

THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Covers
API Version 2, including
Google's geocoder!

Beginning Google Maps Applications with PHP and Ajax

From Novice to Professional

Build awesome web-based mapping applications with this powerful API!

Michael Purvis, Jeffrey Sambells,
and Cameron Turner

*Foreword by Mike Pegg,
Founder of the Google Maps Mania Blog*

Apress®



Geocoding Addresses

As you've probably already guessed, the heart of any mashup is correlating your information with latitudes and longitudes for plotting on your map. Fortunately, geocoding services are available to help you convert postal addresses to precise latitude and longitude coordinates. For locations in the United States and Canada, these services make geocoding addresses relatively easy and quite accurate most of the time. In other parts of the world, the job can become much harder.

In this chapter, while building a store locator map, you'll learn how to do the following:

- Create an XML file describing a set of locations and details.
- Request information from geocoding web services and process their responses.
- Learn the pros and cons of Google's new JavaScript-based geocoder, as well as suggestions on when to use it.
- Precompute and cache the latitude and longitude for the points you intend to plot.

Creating an XML File with the Address Data

In this chapter, you're going to create a simple store location map using the postal address of each location in the chain to map the markers. The important aspect about this kind of data is that it changes slowly over time. A few points are added every now and then as the chain of stores expands, but rarely are points removed. In general, it makes sense to precompute and cache information like latitude and longitude for this type of data, as you'll see in the "Caching Lookups" section later in this chapter.

For this example, we'll use the chain of stores and attractions known as Ron Jon Surf Shop, since its story appeals to our own entrepreneurial style:

It was 1959 and on the New Jersey shore a bright young man named Ron DiMenna was just discovering the sport of surfing with fiberglass surfboards. The pastime soon became a passion and homemade surfboards would no longer do. When his father heard that Ron wanted his own custom surfboard from California, he suggested, "Buy three, sell two at a profit, then yours will be free." His Dad was right and Ron Jon Surf Shop was born.

<http://www.ronjons.com>

With permission, we've taken the addresses of all of the Ron Jon properties from the website and converted them into the sample XML data file for this chapter. Listing 4-1 shows the `ronjons.xml` file that you'll use while following the examples in this chapter. By the end of the chapter, you'll be able to create your own XML file and use the same techniques to map your own list of related addresses.

Listing 4-1. *Ron Jon Properties (from www.ronjons.com as of July 2006)*

```
<?xml version="1.0" encoding="UTF-8"?>
<stores>
  <store>
    <name>"The Original" Ron Jon Surf Shop</name>
    <address>901 Central Avenue</address>
    <city>Long Beach Island</city>
    <state>NJ</state>
    <zip>08008</zip>
    <phone>(609) 494-8844</phone>
    <pin>store</pin>
  </store>
  <store>
    <name>"One of a Kind" Ron Jon Surf Shop</name>
    <address>4151 North Atlantic Avenue</address>
    <city>Cocoa Beach</city>
    <state>FL</state>
    <zip>32931</zip>
    <phone>(321) 799-8888</phone>
    <pin>store</pin>
  </store>
  <store>
    <name>Ron Jon Surf Shop - Sunrise </name>
    <address>2610 Sawgrass Mills Circle</address>
    <address2>Suite 1415</address2>
    <city>Sunrise</city>
    <state>FL</state>
    <zip>33323</zip>
    <phone>(954) 846-1880</phone>
    <pin>store</pin>
  </store>
  <store>
    <name>Ron Jon Surf Shop - Orlando</name>
    <address>5160 International Drive</address>
    <city>Orlando</city>
    <state>FL</state>
    <zip>32819</zip>
    <phone>(407) 481-2555</phone>
    <pin>store</pin>
  </store>
</store>
```

```
<name>Ron Jon Surf Shop - Key West</name>
<address>503 Front Street</address>
<city>Key West</city>
<state>FL</state>
<zip>33040</zip>
<phone>(305) 293-8880</phone>
<pin>store</pin>
</store>
<store>
  <name>Ron Jon Surf Shop - California</name>
  <address>20 City Blvd.</address>
  <address2>West Building C Suite 1</address2>
  <city>Orange</city>
  <state>CA</state>
  <zip>92868</zip>
  <phone>(714) 939-9822</phone>
  <pin>store</pin>
</store>
<store>
  <name>Ron Jon Cape Caribe Resort</name>
  <address>1000 Shorewood Drive</address>
  <city>Cape Canaveral</city>
  <state>FL</state>
  <zip>32920</zip>
  <phone>(321) 328-2830</phone>
  <pin>resort</pin>
</store>
</stores>
```

Caution We've left out declaring a namespace for this XML document to keep the example simple for XML novices. For these simple examples, a namespace is not needed. However, using namespaces is generally a good idea. For more information, check out the excellent primer on namespaces, "XML Namespaces by Example," at <http://www.xml.com/pub/a/1999/01/namespaces.html>.

Using Geocoding Web Services

Converting postal addresses to precise latitude and longitude coordinates is made simple by a few good geocoding services. In this section, we're going to cover some of the most popular geocoding services we've found to date. (For an updated list of the geocoders we know about, check out our website at <http://googlemapsbook.com/geocoders>.)

However, before you dive into the available web services, there are a few server-side requirements you'll need to consider.

Note There are also sources of raw information that you can use to make your own geocoding solutions. So, if you can't find a service that fits your needs, and you have a place to get some raw street data, see Chapter 11 for the basics of creating your own geocoding service.

Requirements for Consuming Geocoding Services

To consume the services, you need a web server permanently connected to the Internet, and it will need to be able to connect to the appropriate services. For the examples in this chapter, you'll be using the PHP CURL extension to retrieve the XML information from the available services, and you'll be using PHP 5's SimpleXML feature to parse the XML you retrieve.

CURL

Many of these services require you to send a carefully crafted URL request to retrieve your information. For this purpose, you'll use the CURL extension in PHP. This extension is not bundled by default with PHP; however, it is one of the most commonly installed extensions, so you should have no trouble finding a host with it available.

Basically, the PHP CURL functions are available through the use of `libcurl`, a library created by Daniel Stenberg, and allow you to connect and communicate with web servers using many different types of protocols. You'll be using a very small subset of functions here, though we encourage you to look deeper into this very useful feature by visiting <http://www.php.net/curl>.

SimpleXML

Most of the geocoding solutions we're about to investigate return an XML document as their result. To process these responses, you'll use PHP 5's SimpleXML features, which are perfectly suited to the level of complexity of the answers you'll receive. SimpleXML brings a unique perspective to XML parsing in that element names are automatically (recursively) converted into properties of an object, and attributes are accessed as if they were items in a named array. From your point of view, all of this happens when the `simplexml_load_string($string)` constructor is called; however, from a memory usage point of view, it happens on demand.

If you've never used SimpleXML, or need a refresher, we encourage you to check out a great article by Zend Technologies available at <http://www.zend.com/php5/articles/php5-simplexml.php>. This article also presents an example for PHP 4's DOM processing, in case you don't have access to PHP 5 on your server (you should really consider upgrading!).

Note If you have PHP 4 and still want something like SimpleXML you might want to try MiniXML from <http://minixml.psychogenic.com>. It gets rave reviews on many forums and news groups, though we have never needed to use it ourselves. The description from their site states that: "MiniXML provides a simple, API to generating and parsing XML. Its advantages are ease-of-use and the fact that no additional libraries are required. It comes with two independent implementations, 100% PHP and 100% PERL, which you can use separately."

