weather.html` can't just call Yahoo! directly. You can find out what happens by running the following code, which instead of calling the local `weather.php` goes directly to http://xml.weather.yahoo.com/forecastrss?94720:[1]

1. http://examples.mashupguide.net/ch10/direct.connect.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Direct connect</title>
    <script type="text/javascript" src="/lib/yui/build/yahoo/yahoo.js"></script>
    <script type="text/javascript" src="/lib/yui/build/event/event.js"></script>
    <script type="text/javascript"
            src="/lib/yui/build/connection/connection.js"></script>
  </head>
  <body>
  <div id="status"></div>
  <script>
    div = document.getElementById('status');

    var handleSuccess = function(o){

      function parseHeaders(headerStr){

        var headers = headerStr.split("\n");
        for(var i=0; i < headers.length; i++){
          var delimitPos = headers[i].indexOf(':');
          if(delimitPos != -1){
            headers[i] = "<p>" +
            headers[i].substring(0,delimitPos) + ":"+
            headers[i].substring(delimitPos+1) + "</p>";
          }
        return headers;
        }
      }

      if(o.responseText !== undefined){
        div.innerHTML = "Transaction id: " + o.tId;
        div.innerHTML += "HTTP status: " + o.status;
        div.innerHTML += "Status code message: " + o.statusText;
        div.innerHTML += "HTTP headers: " +
parseHeaders(o.getAllResponseHeaders);
        div.innerHTML += "Server response: " + o.responseText;
        div.innerHTML += "Argument object: property foo = " + o.argument.foo +
                 "and property bar = " + o.argument.bar;
      }

   }

    var handleFailure = function(o){
      if(o.responseText !== undefined){
        div.innerHTML = "<li>Transaction id: " + o.tId + "</li>";
```

```
        div.innerHTML += "<li>HTTP status: " + o.status + "</li>";
        div.innerHTML += "<li>Status code message: " + o.statusText + "</li>";
      }
    }

    var callback =
    {
      success:handleSuccess, failure: handleFailure,
      argument: { foo:"foo", bar:"bar" }
    };

    var sUrl = "http://xml.weather.yahoo.com/forecastrss?p=94720";
    var request = YAHOO.util.Connect.asyncRequest('GET', sUrl, callback);
  </script>
  <div id="status"></div>
</body>
</html>
```

If you try to run this, you will get a JavaScript error. In Firefox, if you look in the Firefox error console, you'll see the following:

```
Error: uncaught exception: Permission denied to call method XMLHttpRequest.open
```

The main lesson here is that XHR lets you access URLs only from the same domain—for security reasons. Let's prove that by making a new HTML file in the same directory as a local copy of `weather.php`. This security issue, and the workaround by the server-side proxy, is explained here:

`http://developer.yahoo.com/javascript/howto-proxy.html`

In case you are still skeptical, you can change the JavaScript in your HTML to access `weather.php` from this:

```
var sUrl = "http://xml.weather.yahooapis.com/forecastrss?p=94720";
```

to this:

```
var sUrl = "./weather.php?p=94720";
```

When you load `weather.proxy.html`,[2] you no longer get the error. Instead, you get information about the weather—that means communication is happening between your JavaScript and the Yahoo! weather system. Using Firebug, you can actually see the RSS embedded in the `<div>`—but that's not very nice. Let's now move toward getting Flickr information.

2. http://examples.mashupguide.net/ch10/weather.proxy.html

Building a Server-Side Script for Geolocated Photos

Based on what you just learned, you now know that you need to get results about Flickr geotagged photos from the Flickr API into the browser using XHR. Hence, you'll need a server-side proxy for bridging any client-side script with Flickr. That's the aim of this section.

As an exercise, I recommend you write this code yourself before studying the solution presented. Think about how `weather.php` works and how you can use

`flickr.photos.search` to look for geotagged photos. You can imagine a PHP script that gives access to the full range of input parameters for `flickr.photos.search` in searches of public photos and returns the search results in a variety of useful formats. You can find a list of the input parameters for `flickr.photos.search` here:

`http://www.flickr.com/services/api/flickr.photos.search.html`

A script that I wrote to serve as a server-side proxy for `flickr.photos.search` is `flickrgeo.php`. You can run the script here:

`http://examples.mashupguide.net/ch10/flickrgeo.php`

The code is listed here:

`http://examples.mashupguide.net/ch10/flickrgeo.php.txt`

Moreover, you will find a complete listing of the code in Chapter 13, including a description of how it handles KML and KML network links (which is beyond what is covered here). In this section, I'll describe the overall structure of `flickrgeo.php` and discuss some example usage.

With several exceptions, all the parameters for `flickr.photos.search` are also parameters for `flickrgeo.php`:

* `user_id`
* `tags`
* `tag_mode`
* `text`
* `min_upload_date`
* `max_upload_date`
* `min_taken_date`
* `max_taken_date`
* `license`
* `sort`
* `privacy_filter`
* `accuracy`
* `safe_search`
* `content_type`
* `machine_tags`
* `machine_tag_mode`
* `group_id`
* `place_id`
* `extras`
* `per_page`
* `page`

There are three differences between the parameters for `flickr.photos.search` and for `flickrgeo.php`. First, the `api_key` is hardwired for `flickrgeo.php`. Second, instead of using the single `bbox` parameter from `flickr.photos.search` to specify the bounding box for geotagged photos, `flickrgeo.php` takes four parameters: `lat0`, `lon0`, `lat1`, and `lon1` where `lat0`, `lon0` and `lat1`, `lon1` are, respectively, the southwest and northeast corners of the bounding box. Hence, the value of the `bbox` parameter for `flickr.photos.search` is `{lon0},{lat0},{lon1},{lat1}`.

Second, instead of using the format parameter for Flickr API methods, which takes one of `rest` (the default value), `xml-rpc`, `soap`, `json`, or `php`, `flickrgeo.php` uses an `o_format` parameter to control the output of the script. These are the values recognized by the script:

* `rest` returns the default (rest) output from the Flickr API.
* `json` returns the JSON output from the Flickr API.
* `html` returns an HTML form and list of photos.
* `kml` returns the search results as KML (see Chapter 13 for more details).
* `nl` returns the results as a KML network link (see Chapter 13 for more details).

If the `o_format` is not set or is equal to `html`, then you want to return the HTML form and a display of the photos. If the `o_format` is `rest`, return the default output from the Flickr API (`rest`). If it's `json`, you want to return the JSON output with no callback.

For example, a sample invocation of this script shows the first page of geotagged photos tagged with `cat` from all over the world:

