PART 2

Remixing a Single Web Application Using Its API

In Part I, we looked at how to recombine information without resorting to formal programming techniques. There is a lot that can be done by carefully examining various web applications from the perspective of an end user looking for integrative opportunities. We studied, for instance, how you can recombine information through manipulating URLs, introducing tags, and connecting feeds from one application to another.

In the rest of the book, we'll take on the programmer's perspective. In the first two chapters in this part, for example, we turn to learning about how to use web services, starting from the concrete example of Flickr (Chapter 6) and then contrasting and comparing Flickr to other examples (Chapter 7). In Chapter 8, we turn to Ajax-based and JavaScript-based widgets, building upon what we learn in Chapter 6 and Chapter 7.

CHAPTER 6

Learning Web Services APIs Through Flickr

Flickr is an excellent playground for learning XML web services. Among other reasons, Flickr offers clear documentation, an instructive API Explorer that lets you try the API through a browser, and lots of prior art to study in terms of remixes and mashups. Hundreds of third-party apps are using the Flickr API.

As I discussed in previous chapters (especially Chapter 2), application programming interfaces (APIs) are the preferred way of programming a website and accessing its data and services, although not all websites have APIs. We looked at a wide range of things you can do without doing much programming, which in many cases means not resorting to the API. But now we turn to using APIs. Don't forget what you learned while looking at end-user functionality, because you will need that knowledge in applying APIs.

By the end of this chapter, you will see that the Flickr API is an extensive API that can do many things using many options. The heart of the API is simple, though. I'll start

this chapter by presenting and analyzing perhaps the simplest immediately useful thing you can do with the Flickr API. I'll walk you through that example in depth to show you conceptually how to use the search API and how to interpret the results you get. After walking you through how to make that specific request, I'll outline the various ways in which the example can be generalized.

After an overview of the policy and terms of service surrounding the API, I'll show you how to make sense of the Flickr documentation and how to use the excellent Flickr API Explorer to study the API. I'll revisit in depth the mechanics of making a basic call of a Flickr API method, using it as an opportunity to provide a tutorial on two fundamental techniques: processing HTTP requests and parsing XML. I then demonstrate how to knit those two techniques to create a simple HTML interface based on the photo search API.

With an understanding of how to exercise a single method in hand, you'll then look at all the API methods in Flickr. I'll demonstrate how to use the reflection methods in the Flickr API to tell you about the API itself. I'll next explain the ways in which you can choose alternative formats for the requests and responses in the API, laying the foundation for a discussion of REST and SOAP that I'll revisit in the next chapter.

By that point in the chapter, you will have done almost everything you can do with authorization, the trickiest part of the API. Flickr authorization can confusing if you do not understand the motivation behind the steps in the authorization dance. I'll explain the mechanics of the authorization scheme in terms of what Flickr must be accomplishing in authorization—and how all the technical pieces fit together to accomplish those design goals. It's an involved story but one that might elucidate for you other authentication schemes out there with similar design constraints. After the narrative, I've included some PHP code that implements the ideas.

For practical use of the Flickr API to make mashups, you probably do not want to work so close to the API itself but instead use API kits or third-party language-specific wrappers. Therefore, I'll survey briefly three of the PHP API kits for Flickr. I'll conclude this chapter by pointing out some of the limitations of the Flickr API with respect to what's possible with the Flickr user interface.

An Introduction to the Flickr API

It's useful to start with a simple yet illustrative example of the Flickr API before diving into the complexities that can easily obscure the simple idea at the heart of the API. The API is designed for you as a programmer to send *requests* to the API and get *responses* that are easy for you to decipher with your program. In earlier chapters, especially Chapter 2, you learned about you can use the URL language of Flickr to access resources from Flickr. However, for a computer program to use that information, it would have to screen-scrape the information. Screen-scraping is a fragile and cumbersome process. The Flickr API sets a framework for both making requests and getting responses that are well defined, stable, and convenient for computer programs.

