CHAPTER 7

Exploring Other Web APIs

In Chapter 6, you examined the Flickr API in great detail, so I'll turn now to other web APIs. Studying the Flickr API in depth is obviously useful if you plan to use it in your mashups, but I argue here that it's useful in your study of other APIs because you can draw from your understanding of the Flickr API as a point of comparison. (I'll cover the subject of HTTP web APIs bound to a JavaScript context in the next chapter. You'll take what you learn in Chapter 6 and this chapter and study the specific context of working within the modern web browser using JavaScript.)

How do you generalize from what you know about the Flickr API to other web APIs? I will use three major axes/categories for organizing my presentation of web APIs. (I'm presenting some heuristics for thinking about the subject rather than a watertight formula. This scheme won't magically enable you to instantly understand all the various APIs out there.) The categories I use are as follows:

* The *protocols* used by the API. Some questions that I'll discuss include the following: Is the API available with a REST interface? Does it use SOAP or XML-RPC?
* The *popularity* or *influence* of the API. It's helpful to understand some of the more popular APIs because of their influence on the field in general and also because popularity is an indicator of some utility. We'll look at how you might figure out what's popular.
* The *subject matter* of the APIs. Since APIs are often tied to specific subject matter, you'll naturally need to understand the basics of the subject to make sense of the APIs. What are some of those subject areas?

It doesn't take being too long in the field of web services to hear about REST vs. SOAP as a great divide—and hence the impetus for classifying web services by the *protocols* used. You already saw the terms REST and SOAP (as well as XML-RPC) in Chapter 6 to describe the request and response formats available to developers of the Flickr API. I focused on the Flickr REST formats because they are not only the easiest ones to work with but also they are the ones that are most helpful for learning other APIs.

In this chapter, I'll cover what XML-RPC and SOAP are about. Understanding just Flickr's REST request/response structure can get you far—but there are web APIs that have only XML-RPC or SOAP interfaces. So, I'll start by discussing XML-RPC and SOAP and show you the basics of how to use those two protocols. Also, I'll lay out tips for dealing with the practical complexities that sometimes arise in consuming SOAP services.

Note The term REST (an acronym for Representational State Transfer) was coined by Roy Fielding to

describe a set of architectural principles for networks. In Fielding's usage, REST is not specifically tied to HTTP or the Web. At the same time, a popular usage has arisen for REST to refer to exchanging messages over HTTP without using such protocols as SOAP and XML-RPC, which introduce an additional envelope around these messages. These two different usages of the term REST have caused confusion since it is possible to use HTTP to exchange messages without additional envelopes in a way that nonetheless does not conform to REST principles. If a creator of a service associates the service with the term REST (such as the Flickr REST interface), I will also refer to it as REST in this chapter.

Once you have a good understanding of the protocols and architectural issues behind HTTP web services, you're in a good position to consume any web API you come across—at least on a technical level. You still have to understand what a service is about and which services you might want to use. I will cover how to use Programmableweb.com as a great resource to learn about APIs in general. Programmableweb.com helps you understand which are the popular APIs as well as how APIs can be categorized by subject matter. I conclude the chapter with a study of two APIs: the API for YouTube as a simple REST interface and the Blogger API as a specific case of an entire class of APIs that share a uniform interface based on a strict usage of the HTTP methods.

XML-RPC

Although Flickr provides the option of using the XML-RPC and SOAP request and response formats in addition to REST, I wrote all my examples using the Flickr REST request format in Chapter 6. I'll show you how to use the XML-RPC protocol in this section and cover SOAP in the following section.

Tip Before taking on this section, it might be helpful to review Chapter 6's "A Refresher on HTTP" section and remind yourself of the structure of an HTTP request and response and the variety of HTTP request methods.

XML-RPC is defined at `http://www.xmlrpc.com/` as "remote procedure calling using HTTP as the transport and XML as the encoding." XML-RPC specifies how to form remote procedure calls in terms of requests and responses, each of which has parameters composed of some basic data types. There are XML-RPC libraries written in many languages, including PHP and Python.

A central point of having an XML-RPC interface for a web API is akin to that of an API kit—getting an interface that is a closer fit to the native structures and found in the programming language you are using. Let's consider a specific example to make this point.

Recall from Chapter 6 how to use the Flickr REST interface to search for public photos. You do an HTTP GET request on the following URL:

```
http://api.flickr.com/services/rest/?method=flickr.test.echo&api_key={api-key}
```

and parse the resulting XML (using, say, the `libcurl` and `simpleXML` libraries in PHP). Let's see how you do the same query using XML-RPC in Python and PHP for

comparison. In Python, you can use `xmlrpclib`, which is part of the standard Python distribution and is documented at

`http://docs.python.org/lib/module-xmlrpclib.html`

Here's a program to illustrate how to make a call to Flickr: one to `flickr.search.photos`. Note how parameters are passed in and how you can use the `ElementTree` library to parse the output. To use the `xmlrpclib` to make this call, you need to know that the XML-RPC server endpoint URL is as follows:

`http://api.flickr.com/services/xmlrpc/`

and you need to name your parameters and stick them into a dictionary. When I ran the following:

```
API_KEY = "[API-KEY]"

from xmlrpclib import ServerProxy, Error, Fault
server = ServerProxy("http://api.flickr.com/services/xmlrpc/")

try:
    from xml.etree import ElementTree as et
except:
    from elementtree import ElementTree as et

# call flickr.search.photos

args = {'api_key': API_KEY, 'tags':'flower', 'per_page':3}
try:
    rsp = server.flickr.photos.search(args)
except Fault, f:
    print "Error code %s: %s" % (f.faultCode, f.faultString)

# show a bit of XML parsing using elementtree
# useful examples:  http://www.amk.ca/talks/2006-02-07/
# context page for photo: http://www.flickr.com/photos/{user-id}/{photo-id}

# fixes parsing errors when accented characters are present
rsp = rsp.encode('utf-8')
print rsp
tree = et.XML(rsp)
print "total number of photos: %s" %(tree.get('total'))
for p in tree.getiterator('photo'):
    print "%s: http://www.flickr.com/photos/%s/%s" % (p.get("title"), ~CCC
p.get("owner"), p.get("id"))
```

I got this:

```
<photos page="1" pages="485798" perpage="3" total="1457392">
  <photo id="1236197537" owner="7823684@N06" secret="f58310acf3"
         server="1178" farm="2" title="Rainbow over flower" ispublic="1"
         isfriend="0" isfamily="0" />
  <photo id="1236134903" owner="27238986@N00" secret="fa461fb8e3" server="1036"
         farm="2" title="Watercolor" ispublic="1" isfriend="0"
         isfamily="0" />
  <photo id="1237043346" owner="33121739@N00" secret="7a116ff4af" server="1066"
```

```
        farm="2" title="Flowers" ispublic="1" isfriend="0" isfamily="0" />
</photos>

total number of photos: 1457392
Rainbow over flower: http://www.flickr.com/photos/7823684@N06/1236197537
Watercolor: http://www.flickr.com/photos/27238986@N00/1236134903
Flowers: http://www.flickr.com/photos/33121739@N00/1237043346
```

Note how the `xmlrpclib` library takes care of packaging the response and sending you back the XML payload (which doesn't have the `<rsp>` root node that is in the Flickr REST response). However, you still have to parse the XML payload. Whether using XML-RPC or REST is more convenient, you can judge for yourself.

Let's take a look at how some PHP code looks. There are two major PHP libraries for XML-RPC:

* `http://phpxmlrpc.sourceforge.net/`
* `http://pear.php.net/package/XML_RPC/`

Here I show how to use the `PEAR::XML_RPC` package. You can install it using PEAR:

```
pear install XML_RPC
```

The following program shows how to use `PEAR::XML-RPC` to do a number of things:

* You can retrieve the current time by making a call that requires no parameters (`currentTime.getCurrentTime`) from `http://time.xmlrpc.com`.
* In `search_example()`, you can make a specific call to `flickr.photos.search`.
* The class `flickr_client` shows how to generalize `search_example()` to handle more of the Flickr methods.