```
http://examples.mashupguide.net/ch10/flickrgeo.php?tags=cat&lat0=-90&lon0=-
180&lat1=~CCC
90&lon1=180&page=1&per_page=10&o_format=html
```

If you change the `o_format` to `json`, you get JSON output:

```
http://examples.mashupguide.net/ch10/flickrgeo.php?tags=cat&lat0=-90&lon0=-
180&lat1=~CCC
90&lon1=180&page=1&per_page=10&o_format=json
```

This script generates a simple user interface so that you can test the input parameters. That is, you can use the `html` interface to see what photos are coming back and then switch the output to `json`, `rest`, `kml`, or `nl` to be used in your server-side proxy. Much of the code is devoted to generating KML and KML network links, functionality used in Chapter 13. There's also some other convenience functionality: automatic form generation, error checking, and some useful default values for the `bbox` parameter. Again, consult Chapter 13 for more details.

Building a Simple Client-Side Frame

You now have `flickrgeo.php`, a server-side proxy for talking to Flickr. Before you turn your attention to directly connecting Google Maps with Flickr, I'll remind you about two basic interactions between the DOM and JavaScript:

* Reading and writing DOM elements, `<div>` elements, and form elements

* Handling simple events to connect form input and displaying calculations

Reading and Writing Elements

In this section, I will remind you how to do some basic things in browser-based JavaScript. Specifically, I'll review how to manipulate certain DOM elements. This section will seem trivial to experienced JavaScript developers, but the example provides a starting point for the rest of the chapter.

To that end of learning some basic JavaScript techniques, create the following HTML file:[3]

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Dom Play</title>
  </head>
  <body>
    <div id="container"></div>
  </body>
</html>
```

Fire up the JavaScript Shell and the Firebug extension to follow what happens when you type the following commands:

```
document
```

```
[object HTMLDocument]
```

```
div = document.getElementById('container')
```

```
[object HTMLDivElement]
```

```
div.innerHTML = 'hello';
```

```
hello
```

Notice that the word *hello* shows up on the web page now. You've just used JavaScript to write to the DOM, specifically *hello* to the `innerHTML` of the `<div>` element with the ID of `container`.

The next step is to write an example with an input box and a submit button. When you hit submit, the `calc_square()` JavaScript function calculates the square of the number and updates the result box (the `answer` span). Start with the following, though we'll leave the `calc_square()` function empty for now:[4]

3. http://examples.mashupguide.net/ch10/dom.html

4. http://examples.mashupguide.net/ch10/square1.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
```

```
  <head>
    <title>Squaring the input(square1.html)</title>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
  </head>
  <body>
    <script type="text/javascript">
    //<![CDATA[
      function calc_square() {
      }
    //]]>
    </script>
    <form action="#" onsubmit="calc_square(); return false;">
      <label>Input a number:</label>
      <input type="text" size="5" name="num" value="0" />
      <input type="submit" value="Square it!" />
    </form>
    <p>The square of the input is: <span id="answer">0</span></p>
  </body>
</html>
```

In the JavaScript Shell, try the following pieces of code:

```
document
```

```
[object HTMLDocument]
```

```
document.forms[0].innerHTML
```

```
<label>Input a number:</label><input size="5" name="num" value="0" type="text">
<input value="Square it!" type="submit">
```

```
document.forms[0].elements[0].value
```

```
0
```

Change the value in the text box, and try it again to see the new value reflected (note `num` is the ID of the `<input>` element):

```
document.forms[0].num.value
```

```
8
```

The following gets you the `<answer>` element:

```
span.document.getElementById('answer')
```

```
[object HTMLSpanElement]
```

Finally, this will fill in 16 to the `<answer>` element:

```
span.document.getElementById('answer').innerHTML = 16
```

```
16
```

Handling Simple Events to Connect Form Input and Display Calculations

Next, you'll want to figure out how to programmatically submit the form (you'll use this logic later). Instead of having to hit the submit button, you will create a method that responds to the button submission event. Remember, in the previous example, it is the job of the `calc_square()` method (which was left empty) to read the input, calculate the square of the input, and write the answer to the `answer` box. Let's fill in `calc_square` with something like this:[5]

```
<script type="text/javascript">
//<![CDATA[
 function calc_square() {
     var n = document.forms[0].num.value;
     document.getElementById('answer').innerHTML = n*n;
  }
//]]>
</script>
<form action="#" onsubmit="calc_square(); return false;">
  <label>Input a number:</label>
  <input type="text" size="5" name="num" value="0" />
  <input type="submit" value="Square it!" />
</form>
<p>The square of the input is: <span id="answer">0</span></p>
<script type="text/javascript">
//<![CDATA[
  document.forms[0].num.onchange = calc_square;  //register an event
//]]>
</script>
```

5. http://examples.mashupguide.net/ch10/square2.html

Hooking the Client-Side Framework to Flickr

Now that you've constructed some simple JavaScript code to read form elements and do a calculation in response to a button submission event, you're ready to wire up a form to use XHR to access the `flickrgeo.php` server-side proxy. That is, you'll let the user fill in new values and do the form submission by JavaScript. Once the user hits Go!, the script returns a URL to use `flickrgeo.php` to search for geotagged photos. We'll build up the example in three steps:

1. Translate the form parameters into a query to `flickrgeo.php`.
2. Use XHR to do the request to `flickrgeo.php` and display the resulting JSON response.
3. Translate that JSON into HTML for display.

Let's start with the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>flickrgeo.1.html</title>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
  </head>
  <body>
    <script type="text/javascript">
    //<![CDATA[
    function get_pictures() {
    /*
    We're aiming for the following:
    flickrgeo.php?tags=flower&lat0=-90&lon0=-180&lat1=90&lon1=180&page=1~CCC
&per_page=10&o_format=json
    */
    }
    //]]>
    </script>
    <form action="#" onsubmit="get_pictures(); return false;">
      <label>Search for photos with the following tag:</label>
      <input type="text" size="20" name="tag" value="flower" />
      <label> located at: lat0,lon0,lat1,lon1:</label>
      <input type="text" size="10" name="lat0" value="-90.0" />
      <input type="text" size="10" name="lon0" value="-180.0" />
      <input type="text" size="10" name="lat1" value="90.0" />
      <input type="text" size="10" name="lon1" value="180.0" />
      <label>at page</label>
      <input type="text" size="4" name="page" value="1" />
      <label>with</label>
      <input type="text" size="3" name="per_page" value="1" />
      <label> per page.</label>
      <button type="submit">Go!</button>
    </form>
    <div id="pics"></div>
  </body>
</html>
```

Writing a URL for Querying flickrgeo.php

Your goal is to figure out how to fill in `get_pictues()` to translate the input parameters from the form into a URL of the correct form. Here's one possible approach:[6]

```
    <script type="text/javascript">
    //<![CDATA[
    function get_pictures() {
    // flickrgeo.php?tags=flower&lat0=-90&lon0=-
180&lat1=90&lon1=180&page=1&per_page
    // =10&o_format=json
      var s = "";
      f = document.forms[0].getElementsByTagName('input'); // get all input
fields
```

```
    for (i = 0; i < f.length; i++)
      if (i < f.length - 1) {
        s = s + f[i].name + "=" + escape(f[i].value) + "&amp;";
      } else {
        s = s + f[i].name + "=" + escape(f[i].value);
      }
    var url = "flickrgeo.php?" + s + "&o_format=json";
    document.getElementById('pics').innerHTML = "<a href=" + url + ">URL</a>";
  }
  //]]>
  </script>
```

The `get_pictures` function iterates through all the `<input>` tags in the form, extracting the name and value of each tag, out of which to create a URL (with parameters) to `flickrgeo.php`. This URL is an HTTP `GET` request for JSON-formatted results for the given parameters.

Using XHR via the YUI Connection Manager to Read the JSON

The next step is to actually grab the JSON that is available at the URL. Using what you learned earlier (in the section "What Happens with XHR and Direct API Calls"), let's use the YUI Connection Manager to call `flickrgeo.php` and display the raw JSON:[7]

6. http://examples.mashupguide.net/ch10/flickrgeo.1.html

7. http://examples.mashupguide.net/ch10/flickrgeo.2.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>flickrgeo.2.html</title>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <script type="text/javascript" src="/lib/yui/build/yahoo/yahoo.js"></script>
    <script type="text/javascript" src="/lib/yui/build/event/event.js"></script>
    <script type="text/javascript"
src="/lib/yui/build/connection/connection.js">
    </script>
  </head>
  <body>
    <script type="text/javascript">
    //<![CDATA[

    var handleSuccess = function(o){

      div = document.getElementById('pics');
      div.innerHTML = "";  // blank out the div

      if(o.responseText !== undefined){

        div.innerHTML += "Server response: " + o.responseText + "<br/>";
      }

   }
```

```
    var handleFailure = function(o){
      if(o.responseText !== undefined){
          alert("failure");
      }
    }

    var callback =
    {
      success:handleSuccess, failure: handleFailure, argument: {}
    };

    function get_pictures() {
    // flickrgeo.php?tags=flower&lat0=-90&lon0=-
180&lat1=90&lon1=180&page=1&per_page
    // =10&o_format=json
      var s = "";
      f = document.forms[0].getElementsByTagName('input'); // get all input
fields
      for (i = 0; i < f.length; i++)
        if (i < f.length - 1) {
          s = s + f[i].name + "=" + escape(f[i].value) + "&";
        } else {
          s = s + f[i].name + "=" + escape(f[i].value);
        }
      var url = "flickrgeo.php?" + s + "&o_format=json";
      var request = YAHOO.util.Connect.asyncRequest('GET', url, callback);