Before you proceed any further, sign up for a Flickr API key so that you can follow along with this example (see "Obtaining a Flickr API Key").

OBTAINING A FLICKR API KEY

You need a key to use the Flickr API. A *key* is a string of numbers and letters that identifies you as the source of an API request. That is, when you make a call of the API, you typically need to pass in your key (or some other parameter derived from your key). You get a key through registering your application with Flickr:

```
http://www.flickr.com/services/api/keys/apply/
```

Get your own API key to try the exercises in this chapter and the following chapters. You can see the list of your current keys here:

```
http://www.flickr.com/services/api/keys/
```

In the next chapter, you will see that keys are a common mechanism used in other application APIs. Through keys, the API provider knows something about the identity of an API user (typically at least the API key holder's e-mail address if nothing else) and monitors the manner in which a user is accessing the API (such as the rate and volume of calls and the specific requests made). Through such tracking, the API provider might also choose to enforce the terms of use for the API—from contacting the user by e-mail to shutting down access by that key...to suing the user in extreme cases!

Once you have your key, let's make the simplest possible call to the Flickr API. Drop the following URL in your browser:

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

where `api-key` is your Flickr API key. For this request, there are two parameters: `method`, which indicates the part of the API to access, and `api_key`, which identifies the party making the API request. Flickr produces the following response corresponding to your request:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<method>flickr.test.echo</method>
<api_key>[API-KEY]</api_key>
</rsp>
```

Note that the entity body of the response is an XML document containing your key.

Let's now consider a slightly more complicated call to the Flickr API that returns something more interesting. Let's ask Flickr for photos with a given tag. You learned in Chapter 2 that the corresponding URL in the Flickr UI for pictures corresponding to a given tag (say, the tag `puppy`) is as follows:

```
http://www.flickr.com/photos/tags/{tag}/
```

For example:

```
http://www.flickr.com/photos/tags/puppy/
```

The corresponding way to get from the Flickr API to the most recently uploaded public photos for a `tag` is like so:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags={tag}&per_page={per_page}
```

When you substitute your API key, set `tag` to `puppy`, and set `per_page` to `3` to issue the following call:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags=puppy&per_page=3
```

you will get something similar to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page="1" pages="96293" perpage="3" total="288877">
  <photo id="1153699093" owner="7841384@N07" secret="d1fba451c9" server="1023"
         farm="2" title="willy after bath and haircut" ispublic="1" isfriend="0"
         isfamily="0" />
  <photo id="1154506492" owner="7841384@N07" secret="881ff7c4bc" server="1058"
         farm="2" title="rocky with broken leg" ispublic="1" isfriend="0"
         isfamily="0" />
  <photo id="1153588011" owner="90877382@N00" secret="8a7a559e68" server="1288"
         farm="2" title="DSC 6503" ispublic="1" isfriend="0" isfamily="0" />
</photos>
</rsp>
```

What happens in this Flickr API call? In the request, you ask for the three most recently uploaded public photos with the tag `puppy` via the `flickr.photos.search` method. You get back an XML document in the body of the response. I'll show you later in the chapter the mechanics of how to parse the XML document in languages such as PHP. For the moment, notice the information you are getting in the XML response:

* Within the `rsp` `root` element, you find a `photos` element containing three child `photo` elements.
* Attributes in the `photos` element tell you a number of facts about the photo: the `total` attribute is the number of public photos tagged with `puppy` (288,877), the `perpage` attribute is the number of photo elements actually returned in this response (3), the `page` attribute tells you which page corresponds to this response (1), and the `pages` attribute is the total number of pages (96,293), assuming a page size of `perpage`.

Note Just as with the human user interface of Flickr, you get API results as a series of pages. (Imagine if the API were to send you data about every puppy picture in one shot!) The default value for `perpage` is `100`, and the maximum value is `500`. I choose `3` in this example so that you can easily study the entire response.