The Google Maps API Geocoder

We'll begin our investigation of geocoding solutions with the Google Maps API geocoder (http://www.google.com/apis/maps/documentation/#Geocoding_Examples). Google claims that this solution should give street-level accuracy for the United States, Canada, France, Italy, Germany, and Spain. The Google developers hope to roll out support for more countries in the near future, so before you rule them out for a particular country, you might want to check either our website (<http://googlemapsbook.com>) or the official API documentation.

Before June 2006, there was no official geocoder from Google. Many hacks used the `maps.google.com` site's built-in geocoder and screen-scraped the answer. This was an explicitly unauthorized use of the service, and while we never heard of a crackdown on people doing this, Google did frown upon it. As a result, a number of alternative services popped up to fill the void, which we'll cover later in the chapter. Despite being late to the game, Google's geocoder has a number of really interesting features that none of the others have yet, and we'll highlight them throughout the discussion.

First, we'll look at the most basic method for accessing the geocoder: the HTTP-based lookup methods. You can also access the geocoder within JavaScript, as discussed later in this section and in Chapter 10's polyline example.

Like most of the other services we'll investigate, the Google method uses Representational State Transfer (REST) requests for accessing the service. REST is basically a simple HTTP request that passes GET parameters by appending things like `key=value&key2=value2` to the end of the request URL. Generally, a REST service returns some form of text-based data structure like XML. Google's geocoder is (so far) unique in that it can also return Keyhole Markup Language, or KML (for use in Google Earth), and JSON directly.

THE ORIGIN OF REST

Representational State Transfer (REST) is a concept used to connect services in distributed systems like the World Wide Web. The term originated in a 2000 doctoral dissertation about the Web written by Roy Fielding, one of the principal authors of the HTTP specification, and has quickly passed into widespread use in the networking community.

Fielding's vision of REST described a strict abstraction of architectural principles. However, people now often loosely use the term to describe any simple web-based interface that uses XML and HTTP without the extra abstraction layers of approaches like the SOAP protocol. As a result, these two different uses of REST cause some confusion in technical discussions. Throughout this book, we refer to it in the looser, more common, meaning of REST.

Google has outdone many of the other geocoders on the market in that its geocoder returns an excellent answer given fairly poor input. It does not require you to separate out the street number, street name, direction (N, S, E, W, and so on), city, state, or even ZIP code. It simply takes what you give it, uses Google's extensive experience with understanding your search terms, and returns a best guess. Moreover, the service formats the input you give it into a nice, clean, consistent representation when it gives you the latitude and longitude answer. The geocoder even goes so far as to look past poor punctuation and strange abbreviations, which is great if you're taking the input from a visitor to your site.

Like most of the geocoders available on the market, Google limits the number of geocoding requests that you can make before it cuts you off. The Google limit is a generous 50,000 lookups per API key per day, provided you space them out at a rate of one every 1.75 seconds (as of the time of publishing). To maximize this limit and your bandwidth, we suggest you use the server-side caching approach discussed in the “Caching Lookups” section later in this chapter.

Google Geocoder Responses

Let’s look at the Google geocoder’s response for a sample query adapted from the official documentation:

```
http://maps.google.com/maps/geo?q=1600+AmPhItHEaTRe+PKway+Mtn+View+CA&output=xml&key=your_api_key
```

This query returns the XML in Listing 4-2.

Listing 4-2. Sample Response from Google’s REST Geocoder

```
<kml>
<Response>
  <name>1600 AmPhItHEaTRe PKway Mtn View CA</name>
  <Status>
    <code>200</code>
    <request>geocode</request>
  </Status>
  <Placemark>
    <address>
      1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA
    </address>
    <AddressDetails>
      <Country>
        <CountryNameCode>US</CountryNameCode>
        <AdministrativeArea>
          <AdministrativeAreaName>CA</AdministrativeAreaName>
          <SubAdministrativeArea>
            <SubAdministrativeAreaName>Santa Clara</SubAdministrativeAreaName>
            <Locality>
              <LocalityName>Mountain View</LocalityName>
              <Thoroughfare>
                <ThoroughfareName>1600 Amphitheatre Pkwy</ThoroughfareName>
              </Thoroughfare>
              <PostalCode>
                <PostalCodeNumber>94043</PostalCodeNumber>
              </PostalCode>
            </Locality>
```

```
    </SubAdministrativeArea>
  </AdministrativeArea>
  </Country>
</AddressDetails>
<Point>
  <coordinates>-122.083739,37.423021,0</coordinates>
</Point>
</Placemark>
<Response>
</kml>
```

The response has three major components:

- **name:** The name is exactly what you fed into the geocoder, so you know if it interpreted your URL encoding properly.
- **Status:** This is the response code, which indicates whether the lookup was successful or if it failed. Table 4-1 lists the possible response codes and their meanings.
- **Placemark:** This is available only if the geocoding was successful and contains the information you're seeking. The placemark itself contains three important components:
 - **address:** The address is the full, nicely formatted string that Google actually used after it cleaned up the input you gave it. This is useful for a number of reasons, including storing something clean in your database and debugging when the answers seem to come back incorrectly.
 - **Point:** The point is a coordinate in 3D space and represents longitude, latitude, and elevation. Elevation data may or may not be available for a given answer, so take a 0 with a grain of salt, as it is the default and is also returned if no data is available.
 - **AddressDetails:** This is a block of more complicated XML that uses a standard format called eXtensible Address Language (xAL). Unless you're interested in extracting the individual pieces of the address for storage in your database or formatting on your screen, you could safely ignore this chunk of XML and get away with using only the status, address, and point information.

Note Upon launch of their geocoder, Google developers stated that all elevations would return 0 and that they were unsure when they would be able to supply elevation data. Before you use any of the elevation data, check the official API documentation online or the official Google Maps API blog (<http://googlemapsapi.blogspot.com/>) to see which regions now have elevation data available.

Table 4-1. *Google Geocoder Response Codes*

Code	Constant Name	Description
200	G_GEO_SUCCESS	No errors occurred; the address was successfully parsed and its geocode has been returned.
500	G_GEO_SERVER_ERROR	A geocoding request could not be successfully processed, yet the exact reason for the failure is not known.
601	G_GEO_MISSING_ADDRESS	The HTTP q parameter was either missing or had no value.
602	G_GEO_UNKNOWN_ADDRESS	No corresponding geographic location could be found for the specified address. This may be due to the fact that the address is relatively new, or it may be incorrect.
603	G_UNAVAILABLE_ADDRESS	The geocode for the given address cannot be returned due to legal or contractual reasons.
610	G_GEO_BAD_KEY	The given key is either invalid or does not match the domain for which it was given.
620	G_TOO_MANY_QUERIES	You have accessed the service too frequently and are either temporarily or permanently blocked from further use.

xAL

Defining a uniform way to describe addresses across 200 countries is no easy task. Some countries use street names; others don't. Some place higher importance on the postal code; others insist that the street number is most important. Some divide their "administrative" zones into a two-tier system of province/city; others use more tiers like state/county/city/locality. Whatever format is chosen must take all of these situations into account. OASIS has defined a format called xAL, which stands for eXtensible Address Language (in this case). Google has adopted it as a component of the XML response that its geocoder returns.

xAL uses a hierarchical data model (XML) since it seems like such a natural fit for addresses. For example, a country has states, a state has counties, a county has cities, a city has streets, and a street has individual plots of land. Some countries omit one or more of these levels, of course, but in general, that's not a problem.