    }
    //]]>
    </script>
    <form action="#" onsubmit="get_pictures(); return false;">
<label>Search for photos with the following tag:</label>
<input type="text" size="20" name="tags" value="flower" />
<label> located at: lat0,lon0,lat1,lon1:</label>
<input type="text" size="10" name="lat0" value="-90.0" />
<input type="text" size="10" name="lon0" value="-180.0" />
<input type="text" size="10" name="lat1" value="90.0" />
<input type="text" size="10" name="lon1" value="180.0" />
<label>at page</label><input type="text" size="4" name="page" value="1" />
<label>with</label>
<input type="text" size="3" name="per_page" value="1" /><label> per
page.</label>
<button type="submit">Go!</button>
    </form>
    <div id="pics"></div>
  </body>
</html>
```

Note what was added:

* `<script>` elements to include the relevant parts of the Yahoo! UI Library to enable the use of the Connection Manager.

* The definition of callback functions (`handleSuccess` and `handleFailure`), which are referenced by the `callback` object, to handle successful and failed calls, respectively, to `flickrgeo.php`. If the call is successful, the JSON output from `flickrgeo.php` is written into the `<div id="pics"></div>`.
* A call to the Yahoo! Connection Manager in the line `var request = YAHOO.util.Connect.asyncRequest('GET', url, callback);`. Remember that an HTTP `GET` request is made to `url` and the HTTP response is fed to the functions contained in the `callback` object.

Converting the JSON to HTML

The next step is to convert the JSON input to HTML so that you can use it to display the photos. Note how you can use `eval()` to convert the JSON coming back from Flickr to a JavaScript object because you trust the source of this JSON. An alternative to `eval()` is JSON `stringify()`.[8]

Here's some code:[9]

8. http://www.json.org/js.html
9. http://examples.mashupguide.net/ch10/flickrgeo.3.html

```
<body>
  <script type="text/javascript">
  //<![CDATA[

  function rspToHTML(rsp) {
    var s = "";
    // http://farm{farm-id}.static.flickr.com/{server-
id}/{id}_{secret}_[mstb].jpg
    // http://www.flickr.com/photos/{user-id}/{photo-id}
    s = "total number is: " + rsp.photos.photo.length + "<br/>";

    for (var i=0; i < rsp.photos.photo.length; i++) {
      photo = rsp.photos.photo[i];
      t_url = "http://farm" + photo.farm + ".static.flickr.com/" +
photo.server +
        "/" + photo.id + "_" + photo.secret + "_" + "t.jpg";
      p_url = "http://www.flickr.com/photos/" + photo.owner + "/" + photo.id;
      s +=  '<a href="' + p_url + '">' + '<img alt="'+ photo.title + '"src="'
+
        t_url + '"/>' + '</a>';
    }
    return s;
  }

  var handleSuccess = function(o){
    div = document.getElementById('pics');
    div.innerHTML = "";  // blank out the div

    if(o.responseText !== undefined){
      div.innerHTML += "Server response: " + o.responseText + "<br/>";

      //let's deposit the response in a global variable
      //so that we can look at it via the shell.
```

```
        window.response = o.responseText;
        window.rsp = eval('(' + o.responseText + ')');
        div.innerHTML = rspToHTML(window.rsp);
      }
  }

   var handleFailure = function(o){
     ...
   }

   var callback =
   {
     ...
   };

   function get_pictures() {

     ...
   }
   //]]>
   </script>
   <form action="#" onsubmit="get_pictures(); return false;">
     ...
   </form>
   <div id="pics"></div>
 </body>
```

You now have a client-side form that uses XHR to query the Flickr API, get back results in JSON, convert the JSON to HTML, and insert that HTML into the page—without a page reload (see Figure 10-1). The next steps are to integrate these results with Google Maps—the work of the next section.

Insert 858Xf1001.tif

Figure 10-1. Results of `flickrgeo.3.html`. Geotagged photos are displayed as HTML in response to the XHR request.

Mashing Up Google Maps API with Flickr

You now have all the pieces needed to finish up the Flickr and Google Maps mashup. Here's a step-by-step walk-through of the big steps:

1. Set up a basic Google map.
2. Have the map respond to changes in the viewport of the map.
3. Bring together Flickr and GMap into the same HTML page by combining the code into one file—the two pieces are just together but don't interact.
4. Wire up the bounding box of the Google map to be the source of the lat/long coordinates.

5. Write the coordinates into the lat0/lon0 and lat1/lon1 boxes.

6. Make the pictures show up in the map.

Setting Up a Basic Google Map

To start with, let's just get a simple Google map set up by using the Google Maps API (which you learned about in Chapter 8):

1. Make sure you have the Google Maps key needed for your domain. The domain I have is `http://examples.mashupguide.net/ch10`. You can calculate the corresponding API key:

```
http://www.google.com/maps/api_signup?url=http%3A%2F%2Fexamples.mashup~CCC
guide.net%2Fch10
```

2. Copy the following code, substituting your key, to get a map centered on UC Berkeley with the size, map-type control, and keyboard handlers (you can use the arrow keys to control the map):[10]

```
<!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>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>Google Maps JavaScript API Example</title>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=~CCC
[API_KEY]"
      type="text/javascript"></script>
    <script type="text/javascript">

    //<![CDATA[

    function load() {
      if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        window.map = map;
        map.setCenter(new GLatLng(37.872035,-122.257844), 13);
        map.addControl(new GSmallMapControl());
        map.addControl(new GMapTypeControl());
      }
    }

    //]]>
    </script>
  </head>
  <body onload="load()" onunload="GUnload()">
    <div id="map" style="width: 800px; height: 600px"></div>
  </body>
</html>
```

10. http://examples.mashupguide.net/ch10/gmap.1.html

Making the Map Respond to Changes in the Viewport of the Map

The next thing to pull off is to have the map respond to changes in the viewport of the map (that is, when the user has panned or zoomed the map). The mechanism to use is Google Maps events:

`http://www.google.com/apis/maps/documentation/#Events_overview`

You can get a list of supported events here:

`http://www.google.com/apis/maps/documentation/reference.html#GMap2`

The relevant event we need here is the `moveend` event, the one that is fired once the viewport of the map has stopped changing (as opposed to the `move` event, which is fired during the changing of the viewport). To see this event in action, load the Google map you just created and use the JavaScript Shell to add a listener for the `moveend` event:

```
onMapMoveEnd = function () {alert("You moved or zoomed the map");}
```

```
function () { alert("You moved or zoomed the map"); }
```

```
GEvent.addListener(map,'moveend', onMapMoveEnd);
```

```
[object Object]
```

With that event listener added, every time you finish panning or zooming the map, an alert box pops up with the message "You moved or zoomed the map."

Let's now write some code that displays the bounding box in a `<div>` element, updating this information every time the map is moved. We are doing this as a stepping-stone to feeding the bounding box information to `flickrgeo.php`. Here's the code:[11]

11. http://examples.mashupguide.net/ch10/gmap.2.html

```
<!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>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>gmap.2.html</title>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=[API_KEY]"
      type="text/javascript"></script>
    <script type="text/javascript">

    //<![CDATA[

    function updateStatus() {
      var div = document.getElementById('mapinfo');
      div.innerHTML = map.getBounds();
      return (1);
    }

    function onMapMove() {

      updateStatus();
```

```
    }

    function onMapZoom(oldZoom, newZoom) {

      updateStatus();
    }

    function load() {
      if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        window.map = map;
        map.setCenter(new GLatLng(37.872035,-122.257844), 13);
        map.addControl(new GSmallMapControl());
        map.addControl(new GMapTypeControl());
        window.kh = new GKeyboardHandler(map);

        GEvent.addListener(map,'moveend',onMapMove);
        GEvent.addListener(map,'zoomend',onMapZoom);
        updateStatus();
      }
    }