Here's the program:

```
<?php

// flickr_xmlrpc.php
// This code demonstrates how to use XML-RPC using the PEAR::XML-RPC library.
// gettime() is the simple example that involves
// calling a timeserver without passing in any parameters.
// search_example() shows a specific case of how to pass in some parameters
// for flickr.photos.search
// the flickr_client class generalizes search_example() to handle Flickr methods
// in general.

require_once('XML/RPC.php');
$API_KEY ='[API-KEY]';

function process_xmlrpc_resp($resp) {
  if (!$resp->faultCode()) {
      $val = $resp->value()->scalarval();
      return $val;
  } else {
    $errormsg = 'Fault Code: ' . $resp->faultCode() . "\n" . 'Fault Reason: ' .
```

```
      $resp->faultString() . "\n";
    throw new Exception ($errormsg);
  }
}

class flickr_client {

  protected $api_key;
  protected $server;

  public function __construct($api_key, $debug) {
    $this->api_key = $api_key;
    $this->server =
      new XML_RPC_Client('/services/xmlrpc','http://api.flickr.com',80);
    $this->server->setDebug($debug);
  }

  public function call($method,$params) {

    # add the api_key to $params
    $params['api_key'] = $this->api_key;

    # build the struct parameter needed
    foreach ($params as $key=>$val) {
      $xrv_array[$key] = new XML_RPC_Value($val,"string");
    }
    $xmlrpc_val = new XML_RPC_Value ($xrv_array,'struct');

    $msg = new XML_RPC_Message($method, array($xmlrpc_val));
    $resp = $this->server->send($msg);

    return process_xmlrpc_resp($resp);

  } //call

} //class flickr_client

function search_example () {
  GLOBAL $API_KEY;
  $server = new XML_RPC_Client('/services/xmlrpc','http://api.flickr.com',80);
  $server->setDebug(0);

  $myStruct = new XML_RPC_Value(array(
      "api_key" => new XML_RPC_Value($API_KEY, "string"),
      "tags" => new XML_RPC_Value('flower',"string"),
      "per_page" => new XML_RPC_Value('2',"string"),
      ), "struct");

  $msg = new XML_RPC_Message('flickr.photos.search', array($myStruct));
  $resp = $server->send($msg);

  return process_xmlrpc_resp($resp);
}
```

```
function gettime() {

  # http://www.xmlrpc.com/currentTime
  $server = new XML_RPC_Client('/RPC2','http://time.xmlrpc.com',80);
  $server->setDebug(0);

  $msg = new XML_RPC_Message('currentTime.getCurrentTime');
  $resp = $server->send($msg);

  return process_xmlrpc_resp($resp);

}

print "current time: ".gettime();
print "output from search_example \n" . search_example(). "\n";

$flickr = new flickr_client($API_KEY,0);

print "output from generalized Flickr client using XML-RPC\n";
print $flickr-
>call('flickr.photos.search',array('tags'=>'dog','per_page'=>'2'));
?>
```

What's Happening on the Wire?

XML-RPC is meant to abstract away how a remote procedure call is translated into an exchange of XML documents over HTTP so that you as a user of XML-RPC don't have to understand the underlying process. That's the theory with XML-RPC and especially with SOAP, an expansive elaboration on XML-RPC out of which it originally evolved. In practice, with the right tools and under certain circumstances, consuming services with XML-RPC or SOAP is a very simple, trouble-free experience.

At other times, however, you'll find yourself having to know more about the underlying protocol than you really need to know. For that reason, in the following sections I'll show you techniques for making sense of what XML is actually being exchanged and how it's being exchanged over HTTP. This discussion is meant as an explication of XML-RPC in its own right but also as preparation for studying the yet more complicated SOAP later in the chapter. But first, let's look at two tools that I use to analyze XML-RPC and SOAP: Wireshark and `curl`.

Using Wireshark and curl to Analyze and Formulate HTTP Messages

Wireshark (`http://www.wireshark.org/`) is an open source network protocol analyzer that runs on Windows, OS X, and Linux. With it, you can analyze network traffic flowing through your computer, including any HTTP traffic—making it incredibly useful for seeing what's happening when you are using web APIs (or, if you are curious, merely surfing the Web). Refer to the Wireshark site for instructions about how to install and run Wireshark for your platform.

Tip With Wireshark, I found it helpful to *turn off* the Capture Packets in Promiscuous Mode option. Also, for studying web service traffic, I filter for only HTTP traffic—otherwise, there is too much data to view.

`curl` (`http://curl.haxx.se/`) is another highly useful command-line tool for working with HTTP—among many other things:

> *curl is a command line tool for transferring files with URL syntax, supporting FTP, FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, TELNET, DICT, FILE and LDAP. curl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, cookies, user+password authentication (Basic, Digest, NTLM, Negotiate, kerberos . . .), file transfer resume, proxy tunneling, and a busload of other useful tricks.*

Go to `http://curl.haxx.se/download.html` to find a package for your platform. Be sure to look for packages that support SSL—you'll need it when you come to some examples later this chapter. Remember in particular the following documentation:

* `http://curl.haxx.se/docs/manpage.html` is the `man` page for `curl`.
* `http://curl.haxx.se/docs/httpscripting.html` is the most helpful page in many ways because it gives concrete examples.

To learn these tools, I suggest using `curl` to issue an HTTP request and using Wireshark to analyze the resulting traffic. For instance, you can start with the following:

```
curl http://www.yahoo.com
```

to see how to retrieve the contents of a web page. To see the details about the HTTP request and response, turn on the `verbose` option and make explicit what was implicit (that fetching the content of `http://www.yahoo.com` uses the HTTP GET method):

```
curl -v -X GET http://www.yahoo.com
```

You can get more practice studying Wireshark and the Flickr API by performing some function in the Flickr UI or in the Flickr API Explorer and seeing what HTTP traffic is exchanged. Try operations that don't require any Flickr permissions, and then try ones that require escalating levels of permissions. You can see certainly see the Flickr API being invoked and when HTTP `GET` vs. HTTP `POST` is used by Flickr—and specifically what is being sent back and forth.

I'll teach you more about `curl` in the context of the following examples.

Parsing XML-RPC Traffic

When you look at the documentation for the XML-RPC request format for Flickr (`http://www.flickr.com/services/api/request.xmlrpc.html`) and for the response format (`http://www.flickr.com/services/api/response.xmlrpc.html`), you'll find confirmation that the transport mechanism is indeed HTTP (just as it for the REST request and response). However, the request parameters and response are wrapped in many layers of

XML tags. I'll show you how to use Wireshark and `curl` to confirm for yourself what's happening when you use XML-RPC.

Here I use Wireshark to monitor what happens when I run the Python example that uses the `flickr.photos.search` method and then use `curl` to manually duplicate the same request to show how you can formulate XML-RPC requests without calling an XML-RPC library per se. Again, I'm not advocating this as a practical way of using XML-RPC but as a way of understanding what's happening when you do use XML-RPC.

When I ran the Python program and monitored the HTTP traffic, I saw the following request (an HTTP `POST` to `/services/xmlrpc/`):

```
POST /services/xmlrpc/ HTTP/1.0
```

It had the following HTTP request headers:

```
Host: api.flickr.com
User-Agent: xmlrpclib.py/1.0.1 (by www.pythonware.com)
Content-Type: text/xml
Content-Length: 415
```

and the following request body (reformatted here for clarity):

```
<?xml version='1.0'?>
<methodCall>
  <methodName>flickr.photos.search</methodName>
  <params>
    <param>
      <value><struct>
        <member>
          <name>per_page</name>
          <value><int>3</int></value>
        </member>
        <member>
          <name>api_key</name>
          <value><string>[API-KEY]</string></value>
        </member>
        <member>
          <name>tags</name>
          <value><string>flower</string></value>
        </member>
      </struct></value>
    </param>
  </params>
</methodCall>
```