---

* Each of the `photo` elements has attributes that enable you to know a bit about what the photo is about (`title`), map them to the photo's various URLs (`id`, `owner`, `secret`, `server`, and `farm`), and tell you about the photo's visibility to classes of users (`ispublic`, `isfriend`, and `isfamily`).

Let's now consider two related issues about this pattern of request and response:

* What does this XML response mean?

* What can you do with the XML response?

What Does This XML Response Mean?

The user interface (UI) and the API give you much of the same information in different forms, meant for different purposes. The requests for the UI and the API are both HTTP `GET`s—but with their corresponding URLs and parameters. In the UI, the body of the response is HTML + JavaScript for display in a web browser. In the API, the response body is XML, meant for consumption by a computer program. (Remember, you learned about XML feeds in Chapter 4. The format of the XML is not the same as RSS or Atom, but you get the benefits of stuff coming back in XML instead of HTML—you don't have to screen-scrape the information. Also remember from the discussion in Chapter 2 that it is possible to screen-scrape HTML + JavaScript, but it's not ideal.)

Let's see how to convince ourselves of the correspondence of the information in the UI and the API. It's very powerful to see this correspondence—the same information is in the UI and from the API—because you'll get a vivid visual confirmation that you understand what's happening in the API. Let's return to comparing the following (when you are logged out of Flickr—to make sure you see only public photos):

```
http://www.flickr.com/photos/tags/puppy/
```

with the following:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags=puppy&per_page=3
```

What type of comparisons can you do?

* You can compare the total numbers of photos in the UI and the API (which you might expect to be same but are not quite the same because of privacy options—see the "Why Are Flickr UI Results Not the Same As Those in the API?" sidebar).
* You can map the information about the photo elements into the photo URLs in order to see what photos are actually being referred to by the API response.

With what you learned in Chapter 2 and with the attributes from the `photo` element, you can generate the URL for the photo. Take, for instance, the first `photo` element:

```
<photo id="1153699093" owner="7841384@N07" secret="d1fba451c9" server="1023"
       farm="2" title="willy after bath and haircut" ispublic="1" isfriend="0"
       isfamily="0" />
```

With this you can tabulate the parameters listed in Table 6-1.

Table 6-1. Parameters Associated with Photo 1153699093

Parameter	Value
photo-id	1153699093
farm-id	2
server-id	1023

```
photo-secret   d1fba451c9
file-suffix    jpg
user-id7841384@N07
```

Remember, the URL templates for the context page of a photo is as follows:

```
http://www.flickr.com/photos/{user-id}/{photo-id}/
```

And the link to the medium-sized photo is as follows:

```
http://farm{farm-id}.static.flickr.com/{server-id}/{photo-id}_{photo-secret}.jpg
```

So, the following are the URLs:

```
http://www.flickr.com/photos/7841384@N07/1153699093/
http://farm2.static.flickr.com/1023/1153699093_d1fba451c9.jpg
```

You can follow the same procedure for all the photos—but one would probably be enough for you to use to compare with the photos in the UI. (You're likely to see the same photo from the API in the UI and hence confirm that the results are the same.)

Note You might wonder how you derive the URL for the original image. Assuming that the original photo is publicly accessible at all, you add `&extras=original_format` to the query to get the `originalsecret` and `originalformat` attributes.

WHY ARE FLICKR UI RESULTS NOT THE SAME AS THOSE IN THE API?

The information available in the Flickr API and in the Flickr UI are closely aligned, so much so that it's easy to think they are the same. Not so. You as a Flickr user can set whether your photos are visible to site-wide searches in the Flickr UI and whether your photos are visible to other users via the API at the following location:

```
http://flickr.com/account/prefs/optout/?from=privacy
```

If any user with public photos tagged with `puppy` has enabled results from one type but not the other type of search to be visible, then what you get from the API and the UI will be different when you look for `puppy`-tagged photos. I still expect that the results will be similar since I would guess that most people have not hidden their public photos from the public search or the API.

What Can You Do with the XML Response?

Now that you know that you can generate an HTML representation of each photo, let's think about what you use `flickr.photos.search` for. Later in the chapter, I'll walk you through the details of how to generate a simple HTML interface written in PHP. Using that method alone and a bit of web programming, you can generate a simple Flickr search engine that lets you page through search results. You can do many other things as well. For example, you could generate an XML feed from this data. With feeds coming out the API, you'd be able to use all the techniques you learned in Chapter 4 (including mashing up feeds with Yahoo! Pipes). You might not have all the information you could

ever want; there are other methods in the Flickr API that will give you more information about the photos, and I will show you how to use those methods later in the chapter.

Where to go from here? First, you won't be surprised to learn that many other parameters are available to you for `flickr.photos.search` given how many search options there are in the Flickr UI for search (see Chapter 2 for a review). You can learn more about those parameters by reading the documentation for the method here:

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

Here you will see documented all the possible arguments you can pass to the method. In addition, you see an example response that, not surprisingly, should look similar to the XML response we studied earlier. In addition, you will see mention of two topics that I glossed over in my example:

Error handling: The carefully constructed simple request should work as described here. But errors do happen, and Flickr uses an error-handling process that includes the use of error codes and error messages. Any robust source code you write should take into account this error handling.

Authorization: The example we looked at involved only public photos. Things get a lot messier once you work with private photos. In the UI, that means having to establish a user account and being logged in. With the API, there is a conceptually parallel process with one twist. Three parties are involved in Flickr authentication; in addition to the user and Flickr, there is a third-party application that uses the API. To avoid users having to give their passwords to the third-party application to log in on their behalf, there's a complicated dance that could easily obscure the core ideas behind the API. We'll look at authentication later in this chapter.

As interesting as `flickr.photos.search` is (and it is probably the single most useful and functionally rich Flickr method), you'll want to see what other methods there are in the API. I'll show you how to learn about the diversity of functionality available in the Flickr API by using, among other things, the very cool Flickr API Explorer.

You'll find that a good understanding of Flickr's functionality will come in handy when you learn how to use the API. (This is a point I was stressing in Chapter 2.) There's the ad hoc method of learning the API that is to start with a specific problem you want to solve—and then look for a specific piece of functionality in the API that will solve it. You can consult the Flickr API documentation when you need it and use the Flickr API Explorer. You can also try to take a more systematic approach to outlining what's available in the Flickr API (a bit like the detailed discussion of Flickr's URL language I presented in Chapter 2). I will outline a method for doing this. This is cool, because such a method will involve the Flickr API telling us about itself! I will use that as an opportunity to talk about APIs in general.

A large part of this chapter will cover some of the programming details you will encounter working with the Flickr API and other APIs. The way I showed you to formulate the Flickr API through the use of the following:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags=puppy&per_page=3
```

is only one way of three ways to do so. There are also other formats for the response that Flickr can generate. I'll cover the different request and response formats in the "Request and Response Formats" section later in this chapter.

When working with these Flickr web services, you find that a detailed understanding of HTTP, XML, and request and response formats is helpful—but you're likely to want to work at a higher level of abstraction once you get down to doing some serious programming. That's when third-party wrappers to the API, what Flickr calls *API kits*, come into play. I will cover how to use a number of the PHP Flickr API kits later in this chapter.

There is a lot of complexity in using APIs, but just don't forget the essential pattern that you find in the Flickr API: *you make an HTTP request formatted with the correct parameters, and you get back in your response XML that you can then parse*. The rest is detail.

The bottom line is that you can learn a lot by using and studying the Flickr API. It's extremely well designed in so many ways. It's certainly not perfect—and there are other, sometimes better, ways of instantiating the certain functionality of an API. A close study of the Flickr API will help you understand the APIs of other systems—as you will see in Chapter 7.