    //]]>
    </script>
  </head>

  <body onload="load()" onunload="GUnload()">
    <div id="map" style="width: 800px; height: 600px"></div>
    <div id="mapinfo"></div>
  </body>
</html>
```

Bringing Together the Flickr and GMap Code

At this point, you are now ready to bring together the Flickr elements (the input form hooked up to `flickrgeo.php`) and the Google map. The first thing to do is to display the two parts on the same page without having them interact. Getting things displaying side by side ensures that you have the proper dependencies worked out. Once you get there, then you can wire the two pieces together. The first thing to do is to copy and paste code from your Flickr code and GMap code into one file. Here is one possible way to do it:

`http://examples.mashupguide.net/ch10/gmapflickr1.html`

Wiring Up the Bounding Box of the Google Map

Let's get some interaction going between the Flickr parts and the Google map, now that they are contained in the same HTML page. Let's wire up the bounding box of the Google map to be the source of the lat/long coordinates. Now, when you move or zoom the Google map, the new coordinates are written into the form elements (the lat0/lon0 and lat1/lon1 boxes) for the Flickr search.[12]

12. http://examples.mashupguide.net/ch10/gmapflickr2.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>gmapflickr.2.html</title>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=[API_KEY]"
      type="text/javascript"></script>
    <script type="text/javascript">

    //<![CDATA[

    function updateStatus() {
      var div = document.getElementById('mapinfo');
      div.innerHTML = map.getBounds();

      document.forms[0].lat0.value = map.getBounds().getSouthWest().lat();
      document.forms[0].lon0.value = map.getBounds().getSouthWest().lng();
      document.forms[0].lat1.value = map.getBounds().getNorthEast().lat();
      document.forms[0].lon1.value = map.getBounds().getNorthEast().lng();

      get_pictures();
    }

    function onMapMove() {
      updateStatus();
    }

    function onMapZoom(oldZoom, newZoom) {
      updateStatus();
    }

    function load() {

      ...
    }

    //]]>
    </script>
    <script type="text/javascript" src="/lib/yui/build/yahoo/yahoo.js"></script>
    <script type="text/javascript" src="/lib/yui/build/event/event.js"></script>
    <script type="text/javascript"
            src="/lib/yui/build/connection/connection.js"></script>
    <script type="text/javascript">
    //<![CDATA[
    function rspToHTML(rsp) {
      var s = "";
      // http://farm{farm-id}.static.flickr.com/{server-
id}/{id}_{secret}_[mstb].jpg
      // http://www.flickr.com/photos/{user-id}/{photo-id}
      s = "total number available is: " + rsp.photos.total + "<br/>";

      for (var i=0; i < rsp.photos.photo.length; i++) {
        photo = rsp.photos.photo[i];
```

```
        t_url = "http://farm" + photo.farm + ".static.flickr.com/" +
photo.server +
          "/" + photo.id + "_" + photo.secret + "_" + "t.jpg";
        p_url = "http://www.flickr.com/photos/" + photo.owner + "/" + photo.id;
        s +=  '<a href="' + p_url + '">' + '<img alt="'+ photo.title + '"src="'
+
          t_url + '"/>' + '</a>';
      }
      return s;
    }

    var handleSuccess = function(o){

      ...
    }
   }

    var handleFailure = function(o){

      ...
    }

    var callback =
    {
      ...
    };

    function get_pictures() {

      ...
    }
    //]]>
    </script>
  </head>

  <body onload="load()" onunload="GUnload()">
    <form action="#" onsubmit="get_pictures(); return false;">
      <label>Search for photos with the following tag:</label>
      <input type="text" size="20" name="tags" value="flower" />
      <label> located at: lat0,lon0,lat1,lon1:</label>
      <input type="text" size="10" name="lat0" value="-90.0" />
      <input type="text" size="10" name="lon0" value="-180.0" />
      <input type="text" size="10" name="lat1" value="90.0" />
      <input type="text" size="10" name="lon1" value="180.0" />
      <label>at page</label><input type="text" size="4" name="page" value="1" />
      <label>with</label>
      <input type="text" size="3" name="per_page" value="1" />
      <label> per page.</label>
      <button type="submit">Go!</button>
    </form>
  <div id="pics"></div>
  <div id="map" style="width: 800px; height: 600px"></div>
  <div id="mapinfo"></div>
  </body>
```

```
</html>
```

Note that as soon as the page is loaded, the `load` function is called, which in turn calls `updateStatus`. The result is a search for photos using the starting parameters in the form. That is, geotagged photos tagged with `flower` are retrieved and displayed. You can change the starting photos by changing the default value for the `<input>` element to `tags`.

Making the Pictures Show Up in the Map

In this section, you'll complete the wiring between the Flickr results and the map. I'll show you how to display the images in the list on the map. This is done by creating markers for each of the photos and adding those markers as overlays to the map. That involves generating HTML to put into the markers.

I'll remind you how to add overlays to a Google map using the API:

```
point = new GLatLng (37.87309185260284, -122.25508689880371);
marker = new GMarker(point);
map.addOverlay(marker);
```

Here's the code with the new stuff in bold (see Figure 10-2):[13]

13. http://examples.mashupguide.net/ch10/gmapflickr.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>gmapflickr.html</title>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=[API_KEY]"
      type="text/javascript"></script>
    <script type="text/javascript">

    //<![CDATA[
    // set up a blank object to hold markers that are added to the map
    markersInMap = {}

    function updateStatus() {
      var div = document.getElementById('mapinfo');
      div.innerHTML = map.getBounds();

      document.forms[0].lat0.value = map.getBounds().getSouthWest().lat();
      document.forms[0].lon0.value = map.getBounds().getSouthWest().lng();
      document.forms[0].lat1.value = map.getBounds().getNorthEast().lat();
      document.forms[0].lon1.value = map.getBounds().getNorthEast().lng();

      get_pictures();
    }

    // Creates a marker at the given point with the given msg.
    function createMarker(point, msg) {
      var marker = new GMarker(point);
      GEvent.addListener(marker, "click", function() {
        marker.openInfoWindowHtml(msg);
      });
```

```
      return marker;
    }

    function photos_to_markers(rsp) {

      // loop through the photos
      for (var i=0; i < rsp.photos.photo.length; i++) {
        var photo = rsp.photos.photo[i];
        // check whether marker already exists
        if (!(photo.id in markersInMap)) {
          var point = new GLatLng (photo.latitude, photo.longitude);
          var msg = photo.title + "<br>" + genPhotoLink(photo);
          map.addOverlay(createMarker(point, msg));
          markersInMap[photo.id] = "";  // don't know what to store so far.
        }
      }
    }

    function onMapMove() {
      updateStatus();
    }

    function onMapZoom(oldZoom, newZoom) {
      updateStatus();
    }

    function load() {

      ...
    }

    //]]>
    </script>
    <script type="text/javascript" src="/lib/yui/build/yahoo/yahoo.js"></script>
    <script type="text/javascript" src="/lib/yui/build/event/event.js"></script>
    <script type="text/javascript"
            src="/lib/yui/build/connection/connection.js"></script>
    <script type="text/javascript">
    //<![CDATA[