The HTTP response (edited here for clarity) was as follows:

```
HTTP/1.1 200 OK
Date: Sun, 26 Aug 2007 04:33:29 GMT
Server: Apache/2.0.52
[...some cookies....]
Content-Length: 1044
Connection: close
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8" ?>
```

```
<methodResponse>
  <params>
    <param>
      <value>
        <string>
          &lt;photos page=&quot;1&quot; pages=&quot;485823&quot;
          perpage=&quot;3&quot; total=&quot;1457468&quot;&gt;
          &lt;photo id=&quot;1237314286&quot; owner=&quot;41336703@N00&quot;
          secret=&quot;372291c5f7&quot; server=&quot;1088&quot;
farm=&quot;2&quot;
          title=&quot;250807 047&quot; ispublic=&quot;1&quot;
isfriend=&quot;0&quot;
          isfamily=&quot;0&quot; /&gt;
          &lt;photo id=&quot;1236382563&quot; owner=&quot;70983346@N00&quot;
          secret=&quot;459e79fde3&quot; server=&quot;1376&quot;
farm=&quot;2&quot;
          title=&quot;Darling daisy necklace&quot; ispublic=&quot;1&quot;
          isfriend=&quot;0&quot; isfamily=&quot;0&quot; /&gt;
          &lt;photo id=&quot;1237257850&quot; owner=&quot;39312862@N00&quot;
          secret=&quot;fa9d15f9c3&quot; server=&quot;1272&quot;
farm=&quot;2&quot;
          title=&quot;Peperomia species&quot; ispublic=&quot;1&quot;
          isfriend=&quot;0&quot; isfamily=&quot;0&quot; /&gt;
          &lt;/photos&gt;
        </string>
      </value>
    </param>
  </params>
</methodResponse>
```

To make sense of the interchange, it's useful to study the XML-RPC specification (`http://www.xmlrpc.com/spec`) to learn that the Flickr XML-RPC request is passing in one `struct` that holds all the parameters. The request uses HTTP `POST`. What comes back in the response is an entity-encoded XML `<photos>` element (the results that we wanted from the API), wrapped in a series of XML elements used in the XML-RPC protocol to encapsulate the response. This process of serializing the request and deserializing the response is what an XML-RPC library does for you.

We can take this study of XML-RPC one more step. You can use `curl` (or another HTTP client) to confirm that you can synthesize an XML-RPC request independently of any XML-RPC library to handle the work for you. This is not a convenient way to do things, and it defeats the purpose of using a protocol such as XML-RPC—but this technique is helpful for proving to yourself that you really understand what is really happening with a protocol.

To wit, to call `flickr.photos.search` using XML-RPC, you need to send an HTTP `POST` request to `http://api.flickr.com/services/xmlrpc/` whose body is the same as what I pulled out using Wireshark. The call, formulated as an invocation of `curl`, is as follows:

```
curl -v -X  POST --data-binary "<?xml version='1.0' encoding='UTF-8'?> ~CCC

<methodCall><methodName>flickr.photos.search</methodName><params><param><value>
~CCC
```

```
<struct><member><name>per_page</name><value><int>3</int></value></member><member
> ~CCC
  <name>api_key</name><value><string>[API-KEY]</string></value></member><member>
~CCC

<name>tags</name><value><string>flower</string></value></member></struct></value
> ~CCC
  </param></params></methodCall>"  http://api.flickr.com/services/xmlrpc/
```

Note To write `curl` invocations that work from the command line of Windows, OS X, and Linux, I rewrote the XML to use single quotes to allow me to use double quotes to wrap the XML.

You can issue this request through `curl` to convince yourself that you are now speaking and understanding XML-RPC responses!

An XML-RPC library is supposed to hide the details you just looked at from you. One of the major practical problems that I have run into when using XML-RPC (and SOAP) is understanding for a given language and library how exactly to formulate a request. Notice some important lines from the examples. An essentialist rendition of the Python example is as follows:

```
server = ServerProxy("http://api.flickr.com/services/xmlrpc/")
args = {'api_key': API_KEY, 'tags':'flower', 'per_page':3}
rsp = server.flickr.photos.search(args)
rsp = rsp.encode('utf-8')
tree = et.XML(rsp)
print "total number of photos: %s" %(tree.get('total'))
```

Besides the mechanics of calling the right libraries, you had to know how to pass in the URL endpoint of the XML-RPC server—which is usually not too hard—but also how to package up the parameters. Here, I had to use a Python dictionary, whose keys are the names of the Flickr parameters. I then call `flickr.photos.search` as a method of server and get back XML.

The PHP example can be boiled down to this:

```
$server = new XML_RPC_Client('/services/xmlrpc','http://api.flickr.com',80);
$myStruct = new XML_RPC_Value(array(
    "api_key" => new XML_RPC_Value($API_KEY, "string"),
    "tags" => new XML_RPC_Value('flower',"string"),
    "per_page" => new XML_RPC_Value('2',"string"),
    ), "struct");
$msg = new XML_RPC_Message('flickr.photos.search', array($myStruct));
$resp = $server->send($msg);
$val = $resp->value()->scalarval();
```

Again, I knew what I had to tell PHP and the `PEAR::XML_RPC` library, and once someone provides you with skeletal code like I did here, it's not hard to use. However, it has been my experience with XML-RPC and especially SOAP that it takes a lot of work to come up with the incantation that works. Complexity is moved from having to process HTTP and XML directly (as you would have using the Flickr REST interface)

to understanding how to express methods and their parameters in the way a given higher-level toolkit wants from you.

SOAP

SOAP is a complicated topic of which I readily admit to having only a limited understanding. SOAP and the layers of technologies built on top of SOAP—WSDL, UDDI, and the various WS-* specifications (`http://en.wikipedia.org/wiki/WS-%2A`)—are clearly getting lots of attention, especially in enterprise computing, which deals with needs addressed by this technology stack. I cover SOAP and WSDL (and leave out the other specifications) in this book because some of the APIs you may want to use in creating mashups are expressed in terms of SOAP and WSDL. My goal is to provide practical guidance as to how to consume such services, primarily from the perspective of a PHP and Python programmer.

As with XML-RPC, SOAP and WSDL are supposed to make your life as a programmer easier by abstracting away the underlying HTTP and XML exchanges so that web services look a lot like making a local procedure call. I'll start with simple examples, using tools that make using SOAP and WSDL pretty easy to use, in order to highlight the benefits of SOAP and WSDL, and then I'll move to more complicated examples that show some of the challenges. Specifically, I'll show you first how to use a relatively straightforward SOAP service (geocoder.us), proceeding to a more complicated service (Amazon.com's ECS AWS), and then discussing what turns out to be unexpectedly complicated (the Flickr SOAP interface).

The Dream: Plug-and-Go Functionality Through WSDL and SOAP

As you learned in Chapter 6, the process of using the Flickr REST interface generally involves the following steps:

1. Finding the right Flickr method to use
2. Figuring out what parameters to pass in and how to package up the values
3. Parsing the XML payload

Although these steps are not conceptually difficult, they do tend to require a fair amount of manual inspection of the Flickr documentation by any developer working directly with the Flickr API. A Flickr API kit in the language of your choice might make it easier because it makes Flickr look like an object in that language. Accordingly, you might then be able to use the facilities of the language itself to tell you what Flickr methods are available and what parameters they take and be able to get access to the results without having to directly parse XML yourself.

You might be happy as a user of the third-party kit, but the author of any third-party kit for Flickr must still deal with the original problem of manually translating the logic and semantics of the Flickr documentation and API into code to abstract it away for the user of the API kit. It's a potentially tedious and error-prone process. In Chapter 6, I showed you how you could use the `flickr.reflection` methods to automatically list the available API methods and their parameters. Assuming that Flickr keeps the information

coming out of those methods up-to-date, there is plenty of potential to exploit with the reflection methods.

However, `flickr.reflection.getMethodInfo` does not currently give us information about the formal data typing of the parameters or the XML payload. For instance, `http://www.flickr.com/services/api/flickr.photos.search.html` tells us the following about the `per_page` argument: "Number of photos to return per page. If this argument is omitted, it defaults to 100. The maximum allowed value is 500." Although this information enables a human interpreter to properly formulate the `per_page` argument, it would be difficult to write a program that takes advantage of this fact about `per_page`. In fact, it would be useful even if `flickr.reflections.getMethodInfo` could tell us that the argument is an integer without letting us know about its range.

That's where Web Services Definition Language (WSDL) comes in as a potential solution, along with its typical companion, SOAP. There are currently two noteworthy versions of WSDL. Although WSDL 2.0 (documented at `http://www.w3.org/TR/2007/REC-wsdl20-20070626/`) is a W3C recommendation, it seems to me that WSDL 1.1, which never became a *de jure* standard, will remain the dominant version of WSDL for some time (both in WSDL documents you come across and the tools with which you will have easy access). WSDL 1.1 is documented at `http://www.w3.org/TR/wsdl`.