However, you should realize that the xAL specification is designed to describe the address elements, not to be specific about the formatting and presentation of the address. There is no guarantee that the use of whitespace in the different elements will be consistent or even predictable, only that each type of data will be separated in a defined way. Using an XML-based format ensures that the data can be compared, sorted, and understood using simple programmatic methods.

For more information on xAL, visit the official site at <http://www.oasis-open.org/committees/ciq/ciq.html#6> or Google for the term "xAL address."

Google Geocoder Requests

Now let's look at a simple snippet of code that uses CURL to query the HTTP-based geocoding API and SimpleXML to parse the answer. Listing 4-3 shows this code.

Listing 4-3. *Using the Google Maps API Geocoder to Locate the Stores*

```
<?php

$api_key = "yourkey";

// Create a CURL object for later use
$ch = curl_init();
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

// Open the ronjons.xml file
$datafile = simplexml_load_file("ronjons.xml");
if (!$datafile) die("Unable to open input file!");

foreach ($datafile->store as $store) {
    // Construct the geocoder request string (URL)
    $url = "http://maps.google.com/maps/geo?output=xml&key=$api_key&q=";
    $q = $store->address.", ".$store->city.", ".$store->state.", ".$store->zip;
    $url .= urlencode($q);

    echo "\nStore: {$store->name}\n";
    echo "Source Address: $q\n";

    // Query Google for this store's longitude and latitude
    curl_setopt($ch, CURLOPT_URL, $url);
    $response = curl_exec($ch);

    // Use SimpleXML to parse our answer into something we can use
    $googleresult = simplexml_load_string($response);
    echo "Status: ".$googleresult->Response->Status->code."\n";
    if ($googleresult->Response->Status->code != 200)
        echo "Unable to parse Google response for {$store->name}!\n";
    else foreach ($googleresult->Response as $response) {
        foreach ($response->Placemark as $place) {
            list($longitude,$latitude) = split(",", $place->Point->coordinates);
            echo "Result Address: ".$place->address."\n";
            echo " Latitude: $latitude\n";
            echo " Longitude: $longitude\n";
        } // for each placemark
    } // for each Google result
} // for each store

// Close the CURL file and destroy the object
curl_close($ch);
?>
```

In this example, first we use `curl_setopt()` to define CURL's behavior while talking with Google. This includes telling CURL that we don't care about HTTP headers in `$googleresult` with the option `CURLOPT_HEADER = 0`, and instructing CURL to buffer the response (instead of sending it directly to the output buffer) with `CURLOPT_RETURNTRANSFER = 1`.

Next, we open your data file and parse it using SimpleXML. The resultant object is then used in the loop, which in turn creates a REST request URL for each store. Also note that you must use PHP's `urlencode()` function on the query portion of the string to ensure that the information is transmitted cleanly to the service.

Caution Remember that SimpleXML and XML in general are case-sensitive. The fact that our input XML is all lowercase means that we loop using `$datafile->store` (lowercase `s`), while the Google response uses title case, and therefore our inner loop uses `$googleresult->Response` (capital `R`). We've done this deliberately to remind you of this fact. Capitalization conventions are a matter of personal style.

Lastly, we give CURL our REST request and return the response into the variable `$googleresult` using the `curl_exec()` function. This returns the KML-style XML response that contains the meat of what we're interested in—the latitude and longitude, which we simply extract and echo to the screen for now. SimpleXML's node selectors make the job of accessing the data extremely trivial.

Listing 4-4 contains the output you should see when you execute this script. For convenience, you might prefer to output HTML `
` tags instead of newline characters (`\n`), so that you can see the results (without viewing the source) if you are using a browser to run the code, or you could prepend header(`'content-type:text/plain;'`) to the PHP file to convert the output to plaintext mode.

Listing 4-4. Output from the Google Geocoding Script

```
Store: "The Original" Ron Jon Surf Shop
Source Address: 901 Central Avenue, Long Beach Island , NJ, 08008
Status: 200
Result Address: 901 Central Ave, Barnegat Light, NJ 08008, USA
Latitude: 39.748586
Longitude: -74.111764
Result Address: 901 Central Ave, Surf City, NJ 08008, USA
Latitude: 39.661016
Longitude: -74.168010
Result Address: 901 Central Ave, Ship Bottom, NJ 08008, USA
Latitude: 39.649667
Longitude: -74.177253

Store: "One of a Kind" Ron Jon Surf Shop
Source Address: 4151 North Atlantic Avenue, Cocoa Beach, FL, 32931
Status: 200
Result Address: 4151 N Atlantic Ave, Cocoa Beach, FL 32931, USA
```

Latitude: 28.356453
Longitude: -80.608170

Store: Ron Jon Surf Shop - Sunrise
Source Address: 2610 Sawgrass Mills Circle, Sunrise, FL, 33323
Status: 200
Result Address: 2610 Sawgrass Mills Cir, Sunrise, FL 33323, USA
Latitude: 26.150899
Longitude: -80.316233

Store: Ron Jon Surf Shop - Orlando
Source Address: 5160 International Drive, Orlando, FL, 32819
Status: 200
Result Address: 5160 International Dr, Orlando, FL 32819, USA
Latitude: 28.469873
Longitude: -81.450311

Store: Ron Jon Surf Shop - Key West
Source Address: 503 Front Street, Key West, FL, 33040
Status: 200
Result Address: 503 Front St, Key West, FL 33040, USA
Latitude: 24.560287
Longitude: -81.805817

Store: Ron Jon Surf Shop - California
Source Address: 20 City Blvd., Orange, CA, 92868
Status: 200
Result Address: 100 City Blvd E, Orange, CA 92868, USA
Latitude: 33.782107
Longitude: -117.889878
Result Address: 2 City Blvd W, Orange, CA 92868, USA
Latitude: 33.779838
Longitude: -117.893568

Store: Ron Jon Cape Caribe Resort
Source Address: 1000 Shorewood Drive, Cape Canaveral, FL, 32920
Status: 200
Result Address: 699 Shorewood Dr, Cape Canaveral, FL 32920, USA
Latitude: 28.402944
Longitude: -80.604093

There are several interesting things to discuss in this result:

"The Original" Ron Jon Surf Shop: "The Original" store listed Long Beach Island as the city. Google doesn't recognize this as a valid city and has instead used the ZIP code to determine which cities might be more appropriate. More important, each of the answers differs by at least a few tenths of a degree, and this is a significant difference

(about 10 kilometers). It's up to you to decide how to handle this situation. A few suggestions might be to always use the first answer and assume that this is the one Google thinks is best. Another option would be to average the answers. Lastly, you could treat multiple Placemark nodes as a geocoding failure and ignore all of the data.

Ron Jon Surf Shop - California: For the store in California, the website lists the address as 20 City Boulevard but fails to give a direction. Google's two closest matches are 100 City Blvd E. and 2 City Blvd W. Both closest matches are returned in a separate Placemark node, and this is where the xAL data becomes very useful. Since each Placemark node is broken down in a consistent way, you can determine in which component the answer differs from your input. Doing so will allow you to write code that will make educated decisions about what to do with the answers. In this case, you probably want to assume that City Blvd is a straight line and employ some of the math in Chapters 10 and 11 to use a point approximately 20% of the way along the line between the two answers ($20 / (100 - 2) = \sim 20\%$).

Ron Jon Cape Caribe Resort: The Cape Caribe Resort doesn't geocode perfectly. This is probably because the resort is extremely new and the address hasn't yet been officially marked in the data that Google received. What you do in this case is again your decision, but our suggestion would be to assume that when you receive a single answer, it's the best you're going to get.