    function genPhotoLink(photo) {
        var t_url = "http://farm" + photo.farm + ".static.flickr.com/" +
          photo.server + "/" + photo.id + "_" + photo.secret + "_" + "t.jpg";
        var p_url = "http://www.flickr.com/photos/" + photo.owner + "/" +
photo.id;

        return '<a href="' + p_url + '">' + '<img alt="'+ photo.title + '"src="'
+
          t_url + '"/>' + '</a>';
    }

    function rspToHTML(rsp) {

      ...
```

```
    }

    var handleSuccess = function(o){
      div = document.getElementById('pics');
      div.innerHTML = "";  // blank out the div

      if(o.responseText !== undefined){
        //let's deposit the response in a global variable
        //so that we can look at it via the shell.
        window.response = o.responseText;
        window.rsp = eval('(' + o.responseText + ')');
        div.innerHTML = rspToHTML(window.rsp);
        photos_to_markers(window.rsp);
      }
    }

    var handleFailure = function(o){

      ...
    }

    var callback =
    {
      ...
    };

    function get_pictures() {

      ...
    }
    //]]>
    </script>
  </head>

  <body onload="load()" onunload="GUnload()">
    <form action="#" onsubmit="get_pictures(); return false;">
      <label>Search for photos with the following tag:</label>
      <input type="text" size="20" name="tags" value="flower" />
      <label> located at: lat0,lon0,lat1,lon1:</label>
      <input type="text" size="10" name="lat0" value="-90.0" />
      <input type="text" size="10" name="lon0" value="-180.0" />
      <input type="text" size="10" name="lat1" value="90.0" />
      <input type="text" size="10" name="lon1" value="180.0" />
      <label>at page</label><input type="text" size="4" name="page" value="1" />
      <label>with</label>
      <input type="text" size="3" name="per_page" value="1" />
      <label> per page.</label>
      <button type="submit">Go!</button>
    </form>
  <div id="pics"></div>
  <div id="map" style="width: 800px; height: 600px"></div>
  <div id="mapinfo"></div>
  </body>
</html>
```

Insert 858Xf1002.tif

Figure 10-2. The Flickr Google Maps mashup

This is just a beginning of a mashup between Flickr geotagged photos and Google Maps. Some ideas for elaborating this mashup include the following:

* Refining the look and feel of the mashup (including removing `<div id="mapinfo">`, which currently displays the bounding box of the map)
* Dealing with the fact that clicking a marker and its consequent window opening moves the map
* Clustering photos that are at the same location (as is done in the Flickr map interface)

Google Mapplet That Shows Flickr Photos

In addition to the Google Maps API, which allows a developer to embed Google Maps on a third-party site, Google recently introduced Google Mapplets as a way of adding extensions to Google Maps directly as little applications that run in a side panel. (Any mapplet you install and turn on interacts with the same map. For example, if you are using a mapplet for displaying flower shops and another one that displays restaurants, the resulting Google map shows both flower shops and restaurants.) You can find developer information here:

`http://www.google.com/apis/maps/documentation/mapplets/`

In this section, I'll show you how to create a basic mapplet to display Flickr geotagged photos. Mapplets are a combination of JavaScript and HTML, embedded in an XML file. The methods you use are similar but not identical to those found in the Google Maps API, and there's no need to write any server-side components. The Mapplets API provides wrappers for XHR that talk to the Google servers (which in turn act like server-side proxies that we wrote in PHP).

You can find the source for a mapplet that allows users to search for Flickr pictures of a certain tag here:

`http://examples.mashupguide.net/ch10/flickr.mapplet.xml`

Add the mapplet to your collection of maps. (See "Adding a Google Mapplet to Your Google My Maps.")

ADDING A GOOGLE MAPPLET TO YOUR GOOGLE MY MAPS

1. Go to `http://maps.google.com/`, and log in to Google Maps if you are not already logged in.
2. Click the My Maps tab.
3. Click Browse the Directory button or link.
4. Click the Add by URL link to the right of the Search Google Maps Content button.

5. Enter the URL of the mapplet source (for example, `http://examples.mashupguide.net/ch10/flickr.mapplet.xml`), and hit the Add button.

6. Click Back to Google Maps.

7. Now you should now see on the My Maps tab under Created by Others a map called "Flickr Geotagged Photos." You can use the check box to turn it off and on.

Figure 10-3 shows the mapplet in action.

Insert 858Xf1003.tif

Figure 10-3. The Flickr Google Maps mapplet mashup

The source is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Module>
<ModulePrefs title="Flickr Geotagged Photos"
             description="Show Flickr photos"
             author="Raymond Yee"
             author_email="raymondyee@mashupguide.net"
             height="150">
  <Require feature="sharedmap"/>
</ModulePrefs>
<Content type="html"><![CDATA[

<script>

  var map = new GMap2();
  var border = null;

function genPhotoLink(photo) {
    var t_url = "http://farm" + photo.farm + ".static.flickr.com/" +
photo.server +
      "/" + photo.id + "_" + photo.secret + "_" + "t.jpg";
    var p_url = "http://www.flickr.com/photos/" + photo.owner + "/" + photo.id;

    return '<a href="' + p_url + '">' + '<img alt="'+ photo.title + '"src="' +
      t_url + '"/>' + '</a>';
}

// Creates a marker at the given point with the given msg.
function createMarker(point, msg) {
  var marker = new GMarker(point);
  GEvent.addListener(marker, "click", function() {
    marker.openInfoWindowHtml(msg);
  });
  return marker;
}

function createMarkerAndDiv (point,msg) {
    var marker, e, anchors, alink
```

```
    marker = createMarker(point, msg);
    e = document.createElement("div");

    e.innerHTML = msg +  "<a href='#'>Show</a><br>"
    anchors = e.getElementsByTagName('a')
    alink = anchors[anchors.length-1];
    alink.onclick = function(){marker.openInfoWindowHtml(msg);}

    return [marker,e];
}

function cb(s) {
    var rsp = eval('(' + s + ')');
    var marker, e

    // clear the photos
    map.clearOverlays();

    // add border
    map.addOverlay(border);

    var pdiv = document.getElementById("pictures");
    pdiv.innerHTML = "Total number available is: " + rsp.photos.total +
"<br/>";;

    // put the pictures on the map
    for (var i=0; i < rsp.photos.photo.length; i++) {
        var photo = rsp.photos.photo[i];

        var point = new GLatLng (photo.latitude, photo.longitude);
        var msg = photo.title + "<br>" + genPhotoLink(photo);

        md =  createMarkerAndDiv(point,msg);
        marker = md[0];
        e=md[1];

        map.addOverlay(marker);
        pdiv.appendChild(e);
     }
}

function get_pictures() {
    var API_KEY = "[API_KEY]";
    fForm = document.getElementById('FlickrForm');

    map.getBoundsAsync(function(bounds) {
      var lat0 = bounds.getSouthWest().lat();
      var lon0 = bounds.getSouthWest().lng();
      var lat1 = bounds.getNorthEast().lat();
      var lon1 = bounds.getNorthEast().lng();

      // add polyline to mark the search boundaries
      border  = new GPolygon([
```

```
        new GLatLng(lat0, lon0),
        new GLatLng(lat1, lon0),
        new GLatLng(lat1, lon1),
        new GLatLng(lat0,lon1),
        new GLatLng(lat0,lon0)
      ], "#ff0000", 2);

      var url =
"http://api.flickr.com/services/rest/?method=flickr.photos.search" +
        "&api_key=" + API_KEY +
        "&bbox=" + lon0 + "%2C" + lat0 + "%2C" + lon1 + "%2C" + lat1  +
        "&per_page=" + fForm.per_page.value +
        "&page=" + fForm.page.value +
        "&format=json&nojsoncallback=1&extras=geo";

      var tagValue = fForm.tag.value;
      // search by tag only if the box is not blank.
      if (tagValue.length) {
        url = url + "&tags=" + fForm.tag.value;
      } else {
        url = url + "&min_upload_date=82048320O";
      }

      _IG_FetchContent(url, cb);

    } //anonymous function
    ); //map.getBoundsAsync

} //get_pictures

</script>

<form action="#" onsubmit="get_pictures(); return false;" id="FlickrForm">
  <p>Search for photos with the following tag:
  <input type="text" size="20" name="tag" value="flower">
  at page <input type="text" size="4" name="page" value="1"> with
  <input type="text" size="3" name="per_page" value="10"> per page.
  <button type="submit">Go!</button></p>
</form>
<div id="pictures"></div>

]]></Content>
</Module>
```