API Documentation, Community, and Policy

You can find the official documentation for the Flickr API here:

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

As you get familiar with the API, I recommend consulting or lurking in two communities:

* The Flickr API mailing list (`http://tech.groups.yahoo.com/group/yws-flickr/`)
* The Flickr API group on Flickr (`http://www.flickr.com/groups/api/`)

You can get a feel for what people are thinking about in terms of the API and get your questions answered too. When you become more proficient with the API, you can start answering other people's questions. (The first group is more technically oriented, and the second one is more focused on the workflow of Flickr.)

Terms of Use for the API

API providers, including Flickr, require assent to terms of service (ToS, also known as *terms of use*) for access to the API. The terms of use for the Flickr API are at the following location:

`http://www.flickr.com/services/api/tos/`

There is, of course, no substitute for reading the ToS carefully for yourself. Here I list a few highlights of the ToS, including what it tells you about Flickr and how you might find similar issues raised in the ToS of other web APIs. Here are some factors:

Commercial vs. noncommercial use: You need to apply for special permission to use the Flickr API for commercial purposes.

Payment for use: The Flickr API is free for noncommercial use, like many web APIs are.

Rate limits: The ToS states that you can't use an "unreasonable amount of bandwidth."

Compliance with the user-facing website ToS: Programmatic access to Flickr content must comply with all the terms that govern human users of Flickr. This includes, for instance, the requirement to link to Flickr when embedding Flickr-hosted photos.

Content ownership: You need to pay attention to the ownership of photos, including any Creative Commons licenses attached to the photos.

Caching: You are supposed to cache Flickr photos for only a "reasonable" period of time to provide your Flickr service.

Privacy policies: Your applications are supposed to respect (and by proxy enforce) Flickr's privacy policy and the photo owner's settings. You are supposed to have a clearly articulated privacy policy of your own for the photos you access through the Flickr API.

Using the Flickr API Explorer and Documentation

The column on the right at `http://www.flickr.com/services/api/` lists all the API methods available in the Flickr API. There are currently 106 methods in the API organized in the following 23 groups:

* Activity
* Auth
* Blogs
* Contacts
* Favorites
* Groups
* Groups.pools
* Interestingness
* People
* Photos
* Photos.comments
* Photos.geo
* Photos.licenses
* Photos.notes
* Photos.transform

* Photos.upload
* Photosets
* Photosets.comments
* Prefs
* Reflection
* Tags
* Test
* URLs

I've already used two methods in the early parts of the chapter: `flickr.test.echo` (part of the `test` group) and `flickr.photos.search` (part of the `photos` group). In this section, I'll show you how to exercise a specific API method in detail and return to looking at the full range of methods. Here I use `flickr.photos.search` for an example.

You can get the documentation for any method here:

```
http://www.flickr.com/services/api/{method-name}.html
```

For example:

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

Notice the following subsections in the documentation of each method:

* A description of the method's functionality.
* Whether the method requires authentication and, if so, the minimum level of permission needed: one of none, `read`, `write`, or `delete`. `read` is permission to read private information; `write` is permission to add, edit, and delete metadata for photos in addition to the `read` permission; and `delete` is permission to delete photos in addition to the `write` and `read` permissions.
* Whether the method needs to be signed. All methods that require authentication require signing. Some methods, such as all the ones belonging to the `auth` group (for example, `flickr.auth.getToken`) don't need authentication but must be signed. I will describe the mechanics of signing later in the chapter.
* A list of arguments, the name of each argument, whether it is required or mandatory, and a short description of the argument.
* An example response.
* The error codes.

In the documentation, there is a link to the Flickr API Explorer:

```
http://www.flickr.com/services/api/explore/?method={method-name}
```

For example:

```
http://www.flickr.com/services/api/explore/?method=flickr.photos.search
```

The Flickr API Explorer is my favorite part of the Flickr API documentation. Figure 6-1 shows the API Explorer for `flickr.photos.getInfo`. For each method, the API Explorer not only documents the arguments but lets you fill in arguments and call the

method (with your argument values) right within the browser. You have three choices for how to sign the call:

* You can leave the call unsigned.
* You can sign it without attaching any user information.
* You can sign it and grant the call `write` permission for yourself (as the logged-in user).