The Google JavaScript Geocoding API

Google also provides a means to geocode user input without the intervention of your server. This is a first in the realm of geocoders and enables a few things that can be cumbersome with server-side geocoding. This geocoder is built directly into the JavaScript API itself and makes Ajax calls directly to Google's servers from your visitor's computer.

The benefit is that it's quick and convenient because the API abstracts out all of the Ajax stuff, leaving you with a simple client-side JavaScript call. In addition to this, the latitude and longitude data can come back in such a way that it is trivial to use to place a point on your map using the API.

However, you need to keep in mind that while you don't have to contact *your own* server, you are talking to *a server*—Google's. So, you still need to carefully design your application to minimize the wait times your visitor sees while using your application.

Good and Bad Reasons to Use the JavaScript Geocoder

Here are some cases where it might be appropriate to use the JavaScript geocoder:

- When the visitor is inputting an address that you then plot on a map, but would never otherwise *store* for future use or display to another visitor. For example, this might be the case for a store locator that suggests locations based on proximity to a particular address.
- You are unable to create files on your web server that can be written to by your PHP scripts. This should almost never be the case, as a text file could (at the very worst) be set to world-writable (see the “Caching Lookups” section later in this chapter).
- Once Google exposes its route calculation capabilities, it may become useful for computing one endpoint of the path on the fly, but this is pure speculation.

A good reason to use this geocoder is to get a point from the user that is used *solely* for math calculations. We'll walk through an example of using the JavaScript geocoder in Chapter 10, where we show you how to add a corner to a polygon by either clicking on the map or entering an address into a text field.

It is *not* appropriate to geocode a list of points (such as the Ron Jon stores) on the fly client-side simply because it's easy. Overall, this would be a waste of bandwidth. This in turn means a longer download time for your visitor and a less-responsive map. Also, you definitely don't want to use this approach if the user is likely to be looking up the same thing over and over again.

Basically, while useful for quick-and-dirty mapping, the JavaScript geocoder isn't really useful for many professional map applications since you'll almost always have a server-side component. Thus, accessing the REST-based geocoder from your own Ajax service will allow you to integrate and consolidate the geocoding calls with the rest of your application (say, combining geocoding with looking up store hours). Another benefit of using your own server follows from Chapter 3's geocaching discussion about ensuring consistency by guaranteeing that your points are saved back to the server before showing them on a map. The same principle applies here. If you need to record any information at all back to your own server, you might as well use the REST-based geocoder to do the lookups and save yourself one Ajax call.

Client-Side Caching

Google has made a significant effort to limit the impact of lazy mappers (not you!) who will use the JavaScript geocoder just because it's easy. Aside from pleading with developers to "please cache your lookups" when it announced the geocoder, Google has integrated a client-side geocoding cache into the API. It is on by default and merely uses your visitors' RAM to store things they've previously looked up in case they look the same thing up again. You don't need to do anything special to use this cache, but there is something special you can do with it: you can seed it with information you already have. This means that you could precompute all of the addresses for your stores server-side, and then seed the client-side cache with the data. In certain applications, this could provide a huge speed boost for your map.

As of the time of publishing, the jury is still out on the best way to use some of these shiny new features. The official Google Maps API newsgroup is gushing with discussion about the best ways to do things and when to use the client-side cache and JavaScript geocoder to the best effect. We suggest that you check our website (<http://googlemapsbook.com>) and the official documentation to see what the current best practices are when you read this.

The Yahoo Geocoding API

Currently, the Yahoo Geocoding API (<http://developer.yahoo.net/maps/rest/V1/geocode.html>) is really useful only for geocoding addresses in the United States, though with competition from Google, we're sure this will change. Before Google's geocoder came along, this was the geocoder of choice for many people doing US-centric mashups using both the Google Maps API and the Yahoo Maps API. The only real limitation is that you can make only 5,000 lookup requests per day (per IP address).

Caution The rate limit for Yahoo is based on a 24-hour window, not a calendar day. This window begins when you first send a request to the service and is reset 24 hours later. Also the window does not “slide” (as it does with other services), meaning that it’s not a count of the requests made in the *last* 24-hours, but rather a fixed time frame. For a more thorough explanation of the rate limiting in the Yahoo Geocoding Web Service, please visit <http://developer.yahoo.net/search/rate.html>.

To use the API, you must register for a Yahoo application ID (like the Google API key you received in Chapter 2). To obtain your application ID, visit http://api.search.yahoo.com/webservices/register_application after logging in to your Yahoo account. If you do not have a Yahoo account, you’ll need to create one before proceeding. Once you have your application ID, you’ll need to include it in the requests to the service.

Like the Google geocoder, the Yahoo service is REST-based and requires you to append URL-encoded parameters onto the end of the request URL, as listed in Table 4-2.

Table 4-2. Request Parameters to the Yahoo Geocoding API

Parameter	Value	Description
appid	String (required)	The application ID you obtained from Yahoo.
street	String	The name and number of the street address. The number is optional but can improve accuracy.
city	String	The name of the city or town.
state	String	The name of the state, either spelled-out in full or as the two-letter abbreviation, which is more accurate.
zip	Integer	The five-digit ZIP code. This could also be a string of five digits, a dash, and the four-digit extension.
location	String	A free-form string representing an address.*
output	String	The format for the output. Possible values are xml (the default) or php. If php is requested, the results will be returned in serialized PHP format.

**The location parameter overrides the street, city, state, and zip parameters, and allows you to enter many different common formats for addresses. Thus, you are relying on Yahoo to parse the string accurately and as you intended, much like the Google service does. Yahoo’s geocoder is quite good at doing this parsing (for the same reasons as Google’s geocoder), so unless you already have the data broken out into components, your best bet might be to use the single location parameter instead of the individual parameters.*

Yahoo Geocoder Responses

The following is an example of a request for geocoding the Apress headquarters:

```
http://api.local.yahoo.com/MapsService/V1/geocode?appid=YOUR_APPLICATION_ID&street=&street=2560+Ninth+Street&city=Berkeley&state=CA&zip=94710
```

This returns the XML shown in Listing 4-5.

Listing 4-5. *Sample Response from the Yahoo Geocoding API*

```

<?xml version="1.0" encoding="UTF-8"?>
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="urn:yahoo:maps" xsi:schemaLocation="urn:yahoo:maps
  http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
  <Result precision="address" warning="The exact location could not be found,
  here is the closest match: 2560 9th St, Berkeley, CA 94710">
    <Latitude>37.859569</Latitude>
    <Longitude>-122.291673</Longitude>
    <Address>2560 9TH ST</Address>
    <City>BERKELEY</City>
    <State>CA</State>
    <Zip>94710-2500</Zip>
    <Country>US</Country>
  </Result>
</ResultSet>

```

For the purposes of this discussion, we will ignore the `xmlns:` and `xsi:` namespaces. What we care about is the `Result` node and the elements inside it.

Caution As with the Google service, it is possible to get a `ResultSet` with multiple `Result` values. If you would like to see this, try geocoding The White House (1600 Pennsylvania Avenue, Washington DC) while leaving out the ZIP code.

The `Result` node has two attributes in this case:

precision: This is a string indicating how accurate Yahoo thinks the answer is. This can be one of eight values at the moment: `address`, `street`, `zip+4`, `zip+2`, `zip`, `city`, `state`, or `country`. Changes to this list and additional information can be found in Yahoo's API developer documentation (<http://developer.yahoo.net/maps/rest/V1/geocode.html>).

warning: In our experience, nearly all requests had an "exact location could not be found" warning. This seems to occur for valid addresses whenever the capitalization of the street name, abbreviation of the street type, or spelling in the address don't exactly match the form in the database. In the example in Listing 4-5, it happens because the word "Ninth" is spelled out in full, and the Yahoo database has it listed as "9th." Using the `warning` node to determine if Yahoo's answer is a good match can be tricky, so for now, let's assume that the first answer in the result set is always the *best* answer (but not necessarily the *right* answer).