A WSDL document specifies the methods (or in WSDL-speak *operations*) that are available to you, their associated *messages*, and how they turned in concrete calls you can make, typically through SOAP. (There is support in WSDL 2.0 for invoking calls using HTTP without using SOAP.) Let me first show you concretely how to use WSDL, and I'll then discuss some details of its structure that you might want to know even if you choose never to look in depth at how it works.

geocoder.us

Consider the geocoder.us service (`http://geocoder.us/`) that offers both free noncommercial and for-pay commercial geocoding for U.S. addresses. You can turn to the API documentation (`http://geocoder.us/help/`) to learn how to use its free REST-RDF, XML-RPC, and SOAP interface. There are three methods supported by geocoder.us:

`geocode`: Takes a U.S. address or intersection and returns a list of results

`geocode_address`: Works just like `geocode` except that it accepts only an address

`geocode_intersection`: Works just like `geocode` except that it accepts only an intersection

Let's first use the interface that is most familiar to you, which is its REST-RDF interface, and consider the `geocode` method specifically. To find the latitude and longitude of an address, you make an HTTP `GET` request of the following form:

```
http://geocoder.us/service/rest/geocode?address={address}
```

For example, applying the method to the address of Apress:

```
http://geocoder.us/service/rest/geocode?address=2855+Telegraph+Ave%2C+Berkeley%2C+CA
```

gets you this:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
<geo:Point rdf:nodeID="aid78384162">
    <dc:description>2855 Telegraph Ave, Berkeley CA 94705</dc:description>
    <geo:long>-122.260070</geo:long>
    <geo:lat>37.858276</geo:lat>
</geo:Point>
</rdf:RDF>
```

Now let's make the same call using the SOAP interface. Instead of making the SOAP call directly to the `geocode` method, let's use the WSDL document for the service:

```
http://geocoder.us/dist/eg/clients/GeoCoderPHP.wsdl
```

Note Because the first WSDL document (`http://geocoder.us/dist/eg/clients/GeoCoder.wsdl`) referenced by geocoder.us apparently gives PHP 5 heartburn, I instead use the second WSDL document (`GeoCoderPHP.wsdl`) in this chapter.

I will use the WSDL document in a variety of ways to teach you the ideal usage pattern for WSDL, which involves the following steps:

* A SOAP/WSDL tool/library takes a given WSDL document and makes transparent the operations that are available to you.
* For a given operation, the SOAP/WSDL tool makes it easy for you to understand the possible input parameters and formulate the appropriate request message.
* The SOAP/WSDL tool then returns the response to you in some easy-to-parse format and handles any *faults* that come up in the course of the operation.

Using the oXygen XML Editor

My favorite way of testing a WSDL file and issuing SOAP calls is to use a visual IDE such as oXygen (`http://www.oxygenxml.com/`). Among the plethora of XML-related technologies supported by oXygen is the WSDL SOAP Analyser. I describe how you can use it to invoke the geocoder.us geocode operation to illustrate a core workflow.

Note oXygen is a commercial product. You can evaluate it for 30 days free of charge. XML Spy (`http://www.altova.com/`), another commercial product, provides a similar WSDL tool. I know of one open source project that lets you visually explore a WSDL document and invoke operations: the Web Services Explorer for the Eclipse project that is part of the Web Tools project (`http://www.eclipse.org/webtools/`).

When you start the WSDL SOAP Analyser, you are prompted for the URL of a WSDL file. You enter the URL for the geocoder.us WSDL (listed earlier), and oXygen reads the WSDL file and displays a panel with four subpanels. (Figure 7-1 shows the setup of this panel.) The first subpanel contains three drop-down menus for three types of entities defined in the WSDL file:

* Services
* Ports
* Operations

The geocoder.us WSDL file follows a pattern typical for many WSDL files: it has one *service* (`GeoCode_Service`) tied to one *port* (`GeoCode_Port`), which is tied, through a specific *binding*, to one or more *operations*. It's this list of operations that is the heart of the matter if you want to use any of the SOAP services. The panel shows three operations (`geocode`, `geocode_address`, and `geocode_intersection`) corresponding to the three methods available from geocoder.us.

Insert 858Xf0701.tif

Figure 7-1. The WSDL SOAP Analyser panel loaded with the geocoder.us WSDL

The values shown in the three other subpanels depend on the operation you select. The four subpanels list the parameters described in Table 7-1.

Table 7-1. Panels and Parameters from the WSDL Soap Analyser in oXygen

Panel	Parameter	Explanation
WSDL	Services	Drop-down menu of services (for example, `GeoCode_Service`)
	Ports	Drop-down menu of ports (for example, `GeoCode_Port`)
	Operations	Drop-down menu of operations (for example, `geocode`)
Actions	URL	For example, `http://rpc.geocoder.us/service/soap/`
	SOAP action	For example, `http://rpc.geocoder.us/Geo/Coder/US#geocode`
Request		The body of the request (you fill in the parameters)
Response		The body of the response (this is the result of the operation)

As someone interested in just using the geocode operation (rather understanding the underlying mechanics), you would jump immediately to the sample request that oXygen generates:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <oxy:geocode xmlns:oxy="http://rpc.geocoder.us/Geo/Coder/US/"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <location>STRING</location>
    </oxy:geocode>
  </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

To look up the address of Apress, you would replace this:

```
<location>STRING</location>
```

with the following:

```
<location>2855 Telegraph Ave, Berkeley CA 94705</location>
```

and hit the Send button on the Request subpanel to get the following to show up in the Response subpanel:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <namesp6:geocodeResponse
xmlns:namesp6="http://rpc.geocoder.us/Geo/Coder/US/">
      <geo:s-gensym23 xsi:type="SOAP-ENC:Array"
                      xmlns:geo="http://rpc.geocoder.us/Geo/Coder/US/"
        SOAP-ENC:arrayType="geo:GeocoderAddressResult[1]">
        <item xsi:type="geo:GeocoderAddressResult">
          <number xsi:type="xsd:int">2855</number>
          <lat xsi:type="xsd:float">37.858276</lat>
          <street xsi:type="xsd:string">Telegraph</street>
          <state xsi:type="xsd:string">CA</state>
          <zip xsi:type="xsd:int">94705</zip>
          <city xsi:type="xsd:string">Berkeley</city>
          <suffix xsi:type="xsd:string"/>
          <long xsi:type="xsd:float">-122.260070</long>
          <type xsi:type="xsd:string">Ave</type>
          <prefix xsi:type="xsd:string"/>
        </item>
      </geo:s-gensym23>
    </namesp6:geocodeResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

There you have it. Let's review what oXygen and a WSDL document could accomplish for you:

* You can get a list of *operations* available for the services and ports defined in the WSDL (not atypically one service and port combination).
* You are given a template for the body of the request with an indication of the data type of what you need to fill in.
* oXygen packages up the request, issues the HTTP request, handles the response, and presents you with the results.

To confirm that you understand the nuances of the `geocode` SOAP call, you can rewrite the SOAP request as a `curl` invocation—once you notice the role played by the two parameters that oXygen does pick up from the WSDL document:

* The SOAP action of `http://rpc.geocoder.us/Geo/Coder/US#geocode`. In SOAP 1.1, the version of SOAP used for geocoder.us, the SOAP action is transmitted as a `SOAPAction` HTTP request header.
* The URL (or *location*) to target the SOAP call: `http://rpc.geocoder.us/service/soap/`.

SOAP 1.1 AND SOAP 1.2

Ideally, one wouldn't need to dive too much into the SOAP protocol—after all, the whole point of SOAP is to make access to web services look like programming objects on your own desktop or server. But libraries and services do seem to have crucial dependences on the actual version of SOAP being used (for example).

SOAP has become a W3C Recommendation. The latest version of SOAP is 1.2:

`http://www.w3.org/TR/soap12-part1/`

Earlier versions of SOAP are still very much in use—maybe even more so than version 1.2. Version 1.1 is specified here:

`http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`

Here are a few salient differences between the two specifications (the differences are described in detail at `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/#L4697`):

* Different namespaces for the SOAP envelope (`http://www.w3.org/2003/05/soap-envelope` for version 1.2 and `http://schemas.xmlsoap.org/soap/envelope/` for version 1.1)—a practical heuristic to help spot which version of SOAP you are dealing with.
* Different use of the `SOAPAction` parameter for the SOAP HTTP binding. In SOAP 1.2, a `SOAPAction` HTTP request header is no longer used.
* The use of an HTTP response header of `Content-Type` "application/soap+xml" to identify SOAP 1.2.

I point out these differences because libraries and toolsets support different versions of SOAP.

You can now replicate this call with `curl`:

```
curl -v -X POST -H "SOAPAction: http://rpc.geocoder.us/Geo/Coder/US#geocode"
~CCC
--data-binary "<SOAP-ENV:Envelope xmlns:SOAP-
ENV='http://schemas.xmlsoap.org/soap/ ~CCC
envelope/'><SOAP-ENV:Header/><SOAP-ENV:Body><oxy:geocode xmlns:oxy= ~CCC
'http://rpc.geocoder.us/Geo/Coder/US/' SOAP-ENV:encodingStyle='http://schemas.
~CCC
xmlsoap.org/soap/encoding/'><location>2855 Telegraph Ave, Berkeley,
CA</location> ~CCC
</oxy:geocode></SOAP-ENV:Body></SOAP-ENV:Envelope>" ~CCC
http://rpc.geocoder.us/service/soap/
```

Note that you need to know the `SOAPaction` header and URL of the SOAP call *only* if you are trying to understand all the details of the HTTP request and response. oXygen was just being helpful in pointing out those parameters. They, however, were not needed to fill out an address or interpret the latitude or longitude contained in the response.

Note If you're wondering why I'm not using Flickr for my concrete example, Flickr does not offer a WSDL document even though it does present a SOAP interface. I'll return to discussing Flickr in the later section called "The Flickr API via SOAP."

Even without access to oXygen or the Eclipse Web Services Explorer, you can use Tomi Vanek's WSDL XSLT-based viewer (`http://tomi.vanek.sk/index.php?page=wsdl-viewer`) to make sense of a WSDL document. For example, take a look at the results for the geocoder.us WSDL document:

```
http://www.w3.org/2000/06/webdata/xslt?xslfile=http://tomi.vanek.sk/xml/ ~CCC
wsdl-
viewer.xsl&amp;xmlfile=http://geocoder.us/dist/eg/clients/GeoCoderPHP.wsdl&amp;
~CCC
transform=Submit
```

Using Python's SOAPpy

Let's take a look how to use the geocoder.us WSDL using the `SOAPpy` library in Python.

Note You can download `SOAPpy` from `http://pywebsvcs.sourceforge.net/`. Mark Pilgrim's *Dive Into Python* provides a tutorial for `SOAPpy` at `http://www.diveintopython.org/soap_web_services/index.html`.

The following piece of Python code shows the process of creating a WSDL proxy, asking for the methods (or operations) that are defined in the WSDL document, and then calling the `geocode` method and parsing the results:

```
from SOAPpy import WSDL

wsdl_url = r'http://geocoder.us/dist/eg/clients/GeoCoderPHP.wsdl'
server = WSDL.Proxy(wsdl_url)

# let's see what operations are supported
server.show_methods()

# geocode the Apress address
address = "2855 Telegraph Ave, Berkeley, CA"
result = server.geocode(location=address)
print "latitude and longitude: %s, %s" % (result[0]['lat'], result[0]['long'])
```

This produces the following output (edited for clarity):

```
Method Name: geocode_intersection
   In #0: intersection  ((u'http://www.w3.org/2001/XMLSchema', u'string'))
   Out #0: results  ((u'http://rpc.geocoder.us/Geo/Coder/US/',
u'ArrayOfGeocoderIntersectionResult'))

Method Name: geocode_address
   In #0: address  ((u'http://www.w3.org/2001/XMLSchema', u'string'))
   Out #0: results  ((u'http://rpc.geocoder.us/Geo/Coder/US/',
u'ArrayOfGeocoderAddressResult'))

Method Name: geocode
   In #0: location  ((u'http://www.w3.org/2001/XMLSchema', u'string'))
   Out #0: results  ((u'http://rpc.geocoder.us/Geo/Coder/US/',
u'ArrayOfGeocoderResult'))

latitude and longitude: 37.858276, -122.26007
```

Notice the reference to XML schema types in describing the `location` parameter for geocode. The type definitions come, as one expects, from the WSDL document.

The concision of this code shows WSDL and SOAP in good light.

USING SOAP FROM PHP

There are several choices of libraries for consuming SOAP in PHP:

* `NuSOAP` (`http://sourceforge.net/projects/nusoap/`)
* `PEAR::SOAP` package (`http://pear.php.net/package/SOAP`)
* The built-in SOAP library in PHP 5 (`http://us2.php.net/soap`), which is available if PHP is installed with the enable-soap flag

In this book, I use the `PEAR::SOAP` library.

Using PHP PEAR::SOAP

Let's do the straight-ahead `PHP PEAR::SOAP` invocation of `geocode.us`. You'll the same pattern of loading the WSDL document using a SOAP/WSDL library, packaging up a named parameter (`location`) in the request, and then parsing the results.

```
<?php
# example using PEAR::SOAP + Geocoder SOAP search
require 'SOAP/Client.php';

# let's look up Apress
$address = '2855 Telegraph Avenue, Berkeley, CA 94705'; // your Google search
terms

$wsdl_url = "http://geocoder.us/dist/eg/clients/GeoCoderPHP.wsdl";

# true to indicate that it is a WSDL url.
$soap = new SOAP_Client($wsdl_url,true);
```

```
$params = array(
    'location'=>$address
  );

$results = $soap->call('geocode', $params);

# include some fault handling code
if(PEAR::isError($results)) {
    $fault = $results->getFault();
    print "Error number " . $fault->faultcode . " occurred\n";
    print "     " . $fault->faultstring . "\n";
} else {
    print "The latitude and longitude for address is: {$results[0]->lat},
{$results[0]->long}";
}
?>
```

Note I have not been able to figure out how to use `PEAR::SOAP` to tell me the operations that are available for a given WSDL file.

Amazon ECS

Now that you have studied the `geocoder.us` service, which has three SOAP methods, each with a single input parameter, let's turn to a more complicated example, the Amazon E-Commerce Service (ECS):

`http://www.amazon.com/E-Commerce-Service-AWS-home-page/b?ie=UTF8&node=12738641`

See the "Setting Up an Amazon ECS Account" sidebar to learn about how to set up an Amazon ECS account.

SETTING UP AN AMAZON ECS ACCOUNT

To use the service, you need to obtain keys by registering an account (like with Flickr):

`http://www.amazon.com/gp/aws/registration/registration-form.html`

If you already have an account, you can find your keys again:

```
http://aws-portal.amazon.com/gp/aws/developer/account/index.html/?ie=UTF8&
action=access-key
```

You get an access key ID and a secret access key to identify yourself and your agents to AWS. You can also use an X.509 certificate, which the Amazon interface can generate for you.

Although I focus here on the SOAP interface, ECS also has a REST interface. The WSDL for AWS-ECS is found at

`http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl?`

Using one of the SOAP/WSDL toolkits I presented in the previous section (for example, oXygen, the Eclipse Web Services Explorer, or Vanek's WSDL viewer), you

can easily determine the 20 operations that are currently defined by the WSDL document. Here I show you how to use the `ItemSearch` operation.

If you use oXygen to formulate a template for a SOAP request, you'll get the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <ItemSearch
      xmlns="http://webservices.amazon.com/AWSECommerceService/2007-07-16">
      <AWSAccessKeyId>STRING</AWSAccessKeyId>
      [5 tags]
      <Shared>
         [40 tags]
      </Shared>
      <Request>
         [40 tags]
      </Request>
    </ItemSearch>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let's say you wanted to look for books with the keyword `flower`. To create the proper request, you'll need to figure out which of the many tags you must keep and how to fill out the values that you need to fill out. Through reading the documentation for `ItemSearch` (`http://docs.amazonwebservices.com/AWSECommerceService/2007-07-16/DG/ItemSearch.html`) and trial and error, you can boil down the request template to the following:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <ItemSearch
      xmlns="http://webservices.amazon.com/AWSECommerceService/2007-07-16">
      <AWSAccessKeyId>STRING</AWSAccessKeyId>
      <Request>
        <Keywords>STRING</Keywords>
        <SearchIndex>STRING</SearchIndex>
      </Request>
    </ItemSearch>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You can pull together a full request by filling out your Amazon key and entering `flower` and `Books` for the `<Keywords>` and `<SearchIndex>` into a `curl` invocation:

```
curl -H "SOAPAction: http://soap.amazon.com" -d "<?xml version='1.0' ~CCC
encoding='UTF-8'?><SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/
~CCC
soap/envelope/'><SOAP-ENV:Header/><SOAP-ENV:Body><ItemSearch ~CCC
xmlns='http://webservices.amazon.com/AWSECommerceService/2007-07-16'> ~CCC
<AWSAccessKeyId>[AMAZON-
KEY]</AWSAccessKeyId><Request><Keywords>flower</Keywords> ~CCC
```

```
<SearchIndex>Books</SearchIndex></Request></ItemSearch></SOAP-ENV:Body> ~CCC
</SOAP-ENV:Envelope>"
http://soap.amazon.com/onca/soap?Service=AWSECommerceService
```

to which you get something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ItemSearchResponse
      xmlns="http://webservices.amazon.com/AWSECommerceService/2007-07-16">
      <OperationRequest>
[....]
      </OperationRequest>
      <Items>
        <Request>
          <IsValid>True</IsValid>
          <ItemSearchRequest>
            <Keywords>flower</Keywords>
            <SearchIndex>Books</SearchIndex>
          </ItemSearchRequest>
        </Request>
        <TotalResults>34489</TotalResults>
        <TotalPages>3449</TotalPages>
        <Item>
          <ASIN>0812968069</ASIN>
       <DetailPageURL>
          http://www.amazon.com/gp/redirect.html%3FASIN=0812968069%26 ~CCC
tag=ws%26lcode=sp1%26cID=2025%26ccmID=165953%26location=/o/ASIN/0812968069%253F
~CCC
SubscriptionId=0Z8Z8FYGP01Q00KF5802</DetailPageURL>
          <ItemAttributes>
            <Author>Lisa See</Author>
            <Manufacturer>Random House Trade Paperbacks</Manufacturer>
            <ProductGroup>Book</ProductGroup>
            <Title>Snow Flower and the Secret Fan: A Novel</Title>
          </ItemAttributes>
        </Item>
[...]
      </Items>
    </ItemSearchResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice what makes this example more complicated than `geocoder.us`:

* There are many more operations.
* There are many more parameters, and it's not obvious what is mandatory without reading the documentation and experimenting.

* The XML in the request and response involve complex types. Notice that `<Keywords>` and `<SearchIndex>` are wrapped within `<Request>`. This representation means you have to understand how to get your favorite SOAP library to package up the request and handle the response.

Using the Python `SOAPpy` library, you perform the same SOAP call with the following:

```
# amazon search using WSDL
KEY = "[AMAZON-KEY]"

from SOAPpy import WSDL

class amazon_ecs(object):
    def __init__(self, key):
        AMAZON_WSDL =
"http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl?"
        self.key = key
        self.server = WSDL.Proxy(AMAZON_WSDL)
    def ItemSearch(self,Keywords,SearchIndex):
        return self.server.ItemSearch(AWSAccessKeyId=self.key,Request= ~CCC
{'Keywords':Keywords,'SearchIndex':SearchIndex})

if __name__ == "__main__":
    aws = amazon_ecs(KEY)
    results= aws.ItemSearch('flower','Books')
    print results.Items.TotalPages,  results.Items.TotalResults
    for item in results.Items.Item:
        print item.ASIN, item.DetailPageURL, item.ItemAttributes.Author
```

Notice in particular how to represent the nested parameters in this:

```
self.server.ItemSearch(AWSAccessKeyId=self.key,Request= ~CCC
{'Keywords':Keywords,'SearchIndex':SearchIndex})
```

Also notice how to read off the nested elements in the XML response:

```
print results.Items.TotalPages,  results.Items.TotalResults
for item in results.Items.Item:
    print item.ASIN, item.DetailPageURL, item.ItemAttributes.Author
```

When you look at this Python code and my description of how to use oXygen to interface with Amazon ECS via WSDL and SOAP, you might think to yourself that doing so doesn't look that hard. The combination of WSDL and SOAP does indeed bring some undeniable conveniences such as the automated discovery of what methods are available to you as a programmer. However, my experience of SOAP and WSDL is that they are still a long way from plug-and-go technology—at least in the world of scripting languages such as PHP and Python. It took me a great amount of trial and error, reverse engineering, reading source code, and hunting around to even get to the point of distilling for you the various examples of how to use SOAP and WSDL you see here. I would have wanted to reduce using SOAP and WSDL to full-proof recipes that hid from you what was happening underneath.

For instance—returning to the example—I was not able to able to craft a satisfactory working example of using `PEAR::SOAP` to call `ItemSearch`. Some of the issues I struggled

with included how to pass in parameters with complex types to a SOAP call, how to parse the results, and how to debug the entire process. I'd be willing to bet that there is in fact a way to make this call work with `PEAR::SOAP` or in some other PHP toolkit. However, if I had wanted to call this SOAP service only for a mashup, I would likely have given up even earlier on figuring out how to make it work.

Note It might be true that if you use Java or .NET, programming environments for which there is deep support for SOAP and WSDL, you might have an easier time using this technology. Don't let me discourage you from trying those tools. I hope to find out for myself whether libraries such as Axis from the Apache Project (`http://ws.apache.org/axis/java/index.html`) or the WSDL functionality in .NET do indeed make my life as a SOAP developer easier.

The Flickr API via SOAP

The Flickr SOAP request and response formats are documented here:

```
http://www.flickr.com/services/api/request.soap.html
http://www.flickr.com/services/api/response.soap.html
```

The first thing to notice about the Flickr SOAP interface is that *Flickr provides no WSDL document to tell us how to use it*. Hence, if you want to use Flickr SOAP, you need to figure out how call it directly yourself. But why bother? Flickr has a wonderfully supported REST interface that you already know how to use. If you go down the road of using the SOAP interface, you'll have to deal with many challenges, some of which I have already discussed.

Learning About Specific Web APIs

In the previous section, I showed you how to call web APIs that use XML-RPC and SOAP. That still leaves many APIs that fall under the name of REST—ones that look a lot like the Flickr REST interface. These APIs take some things we are familiar with from web browsers, such as going to a specific URL to get back some results and submitting HTML forms to make a query, but they have one important difference: instead of sending mostly HTML (which is directed at human consumption), you send primarily XML, a *lingua franca* of computer data exchange. For that reason, you should remind yourself of what you've learned from the previous chapters as you embark on a study of other REST APIs.

In the following sections, I'll make sense of the world of web APIs, covering how to find out what APIs are available and then how to use a particular API. I'll start my discussion by introducing perhaps the single most useful website about web APIs: Programmableweb.com. There's a lot of information that is both readily apparent and waiting to be discovered in this treasure trove of data.

Note Other directories of web services that are worth exploring are `http://www.xmethods.net/`, which lists publicly available SOAP services, and `http://strikeiron.com/`, a provider of commercial

web services that you can try for free.

Programmableweb.com

Programmableweb.com, started and maintained by Jon Musser, is an excellent resource for learning about what APIs are available, the basic parameters for the APIs, and the mashups that use any given API. Some noteworthy sections are as follows:

* `http://www.programmableweb.com/apis` is the "API dashboard" that lists the latest APIs to be registered and the most popular APIs being used in mashups.
* `http://www.programmableweb.com/apilist/bycat` lists APIs by categories. Understanding the various categories that have emerged is helpful for understanding for which fields of endeavor people are making APIs.
* `http://www.programmableweb.com/apilist/bymashups` lists APIs by how many times they are used in the mashups registered at Programmableweb.com.

I highly recommend a close and periodic study of Programmableweb.com for anybody wanting to learn about web APIs and mashups. Let me show some of the things you can learn from the website, based both on what is directly presented on the site and on data that John Musser has sent me. Although web APIs and corresponding mashups are rapidly changing, the data (and derived results), accurate for August 11, 2007, demonstrates some trends that I think will hold for a while yet.

The first thing to know is that of the 494 web APIs in the database, we get the distribution of number of APIs by protocol supported shown in Table 7-2. Note that some APIs are multiply represented.

Table 7-2. Number of APIs vs. Protocol in Programmableweb.com

Protocol	Number of APIs with Support
REST	255
SOAP	131
XML-RPC	19
JavaScript	30
Other	16

Some other observations drawn from the database are as follows:

* Ninety-three APIs have WSDL files associated with them.
* Of the 131 APIs that support SOAP, 42 also support REST—leaving 89 that support SOAP but not REST. Eighty-eight APIs support only SOAP.
* XML-RPC is the only choice for nine APIs.
* JavaScript is listed as the exclusive protocol for 25 APIs.

The following are my conclusions based on this data:

* REST is the dominant mode of presenting web APIs, but a significant number of APIs exist where your only choice is SOAP.
* There are a relatively small number of APIs listing XML-RPC as the only choice of protocol.

It's therefore useful to know how to use SOAP and XML-RPC, even if they are not your first choice.

Note A large number of APIs list JavaScript as a protocol. I'll cover such APIs in the next chapter.

Table 7-3 lists the top 20 APIs on Programmableweb.com by mashup count and also lists the type of protocols supported by the API.

Table 7-3. Top 21 APIs by Mashup Count

API Name	Number of Mashups	Protocols Support
Google Maps	1110	JavaScript
Flickr	243	REST, SOAP, XML-RPC
Amazon E-Commerce Service	174	REST, SOAP
YouTube	149	REST, XML-RPC
Microsoft Virtual Earth	97	JavaScript
Yahoo! Maps	95	REST, JavaScript, Flash
411Sync	89	RSS input over HTTP, SOAP
eBay	89	SOAP, REST
del.icio.us	83	REST
Google Search	79	SOAP
Yahoo! Search	78	REST
Yahoo! Geocoding	66	REST
Technorati	40	REST
Yahoo! Image Search	31	REST
Yahoo! Local Search	30	REST
Last.fm	28	REST
Google home page	27	JavaScript
Google Ajax Search	24	JavaScript
Upcoming.org	21	REST
Windows Live Search	21	SOAP
Feedburner	21	REST

What can you do with this information? To learn about popular APIs, one approach would be to go down the list systematically to figure out how each works. Indeed, through the rest of the book, I'll cover many of the APIs in the table. The Flickr API is the second most used API in mashups and is a main subject throughout this book. I'll cover the JavaScript-based maps (first and foremost Google Maps but also Yahoo! Maps and Virtual Earth) first in Chapter 8 and then in depth in Chapter 13. I'll cover the Yahoo! Geocoding API extensively also in Chapter 13. I'll cover various search APIs (Google Search, Yahoo! Search, Yahoo! Image Search, and Windows Live Search) in Chapter 19. Finally, I'll cover the del.icio.us API in Chapter 14 on social bookmarking. Indeed, the fact that I cover many APIs clustered by subject matter indicates that it is a natural way to think about APIs.

YouTube

YouTube is probably the most famous video-sharing site on the Web—and it also uses tagging as one way of organizing content. The YouTube API is documented at `http://www.youtube.com/dev`.

The YouTube API supports both a REST interface and an XML-RPC interface. The examples I give in this section use the REST interface. You can find a list of methods at `http://www.youtube.com/dev_docs`.

To use the API, you need to set up your own development profile; see `http://www.youtube.com/my_profile_dev`.

An interesting feature of the registration process is that you enter your own secret (instead of having one set by YouTube). When you submit your profile information, you then get a "developer ID." The following are some sample calls. To get the user profile for a user (for example, `rdhyee`), you do an HTTP `GET` on the following:

```
http://www.youtube.com/api2_rest?method=youtube.users.get_profile& ~CCC
dev_id={youtube-key}&user=rdhyee
```

YouTube will send you a response something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<ut_response status="ok">
  <user_profile>
    <first_name>Raymond</first_name>
    <last_name/>
    <about_me/>
    <age>40</age>
    <video_upload_count>2</video_upload_count>
    <video_watch_count>102</video_watch_count>
    [....]
  </user_profile>
</ut_response>
```

To get the list of `rdhyee`'s favorite videos, use this:

```
http://www.youtube.com/api2_rest?method=youtube.users.list_favorite_videos& ~CCC
dev_id={youtube-key}&user=rdhyee
```

To get details of a video with an ID of `XHnE4umovw4`, use this:

```
http://www.youtube.com/api2_rest?method=youtube.videos.get_details& ~CCC
```

```
dev_id={youtube-key} &video_id=XHnE4umovw4
```

To get videos for the tag `HolidayWeekend`, use this:

```
http://www.youtube.com/api2_rest?method=youtube.videos.list_by_tag& ~CCC
dev_id={youtube-key}&tag=HolidayWeekend&page=1&per_page=100
```

There's more to the API, but you can get a feel for how it works through these examples.

Caution Expect the YouTube API to evolve into something more like the rest of Google's APIs: `http://code.google.com/apis/youtube/overview.html`.

GData and the Blogger API

The Atom Publishing Protocol (APP), a companion to the Atom Syndication Format (Atom 1.0) described in Chapter 2, represents the next generation of the blogging APIs. APP is currently a draft IETF proposal:

```
http://tools.ietf.org/wg/atompub/draft-ietf-atompub-protocol/
```

which is linked from here:

```
http://tools.ietf.org/wg/atompub/
```

One of APP's biggest supporters thus far has been Google, which has implemented GData, which is based on Atom 1.0 and RSS 2.0 feeds, combined with APP. GData, which incorporates Google-specific extensions, is the foundation of the APIs for many of its services, including Google Base, Blogger, Google Calendar, Google Code Search, and Google Notebook:

```
http://code.google.com/apis/gdata/index.html
```

The API for Blogger is documented here:

```
http://code.google.com/apis/blogger/developers_guide_protocol.html
```

In the following sections, you'll learn the basics of the API for Blogger as a way of understanding GData and APP in general.

Obtaining an Authorization Token

The first thing you need to have is a Google account to use Blogger. If you don't have one, go to the following location to create one:

```
https://www.google.com/accounts/NewAccount
```

Next, with a Google account, you obtain an authorization token. One way to do so is to follow the procedure for ClientLogin (one of two Google authorization mechanisms) detailed here:

```
http://code.google.com/apis/blogger/developers_guide_protocol.html#client_login
```

Specifically, you make an HTTP `POST` request to the following location:

```
https://www.google.com/accounts/ClientLogin
```

The body must contain the following parameters (using the `application/x-www-form-urlencoded` content type):

`Email`: Your Google email (for example, `raymond.yee@gmail.com`)

`Password`: Your Google password

`source`: A string of the form *companyName-applicationName-versionID* to identify your program (for example, `mashupguide.net-Chap7-v1`)

`service`: The name of the Google service, which in this case is `blogger`

Using the example parameters listed here, you can package up the authorization request as the following `curl` invocation:

```
curl -v -X POST -d "Passwd={passwd}&source=mashupguide.net-Chap7-v1& ~CCC
Email=raymond.yee%40gmail.com&service=blogger" ~CCC
https://www.google.com/accounts/ClientLogin
```

If this call succeeds, you will get in the body of the response an `Auth` token (of the form `Auth=[AUTH-TOKEN])`. Retain the `AUTH-TOKEN` for your next calls.

Figuring Out Your Blogger User ID

If you don't have a blog on Blogger.com, create one here:

```
http://www.blogger.com/create-blog.g
```

Now figure out your Blogger user ID by going to and noting the URL associated with the View link (beside the Edit Profile link):

```
http://www.blogger.com/home
```

Your View link will be of the following form:

```
http://www.blogger.com/profile/{userID}
```

For example, my blog profile is as follows:

```
http://www.blogger.com/profile/13847941708302188690
```

So, my user ID is `13847941708302188690`.