A few words about the logic of this code:

* This code is compact partly because the `_IG_FetchContent()` method makes accessing the Flickr API fairly straightforward because you can code the URL directly to the Flickr API instead of having to create your own server-side proxy (such as `flickrgeo.php`).
* Mapplets do not provide much room to display content in the sidebar. Hence, the mapplet can be better optimized to make use of the small space.

Summary

In this chapter, you learned how to create a mashup of two different APIs, the Flickr API and the Google Maps API, to display geotagged Flickr photos on a Google map. After reviewing geotagging in Flickr, you learned how to access XML web services using the `XMLHttpRequest` browser object (XHR) and deal with security constraints in the browser by creating server-side proxies to access web services. You then looked at how to use `flickrgeo.php`, a server-side proxy to search photos in Flickr. You then set up a simple client-side framework that we transformed one step at a time into a mashup between Flickr and Google Maps. Finally, you refactored that work into a Flickr/Google mapplet to create a pure client-side solution.

Although this chapter focused on Flickr and Google Maps, what you learned in this chapter can be generalized for other mashups. For instance, you'll continue to see repeated interactions between server-side and client-side components. Building mashups in controlled steps, adding functionality one piece at a time, is a good way to work. Frameworks such as Google Mapplets let you write widgets in HTML and JavaScript by providing server-side proxies to access web services from other parties (such as the `_IG_FetchContent()` method in Google Mapplets).

When creating mashups, you are often faced with issues of "impedance matching"—that is, how to translate information from one source into a form that is usable by the consumer of that information. In this chapter, we focused on extracting geocoding information from Flickr and then translating it for use by Google Maps. Data flow went the other way too: how to get the viewport of the Google map to define the bounding box for a query for geotagged photos in Flickr. You will see the need to deal with impedance matching throughout the rest of the book.