Next, we have the actual result fields corresponding to latitude, longitude, address, city, state, ZIP code, and country. Most of this data probably corresponds to the information you used to make the request; however, getting back all of this information is useful in picking the "right" answer in the event of Yahoo returning multiple matches. For now, the latitude and longitude fields are the ones we're most interested in, as those will be used to plot the Ron Jon store locations on our map.

Yahoo Geocoder Requests

So now that you have a handle on what you should be expecting out of the Yahoo API, let's create some PHP code to automate this process. Listing 4-6 shows the script.

Listing 4-6. *Using the Yahoo Geocoding API to Locate the Stores*

```
<?php
// Your Yahoo! Application id
$appid = "YOUR_YAHOO_APPLICATION_ID";

// Create a CURL object for later use
$ch = curl_init();
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

// Open the ronjons.xml file
$datafile = simplexml_load_file("ronjons.xml");
if (!$datafile) die("Unable to open input file!");

foreach ($datafile->store as $store) {
    // Construct the request string
    $url = "http://api.local.yahoo.com/MapsService/V1/geocode?appid=$appid";
    if ($store->address) $url .= "&street=".urlencode($store->address);
    if ($store->city) $url .= "&city=".urlencode($store->city);
    if ($store->state) $url .= "&state=".urlencode($store->state);
    if ($store->zip) $url .= "&zip=".$store->zip;

    echo "Store: {$store->name}\n";

    // Query Yahoo for this store's lat/long
    curl_setopt($ch, CURLOPT_URL, $url);
    $response = curl_exec($ch);

    // Use SimpleXML to parse our answer into something we can use
    $yahoorresult = simplexml_load_string($response);
    if (!$yahoorresult) echo "Unable to parse Yahoo response for {$store->name}!\n";
    else foreach ($yahoorresult->Result as $result) {
        echo "Result Precision: {$result['precision']}\n";
        if ($result['precision'] != "address") {
            echo "Warning: {$result['warning']}\n";
            echo "Address: {$result->Address}\n";
        }
        echo "Latitude: {$result->Latitude}\n";
        echo "Longitude: {$result->Longitude}\n\n";
    } // for each Yahoo result
} // for each store
```

```
// Close the CURL file and destroy the object
curl_close($ch);
?>
```

The code in Listing 4-6 is similar to the one for the Google geocoder (Listing 4-3). In fact, this is a template we will use a few more times in this chapter, and one that will serve you well for most REST-based services that return XML. The only real difference in the Yahoo example is that we've chosen to use the individual parameters since our data file already has them split up. This means that we need to use PHP's `urlencode()` on any parameter that might need it (those with spaces or special characters, for example), instead of on a single mammoth string. If you used the location parameter, this example could probably be 95% identical to the one in Listing 4-3.

We also check for the presence of each option before appending it to the URL of the REST request, despite the fact that Yahoo will silently ignore blank inputs. After all, defensive programming is always good practice, no matter how trivial the task—especially for experimental code that will probably grow into production code.

Listing 4-7 gives the resulting output from Listing 4-6.

Listing 4-7. *Output from the Yahoo Geocoding Script*

```
Store: "The Original" Ron Jon Surf Shop
Result Precision: address
Latitude: 39.6351
Longitude: -74.1883
```

```
Store: "One of a Kind" Ron Jon Surf Shop
Result Precision: address
Latitude: 28.356577
Longitude: -80.608069
```

```
Store: Ron Jon Surf Shop - Sunrise
Result Precision: address
Latitude: 26.156292
Longitude: -80.316945
```

```
Store: Ron Jon Surf Shop - Orlando
Result Precision: address
Latitude: 28.469972
Longitude: -81.450143
```

```
Store: Ron Jon Surf Shop - Key West
Result Precision: address
Latitude: 24.560448
Longitude: -81.805998
```

```
Store: Ron Jon Surf Shop - California
Result Precision: address
Latitude: 33.783329
Longitude: -117.890562
```

```
Store: Ron Jon Cape Caribe Resort
Result Precision: street
Warning:
Address: [600-699] SHOREWOOD DR
Latitude: 28.40232
Longitude: -80.59554
```

```
Result Precision: street
Warning:
Address: SHOREWOOD DR
Latitude: 28.40168
Longitude: -80.59774
```

Note You may need to view the source to see formatted output from Listing 4-7.

The only real surprise here is the last entry, Cape Caribe Resort, failed to geocode any more accurately than the general location of the street. This seems to corroborate Google's answer quite nicely (remember that it gave us 699 Shorewood instead of 1000 Shorewood). For now, simply remember that you'll always need to do some sort of error checking on the results or you might end up sending your customers to the wrong place. This entry also shows an example of multiple results being returned, as discussed earlier.

A possible solution to the ambiguous answer problem is to cross-reference (and average) the answers you get from one service (Google) with another (Yahoo). This is an onerous task if done for all of the data, but might be an excellent solution for your particular application if applied only to data that gives you grief.

Geocoder.us

Let's adapt our code for another US-centric geocoding service. Geocoder.us is a very popular service and was introduced well before Yahoo's and Google's services hit the market. For a long while, it was the measuring stick against which all other services were compared. The service was developed by two enterprising programmers, who took the freely available 2004 US Census Bureau's data and converted it into a web service.

Note The developers of Geocoder.us have made the Perl code that they wrote for their service available under an open source license and a module called `Geo::Coder::US`. If this interests you, then Chapter 11 may also interest you. In Chapter 11, we dig deep into the US Census data to build our own geocoder from scratch using PHP instead of Perl.

Just as with the Google and Yahoo services, there are limitations to the Geocoder.us service. The free service cannot be used for commercial purposes, and is rate-limited to prevent abuse,

though the developers haven't published exactly what the limit is. You can purchase a high-volume or commercial account that will get you four lookups per penny (20,000 lookups for \$50) with no rate limiting whatsoever.

Geocoder.us offers four different ways to access its web services: an XML-RPC interface, a SOAP interface, a REST interface that returns an RDF/XML document, and a REST interface that returns a plaintext CSV result. The accuracy, methods, and return values are equivalent across all of these interfaces. It's merely a matter of taste as to which one you'll use. For our example, we'll use the REST-based service and the CSV result (for some variety).

The following is an example of a Geocoder.us request for geocoding the Apress headquarters:

```
http://geocoder.us/service/csv/geocode?address=2560+Ninth+Street,+Berkeley+CA+94710
```

This returns the CSV string 37.859524,-122.291713,2560 9th St,Albany,CA,94710. You can see that it has mistaken Berkeley for Albany, despite the fact that the ZIP codes match. The latitude and longitude are nearly identical to the results Yahoo gave.

Let's again reuse the code from Listing 4-3 and adapt it to suit this new service. As with the Google geocoder, only one parameter is passed into this REST service, and it is called `address`. At minimum, either a city and state or a ZIP code must be contained in the `address` parameter. Listing 4-8 shows the adapted code.

Caution The code in Listing 4-8 takes a while to run. We'll discuss why in a moment, but for now be patient.