Getting a List of a User's Blogs and a Blog's Posts

Note that Blogger lists user blogs in a user's profile:

```
http://www.blogger.com/profile/{userID}
```

From an API point of view, you can retrieve an Atom feed of a user's blog here:

```
http://www.blogger.com/feeds/{userID}/blogs
```

That the list of blogs is an Atom feed and not some custom-purpose XML (such as that coming out of the Flickr API) is useful. You can look at the feed of your blogs to pull out the blog ID for one of your blogs. For instance, one of my blogs is entitled "Hypotyposis Redux" and is listed in the feed with the following tag:

```
<id>tag:blogger.com,1999:user-354598769533.blog-5586336</id>
```

From this you can determine its blog ID of `5586336`. With this `blogID`, you can now send HTTP `GET` requests to retrieve an Atom feed of posts here:

```
http://www.blogger.com/feeds/{blogID}/posts/default
```

For example:

```
http://www.blogger.com/feeds/5586336/posts/default
```

Creating a New Blog Entry

Let's now look at how to create a new post. A central design idea behind the Atom Publishing Protocol and hence its derivatives—GData generally and the Blogger API specifically—is the notion of a *uniform interface* based on the standard HTTP methods. At this point, it's useful to refer to the "Methods Definition" of the HTTP 1.1 specification (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`), specifically the definition for `POST`:

> *The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.*

You may be surprised to read this definition of the `POST` method, considering, for instance, how POST is used for every single SOAP 1.1 call bound to HTTP—whether the call is for retrieving a simple piece of information, creating a new resource, or deleting it.

Let's see how to create an HTTP `POST` request to create a new blog entry and confirm how the process follows the earlier definition:

1. Create a blog entry formatted as an `<entry>` Atom element, something like this:

```
<entry xmlns='http://www.w3.org/2005/Atom'>
  <title type='text'>Using Blogger to demo APP</title>
  <content type='xhtml'>
    <div xmlns='http://www.w3.org/1999/xhtml'>
      <p>This message is being created from invoking the blogger
APP-based API.</p>
      <p>This process is documented at <a href="http://code.google.com/
apis/blogger/developers_guide_protocol.html#CreatingPosts">Blogger Data
API -- Creating Posts</a></p>
    </div>
  </content>
  <author>
    <name>Raymond Yee</name>
    <email>raymond.yee@gmail.com</email>
  </author>
</entry>
```

2. Save the file, say with the filename `blogger.message.1.xml`.

3. Issue the following `curl` invocation to `POST` the contents of your file to `http://www.blogger.com/feeds/{blogID}/posts/default`—which is the feed of all your entries for the blog—to create a new entry for the blog (a "new subordinate of the resource identified by the Request-URI").

```
curl -X POST -v --data-binary "@blogger.message.1.xml" -H "Content-Type: ~CCC
application/atom+xml " -H "Authorization: GoogleLogin  auth=[AUTH-TOKEN]" ~CCC
http://www.blogger.com/feeds/{blogID}/posts/default
```

4. If things go fine, you'll get an HTTP 201 Created code and an `<entry>` holding the new post. This `<entry>` tells you things such as the post ID of your new entry. The response will look like the following:

```
HTTP/1.1 201 Created
Content-Type: application/atom+xml; charset=UTF-8
Cache-Control: max-age=0, must-revalidate, private
Location: http://www.blogger.com/feeds/5586336/posts/default/409227349217351
7704
Content-Location: http://www.blogger.com/feeds/5586336/posts/default/4092273
492173517704
Transfer-Encoding: chunked
Date: Sat, 25 Aug 2007 14:38:49 GMT
Server: GFE/1.3
<?xml version='1.0' encoding='UTF-8'?><?xml-stylesheet href="http://www.b
logger.com/styles/atom.css" type="text/css"?><entryxmlns='http://www.w3.o
rg/2005/Atom'><id>tag:blogger.com,1999:blog-5586336.post-4092273492173517
704</id><published>2007-08-25T07:38:00.001-07:00</published><updated>2007
-08-25T07:38:49.607-07:00</updated><title type='text'>Using Blogger to de
mo APP</title><content type='html'>&lt;br /&gt;    &lt;div xmlns='http://
www.w3.org/1999/xhtml'&gt;&lt;br /&gt;
   &lt;p&gt;This message is being created from invoking the blogger APP-
based API.&lt;/p&gt;&lt;br /&gt;      &lt;p&gt;This process is documented
at &lt;a href='http://code.google.com/apis/blogger/developers_guide_proto
col.html#CreatingPosts'&gt;Blogger Data API -- Creating Posts&lt;/a&gt;&l
t;/p&gt;&lt;br /&gt;    &lt;/div&gt;&lt;br /&gt;  </content><link rel='al
ternate' type='text/html' href='http://hypotyposis.blogspot.com/2007_08_0
1_archive.html#4092273492173517704' title='Using Blogger to demo APP'/><l
ink rel='replies' type='applicati* Connection #0to host www.blogger.com l
eft intact* Closing connection #0
```

5. In this example, the POST request created a new blog entry with a post ID of `40922734921735177`.

Updating the Blog Entry

You can update your blog entry using an HTTP `PUT` request, in accordance to the HTTP 1.1 specification that states the following:

> *The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.*

Let's package this request for `curl` after first creating an updated message in the `blogger.message.2.xml` file:

```
curl -X PUT -v --data-binary "@blogger.message.2.xml" -H "Content-Type: ~CCC
application/atom+xml " -H "Authorization: GoogleLogin  auth=[AUTH-TOKEN]" ~CCC
http://www.blogger.com/feeds/{blogID}/posts/default/{postID}
```

If you are unfamiliar with using the HTTP `PUT` method, you're hardly alone. As mentioned in Chapter 6, there is little support for it. (Remember, for instance, that the HTML forms define the `GET` and `POST` methods.) Recognizing that `PUT` might not be supported by the client doing the entry update (or that firewalls might block `PUT` requests), you can tunnel the `PUT` request through a `POST` request like so:

```
curl -X POST -v --data-binary "@blogger.message.2.xml" -H "X-HTTP-Method-
Override: ~CCC
PUT"  -H "Content-Type: application/atom+xml " -H "Authorization: GoogleLogin
~CCC
auth=[AUTH-TOKEN]"
http://www.blogger.com/feeds/{blogID}/posts/default/{postID}
```

Deleting a Blog Entry

You can use the HTTP `DELETE` method to delete an entry but send that request to the URL of the entry itself. As a `curl` invocation, the request looks like this:

```
curl -X DELETE -v -H "Content-Type: application/atom+xml " -H "Authorization:
~CCC
GoogleLogin  auth=[AUTH-TOKEN]" ~CCC
http://www.blogger.com/feeds/{blogID}/posts/default/{postID}
```

As with updating a blog entry, you can tunnel a `DELETE` request through an HTTP `POST` request using an "X-HTTP-Method-Override: DELETE" request header.

Using the Blogger API As a Uniform Interface Based on HTTP Methods

Now that you have seen how to use the Blogger API to retrieve feeds of blogs and blog entries, create new blog entries, update an entry, and delete an entry, you should notice how all these actions are performed while hewing closely to HTTP methods as they are actually defined in the HTTP specification. This pattern of using the HTTP methods as the fundamental methods of the API, in fact, repeats itself in all the APIs that are based on the Atom Publishing Protocol and therefore GData. Thus, the uniform interface of GData is the full collection of standard HTTP methods.

Summary

In this chapter, I discussed how to consume web APIs that use the XML-RPC and SOAP/WSDL protocols. Although these protocols, especially SOAP and WSDL, are geared toward simplifying the process for making calls to web services, they sometimes are fragile in practice. Consequently, if you use them, you should learn how to debug them with the techniques I showed you in this chapter.

With techniques to work with REST, XML-RPC, and SOAP web APIs in hand, you can then start moving beyond Flickr to look at a wide range of APIs. I showed you how to use Programmableweb.com to learn about those APIs and to draw some broad conclusions about APIs, the protocols they use, which ones are popular, and which subject matter they cover. I concluded this chapter with a study of the YouTube API (as an example of a simple REST API other than Flickr) and the Blogger API (as an instance of a uniform interface intimately tied to the HTTP methods). In the next chapter, you'll study JavaScript-based APIs and look at how to consume web APIs in the browser.