Listing 4-8. *Using the Geocoder.us Service to Locate the Stores*

```
<?php
// Create a CURL object for later use
$ch = curl_init();
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

// Open the ronjons.xml file
$datafile = simplexml_load_file("ronjons.xml");
if (!$datafile) die("Unable to open input file!");

foreach ($datafile->store as $store) {
    // Construct the request string
    $url = "http://geocoder.us/service/csv/geocode?address=";
    $address = "";
    if ($store->address) $address .= $store->address." ";
    if ($store->city) $address .= $store->city." ";
    if ($store->state) $address .= $store->state." ";
    if ($store->zip) $address .= $store->zip;
    $url .= urlencode($address);

    echo "Store: {$store->name}\n";
}
```

```

// Query Geocoder.us for this store's lat/long
curl_setopt($ch, CURLOPT_URL, $url);
$response = curl_exec($ch);

// Split up the CSV result into components
list($lat,$long,$address,$city,$state,$zip) = split(",",$response);
echo "Latitude: $lat\n";
echo "Longitude: $long\n\n";
} // for each store

// Close the CURL file and destroy the object
curl_close($ch);
?>

```

The only real difference here is with the CSV-style response. We've used a convention for splitting here that is common to the code snippets found on <http://www.php.net>, namely, using `list()` to get named strings instead of an array when calling the `split()` function. Listing 4-9 shows the output of the code in Listing 4-8.

Listing 4-9. *Output from the Geocoder.us Script*

```

Store: "The Original" Ron Jon Surf Shop
Latitude: 39.649509
Longitude: -74.177136

Store: "One of a Kind" Ron Jon Surf Shop
Latitude: 28.356433
Longitude: -80.608227

Store: Ron Jon Surf Shop - Sunrise
Latitude: 26.150513
Longitude: -80.316476

Store: Ron Jon Surf Shop - Orlando
Latitude: 28.466795
Longitude: -81.449860

Store: Ron Jon Surf Shop - Key West
Latitude: 24.560083
Longitude: -81.806069

Store: Ron Jon Surf Shop - California
Latitude: 33.781086
Longitude: -117.892520

Store: Ron Jon Cape Caribe Resort
Latitude: 2: couldn't find this address! sorry
Longitude:

```

When executing the code, the first thing you'll probably notice is that this request takes a long time to run. We believe this is a result of Geocoder.us rate limiting being based on requests per minute instead of requests per day. When testing, it took well over a minute to geocode just the seven points in the `ronjons.xml` data file.

The next thing you'll see if you look carefully is that the latitude and longitude results are the same as those from Yahoo only to three decimal places (on average). This is not a large difference and is the result of using different interpolation optimizations on the same data set, which we'll discuss in Chapter 11.

Notice that “The Original” store has given us a single answer this time, instead of multiple answers, and that the resort has given us grief yet again, except in this case, we didn't even get a best guess.

Note To determine just how large a distance difference the various geocoders give you for each of the results, you'll need to use the spherical distance equations (such as the Haversine method) we provide in Chapter 10.

Geocoder.ca

Geocoder.ca is similar to the service provided by Geocoder.us, but it is specifically targeted at providing information about Canada. (This service is in no way affiliated with Geocoder.us, and it uses a completely different data set, provided by Statistics Canada.)

The people behind Geocoder.ca built it specifically for their own experiments with the Google Maps API when they had trouble finding a timely, accurate, and cost-effective solution for geocoding Canadian addresses. They obtained numerous sources of data (postal, census, and commercial) and cross-referenced everything to weed out the inevitable errors in each set. This means that the Geocoder.ca service is quite possibly the *most* accurate information for Canada so far. (However, now that Google's solution covers Canada with relatively good accuracy, we're afraid that this extremely comprehensive service will become marginalized.)

Geocoder.ca provides a lot of neat features like intersection geocoding, reverse geocoding, and a suggestion system for correcting mistyped (or renamed) street names—none of which are provided by Google's geocoder, or any other for that matter. We don't cover any of these alternative features in this chapter, but you can find more information about them at their website if you're interested.

Remember that there is still no free lunch, so as with the other services, there are also limitations on the Geocoder.ca service. The free service is limited to between 500 and 2000 lookups per day per source IP address, depending on server load (light days you get more; heavy days less). The developers are willing to extend the limits for nonprofit organizations, but everyone else will need to purchase an account for commercial uses. The cost is currently the same as Geocoder.us: 20,000 lookups for \$50. Purchasing a commercial account might be an excellent way to cross-reference Google's multiple-result answers quickly, cheaply, and effectively.

An example of a query for geocoding the CN Tower in Toronto, Ontario is as follows:

```
http://geocoder.ca/?&stno=301&addresst=Front%2BStreet%2BWest&city=Toronto&prov=ON&postal=M5V2T6&geoit=XML.
```

This yields the exceedingly simple XML result in Listing 4-10.

Listing 4-10. *Sample Response from Geocoder.ca*

```
<?xml version="1.0" encoding="UTF-8" ?>
<geodata>
    <latt>43.643865000</latt>
    <longt>-79.388545000</longt>
</geodata>
```

Notice that the XML response uses `latt` and `longt`. The trailing `t` is easy to miss when reading the raw XML.

For an example, the Ron Jon Surf Shop data will not work, since the chain has yet to open a store in Canada. Instead, we'll again use the CN Tower in Toronto, Ontario. The address for the CN Tower is 301 Front Street West, Toronto, Ontario M5V 2T6 Canada. Listing 4-11 shows a small PHP snippet for geocoding this single address, which could easily be looped and abstracted as in previous examples to do multiple addresses. Feel free to substitute your own address if you live in the Great White North or know someone who does.

Listing 4-11. *Using Geocoder.ca to Locate the CN Tower in Toronto*

```
<?php
// Address to geocode (the CN Tower)
$street_no = "301";
$street = "Front Street West";
$city = "Toronto";
$prov = "ON";
$postal = "M5V2T6";

// Create a CURL object for later use
$ch = curl_init();
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

// Construct the request string
$url = "http://geocoder.ca/?";
$url .= "&stno=".urlencode($street_no);
$url .= "&address=".urlencode($street);
$url .= "&city=".urlencode($city);
$url .= "&prov=".$prov;
$url .= "&postal=".$postal;
$url .= "&geoit=XML";

// Query Geocoder.ca for the lat/long
curl_setopt($ch, CURLOPT_URL, $url);
$response = curl_exec($ch);

// Use SimpleXML to parse our answer into something we can use
$resultset = simplexml_load_string($response);
if (!$resultset) die("Unable to parse the response!");
```

```
echo "The CN tower is located here:\n";
echo "Latitude: {$resultset->latt}\n";
echo "Longitude: {$resultset->longt}\n";

// Close the CURL file and destroy the object
curl_close($ch);
?>
```

The most important lines in Listing 4-11 are highlighted in bold. The first is that the Geocoder.ca service prefers you to split out the street number from the street name. This isn't strictly necessary, but it does imply that greater accuracy can be achieved by doing so. The second is that the `goit` parameter *must* be included. At this point, there is no alternative value for this parameter, but there probably will be in the future. Lastly, when parsing the results, again, remember that the XML response uses `latt` and `longt`.

Listing 4-12 shows the output from Listing 4-11.

Listing 4-12. *Output from our Geocoder.ca Script*

```
The CN tower is located here:
Latitude: 43.643865000
Longitude: -79.388545000
```

When you compare this answer with the one Google gives you (43.642411, -79.386649) by clicking on a map, as in Chapter 3, you see that Geocoder.ca has done an excellent job of finding the correct coordinates for the CN Tower given its street address.

Services for Geocoding Addresses Outside Google's Coverage

For addresses outside the set provided by Google's geocoder, the job becomes much more difficult due to the lack of good, freely available data. In Chapter 11, you'll see how to create your own service from some sources of free data for the UK and the US. Maybe some of you will be inspired to find data for your country and create a service for the rest of us.

For now, however, we're simply going to share a few of the geocoding services we've found to date for areas outside Google's coverage area. We can't guarantee the accuracy or completeness of data from these services, since we don't have any real addresses to test with or enough knowledge of the local geography to wing it. We'll try to keep an updated list of services as we hear about them on our website at <http://googlemapsbook.com/geocoders>. Please let us know if you find or make more!

Geonames.org

Geonames.org has quite a few web services that might fit your needs. There is a full-text search of its database of place names, landmarks, and other geospatial data at <http://www.geonames.org/export/geonames-search.html>. However, you can also find (partial) postal code lookups for many countries (currently over 50), as well as reverse geocoding solutions for finding the name of the country or closest named feature for a given latitude and longitude (reverse geocoding). In Appendix A, we discuss the use of complete database dumps as a possible means to acquire the data you need to build your application without using external geocoding services.

ViaMichelin.com

One interesting solution for geocoding addresses in western Europe is <http://www.viamichelin.com>. The company that runs this service is part of the same company that makes Michelin tires (remember the Michelin Man?). The service offers route calculation, geocoding, and even an alternative source of map data. ViaMichelin is in competition with Google when it comes to maps, but for European locations where Google does not yet have geocoding services, the ViaMichelin solution could mean the difference between a successful project and a failure.

Bulk Geocoders

Many bulk geocoding services out there will accept a CSV or Excel file from you, determine latitude and longitude to the best of their ability, and give you the results a few hours to a few days later. These services typically charge a per-point fee when they are successful and nothing when they are not. Many of them use the Microsoft MapPoint Web Service to do the work. The quality of the data varies (with provider, price, and country), so we suggest that you do your research before hiring one of them to geocode your points.

Caching Lookups

As programmers, we hate wasting resources, and as service providers, we hate having our resources wasted. Therefore, for many of the examples in the rest of this book, you'll be precomputing the latitude and longitude programmatically and storing that information along with the point data you want to use in your mashup. This saves your bandwidth by not requiring unnecessary CURL/API requests, and saves the bandwidth of the services you'll be using for geocoding. Best of all, it provides a much faster user experience for your map visitors, which is almost always the single largest factor in determining the success of a website or service.

Caution Caching is not *always* the right answer. In some of the more novel mashups we've seen, the data is so dynamic that caching the latitude and longitude of the plotted point is actually more of a waste than not caching it. These examples are typically mashups where a single given point is plotted for only one or two visitors before it's never seen again, such as plotting the current position of a GPS device.

To cache the data for your store locator map, you'll modify the code in Listing 4-6 to create a script (Listing 4-13) that does a bulk geocoding of all of the stores and adds the latitude and longitude to the data file in Listing 4-14. In the next section, you'll use this data file to make your map, and assume that the stores already have latitude and longitude values associated with them.

Listing 4-13. Modified Code Showing Write-Back of Cached Data

```
<?php
// Your Yahoo! Application Code
$appid = YOUR_YAHOO_APPLICATION_ID;
```

```

// Create a CURL object for later use
$ch = curl_init();
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

// Open the ronjons.xml file
$datafile = simplexml_load_file("ronjons.xml");

// Open a file to store our consolidated information in
$newfile = fopen("ronjons_cache.xml", "w+");
fputs($newfile, '<?xml version="1.0" encoding="UTF-8"?>'. "\n");
fputs($newfile, '<stores>'. "\n");

foreach ($datafile->store as $store) {
    // Construct the request string
    $url = "http://api.local.yahoo.com/MapsService/V1/geocode?appid=$appid";
    if ($store->address) $url .= "&street=".urlencode($store->address);
    if ($store->city) $url .= "&city=".urlencode($store->city);
    if ($store->state) $url .= "&state=".urlencode($store->state);
    if ($store->zip) $url .= "&zip=".trim($store->zip);

    // Query Yahoo for this store's lat/long
    curl_setopt($ch, CURLOPT_URL, $url);
    $response = curl_exec($ch);

    // Use SimpleXML to parse our answer into something we can use
    $yahooresult = simplexml_load_string($response);
    foreach ($yahooresult->Result as $result) {
        $latitude = $result->Latitude;
        $longitude = $result->Longitude;
    } // for each Yahoo Result

    // Lastly output the XML to our file
    fputs($newfile, ' <store>'. "\n");
    fputs($newfile, ' <name>'.trim($store->name).'</name>'. "\n");
    fputs($newfile, ' <address>'.trim($store->address).'</address>'. "\n");
    if ($store->address2)
        fputs($newfile, ' <address2>'.trim($store->address2).'</address2>'. "\n");
    fputs($newfile, ' <city>'.trim($store->city).'</city>'. "\n");
    fputs($newfile, ' <state>'.trim($store->state).'</state>'. "\n");
    fputs($newfile, ' <zip>'.trim($store->zip).'</zip>'. "\n");
    fputs($newfile, ' <phone>'.trim($store->phone).'</phone>'. "\n");
    fputs($newfile, ' <pin>'.trim($store->pin).'</pin>'. "\n");
    fputs($newfile, ' <latitude>'.trim($latitude).'</latitude>'. "\n");
    fputs($newfile, ' <longitude>'.trim($longitude).'</longitude>'. "\n");
    fputs($newfile, ' </store>'. "\n");
} // for each store

```

```
// Close the CURL file and destroy the object
curl_close($ch);

// Close the new file freeing the memory
 fputs($newfile, '</stores>'. "\n");
 fclose($newfile);
?>
```

As you can see from the code, in our example we've elected to use the standard `fopen()`, `fwrite()`, and `fclose()` PHP commands to create the new file. SimpleXML doesn't provide a facility to add elements to an open XML document, and getting into a full-blown DOM example would be counterproductive.

Your modified script now creates a new file on the file system, as shown in Listing 4-14. You could have just as easily written the file on top of the existing `ronjons.xml` file, but if the conversion failed, you could lose all your existing data. The only trick is that you'll need to grant the web server user access to make new files in the folder you're working in, or you'll need to create a blank file and make it world-writable before executing this code.

Listing 4-14. *The New `ronjons_cache.xml` File with Caching (`ronjons_cache.xml`)*

```
<?xml version="1.0" encoding="UTF-8"?>
<stores>
  <store>
    <name>"The Original" Ron Jon Surf Shop</name>
    <address>901 Central Avenue</address>
    <city>Long Beach Island</city>
    <state>NJ</state>
    <zip>08008</zip>
    <phone>(609) 494-8844</phone>
    <pin>store</pin>
    <latitude>39.649652</latitude>
    <longitude>-74.177547</longitude>
  </store>
  <store>
    <name>"One of a Kind" Ron Jon Surf Shop</name>
    <address>4151 North Atlantic Avenue</address>
    <city>Cocoa Beach</city>
    <state>FL</state>
    <zip>32931</zip>
    <phone>(321) 799-8888</phone>
    <pin>store</pin>
    <latitude>28.356577</latitude>
    <longitude>-80.608069</longitude>
  </store>
  <store>
    <name>Ron Jon Surf Shop - Sunrise</name>
    <address>2610 Sawgrass Mills Circle</address>
    <address2>Suite 1415</address2>
```

```
<city>Sunrise</city>
<state>FL</state>
<zip>33323</zip>
<phone>(954) 846-1880</phone>
<pin>store</pin>
<latitude>26.156292</latitude>
<longitude>-80.316945</longitude>
</store>
<store>
  <name>Ron Jon Surf Shop - Orlando</name>
  <address>5160 International Drive</address>
  <city>Orlando</city>
  <state>FL</state>
  <zip>32819</zip>
  <phone>(407) 481-2555</phone>
  <pin>store</pin>
  <latitude>28.469972</latitude>
  <longitude>-81.450143</longitude>
</store>
<store>
  <name>Ron Jon Surf Shop - Key West</name>
  <address>503 Front Street</address>
  <city>Key West</city>
  <state>FL</state>
  <zip>33040</zip>
  <phone>(305) 293-8880</phone>
  <pin>store</pin>
  <latitude>24.560448</latitude>
  <longitude>-81.805998</longitude>
</store>
<store>
  <name>Ron Jon Surf Shop - California</name>
  <address>20 City Blvd.</address>
  <address2>West Building C Suite 1</address2>
  <city>Orange</city>
  <state>CA</state>
  <zip>92868</zip>
  <phone>(714) 939-9822</phone>
  <pin>store</pin>
  <latitude>33.783329</latitude>
  <longitude>-117.890562</longitude>
</store>
<store>
  <name>Ron Jon Cape Caribe Resort</name>
  <address>1000 Shorewood Drive</address>
  <city>Cape Canaveral</city>
  <state>FL</state>
```

```

<zip>32920</zip>
<phone>(321) 328-2830</phone>
<pin>resort</pin>
<latitude>28.40168</latitude>
<longitude>-80.59774</longitude>
</store>
</stores>

```

Note Ideally, you would be using some sort of relational database rather than a flat file on your file system for storing the points for your map. This would allow you to check each point at mapping time and look up (and cache) only those that don't have geocoded data yet. We'll begin using SQL databases in the next chapter.

The performance gain for caching just seven points is probably not noticeable on your high-speed connection. However, as your code scales to hundreds or thousands of data points, it will become critical. Also, if you are paying for each lookup, even at hundreds of lookups per dollar, the costs can add up quickly if a popular blog links to your map.

Note that the one place to avoid using caching is when your visitors are required to enter their current location so that the map can tailor itself to their situation and surroundings. This is often used in a store finder application where visitors enter their address and how far they are willing to drive to buy your product from a brick-and-mortar store.

Building a Store Location Map

Now that you have your stores and their latitude and longitude coordinates, you're ready to make your map. This will be a very basic map, but it serves our demonstration nicely. You'll customize the marker `GIcon` using the Ron Jon Surf Shop logo, and use the info window to display the store's address and phone number to visitors when they click the marker.

To make things a little easier, you can begin by taking the map you created in Chapter 2, with the addition of the icon creation from Chapter 3, and use the `map_data.php` file to convert your XML file of cached locations into the data structure from the `map_data.php` file in Listing 2-6. Listings 4-15, 4-16, and 4-17 show the modified `map_data.php` file, its output, and the `map_functions.js` file, respectively.

Listing 4-15. PHP Generation of the JavaScript (JSON) Data File in `map_data.php`

```

<?php
// Open the ronjons_cache.xml file and load the data for the pins
$datafile = simplexml_load_file("ronjons_cache.xml");
echo "var markers = [\n";
foreach ($datafile->store as $store) {
    $description = "{$store->address}<br />";
    if ($store->address2) $description .= "{$store->address2}<br/>";
    $description .= "{$store->city}, {$store->state}<br/>";
    $description .= "{$store->zip}<br/>";
    $description .= "Phone: {$store->phone}<br/>";
}

```

```

echo "{
    'latitude': {$store->latitude},
    'longitude': {$store->longitude},
    'name': '{$store->name}',
    'description': '$description'
},\n";
}

echo "];\n";
?>

```

Listing 4-16. *Generated JSON Data Structure in map_data.php*

```

var markers = [
{
    'latitude': 39.649652,
    'longitude': -74.177547,
    'name': '"The Original" Ron Jon Surf Shop',
    'description': '901 Central Avenue<br />Long Beach Island, NJ<br/>08008<br/>Phone: (609) 494-8844<br/>'
}, {
    'latitude': 28.356577,
    'longitude': -80.608069,
    'name': '"One of a Kind" Ron Jon Surf Shop',
    'description': '4151 North Atlantic Avenue<br />Cocoa Beach, FL<br/>32931<br/>Phone: (321) 799-8888<br/>'
}, {
    'latitude': 26.156292,
    'longitude': -80.316945,
    'name': 'Ron Jon Surf Shop - Sunrise',
    'description': '2610 Sawgrass Mills Circle<br />Suite 1415<br/>Sunrise, FL<br/>33323<br/>Phone: (954) 846-1880<br/>'
}, {
    'latitude': 28.469972,
    'longitude': -81.450143,
    'name': 'Ron Jon Surf Shop - Orlando',
    'description': '5160 International Drive<br />Orlando, FL<br/>32819<br/>Phone: (407) 481-2555<br/>'
}, {
    'latitude': 24.560448,
    'longitude': -81.805998,
    'name': 'Ron Jon Surf Shop - Key West',
    'description': '503 Front Street<br />Key West, FL<br/>33040<br/>Phone: (305) 293-8880<br/>'
}, {
    'latitude': 33.783329,
    'longitude': -117.890562,
    'name': 'Ron Jon Surf Shop - California',

```

```

    'description': '20 City Blvd.<br />West Building C Suite 1<br/>Orange, CA<br/>92868<br/>Phone: (714) 939-9822<br/>'
  }, {
    'latitude': 28.40168,
    'longitude': -80.59774,
    'name': 'Ron Jon Cape Caribe Resort',
    'description': '1000 Shorewood Drive<br />Cape Canaveral, FL<br/>32920<br/>Phone: (321) 328-2830<br/>'
  },
];

```

Listing 4-17. *map_functions.js from Chapter 2 Modified to Add Customized Icons and Info Windows*

```

var centerLatitude = 40.6897;
var centerLongitude = -95.0446;
var startZoom = 3;

var map;

var RonJonLogo = new GIcon();
RonJonLogo.image = 'ronjonsurfshoplogo.png';
RonJonLogo.iconSize = new GSize(48, 24);
RonJonLogo.iconAnchor = new GPoint(24, 14);
RonJonLogo.infoWindowAnchor = new GPoint(24, 24);

function addMarker(latitude , longitude, description) {
    var marker = new GMarker(new GLatLng(latitude, longitude), RonJonLogo);
    GEvent.addListener(marker, 'click',
        function() {
            marker.openInfoWindowHtml(description);
        }
    );

    map.addOverlay(marker);
}

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

    for(id in markers) {
        addMarker(markers[id].latitude , markers[id].longitude,
            markers[id].description);
    }
}

window.onload = init;

```

Figure 4-1 shows the completed map.



Figure 4-1. The completed map of the Ron Jon Surf Shop US locations

There you have it. The best bits of all of our examples so far combined into a map application. Data is geocoded, automatically cached for speed, and plotted quickly based on a JSON representation of our XML data file.

Summary

This chapter covered using geocoding services with your maps. It's safe to assume that you'll be able to adapt the general ideas and examples here to use almost any web-based geocoding service that comes along in the future. From here on, we'll assume that you know how to use these services (or ones like them) to geocode and cache your information efficiently.

This ends the first part of the book. In the next part, we'll move on to working with third-party data sets that have hundreds of thousands of points. Our examples will use the FCC's antenna structures database that currently numbers well over a hundred thousand points.