Insert 858Xf0601.tif

Figure 6-1. The Flickr API Explorer for `flickr.photos.getInfo`. (Reproduced with permission of Yahoo! Inc. ® 2007 by Yahoo! Inc. YAHOO! and the YAHOO! logo are trademarks of Yahoo! Inc.)

When you hit the Call Method button, the XML of the response is displayed in an iframe, and the URL for the call is displayed below the iframe. You can use the Flickr API Explorer to understand how a method works. In the case of the unsigned call, you can copy the URL and substitute your own API key to use it in your own programs.

For example, if you use the Flickr API Explorer to call `flickr.photos.search` with the tag set to `puppy` and then click the Do Not Sign Call button, you'll get a URL similar to this:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags=puppy
```

Copy and paste the URL you get from the Flickr API Explorer into a web browser to convince yourself that in this case of searching for public images, you can now call the Flickr API through a simple URL that returns results to you in XML.

Note The Flickr API Explorer uses an `api_key` that keeps changing. But that's fine because you're supposed to use your own API key in your own applications. Substituting your own key is not hard for an unsigned call.

Now when I click Sign Call As Raymond Yee with Full Permissions, I get the following URL:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags=puppy&auth_token=72157601583650732-e30f91f3313b3d14&
api_sig=3d7a2d1975e9699246a299d2deaf5b70
```

When I use that URL immediately—before the key expires—I get to perform searches for `puppy`-tagged photos with `write` permission for my user account. This URL is useful to test the functionality of the method. It's not so useful for dropping into a program. Getting it to work is not simply a matter of substituting your own `api_key` but also getting a new `auth_token` and calculating the appropriate `api_sig` (that is, signing the call)—tasks that take a couple of more calls to the Flickr API and a bit of

computing. It's this set of calculations, which makes authorization one of the trickiest parts of the Flickr API, that I will show you how to do later in the chapter.

Calling a Basic Flickr API Method from PHP

Now that you have used the Flickr API Explorer and documentation to make sense of the details of a given API method and to package a call in the browser, you will now learn how to make a call from a simple third-party application that you write. In this section, I return to the `flickr.photos.search` example I used earlier in this chapter:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags={tag}&per_page={per_page}
```

Specifically, the following:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key={api_ke
y}
&tags=puppy&per_page=3
```

generates a response similar to this:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page="1" pages="96293" perpage="3" total="288877">
  <photo id="1153699093" owner="7841384@N07" secret="d1fba451c9" server="1023"
         farm="2" title="willy after bath and haircut" ispublic="1" isfriend="0"
         isfamily="0" />
  <photo id="1154506492" owner="7841384@N07" secret="881ff7c4bc" server="1058"
         farm="2" title="rocky with broken leg" ispublic="1" isfriend="0"
         isfamily="0" />
  <photo id="1153588011" owner="90877382@N00" secret="8a7a559e68" server="1288"
         farm="2" title="DSC 6503" ispublic="1" isfriend="0" isfamily="0" />
</photos>
</rsp>
```

In the earlier narrative, I described to you how you can extract from the XML response such quantities as the total number of photos and how to derive from a `photo` element such as this:

```
<photo id="1153699093" owner="7841384@N07" secret="d1fba451c9" server="1023"
       farm="2" title="willy after bath and haircut" ispublic="1" isfriend="0"
       isfamily="0" />
```

Here are the URLs for the corresponding photo:

```
http://www.flickr.com/photos/7841384@N07/1153699093/
http://farm2.static.flickr.com/1023/1153699093_d1fba451c9.jpg
```

In the following sections, I'll show you how to instantiate that logic into code. Specifically, we will write a simple third-party Flickr app in PHP that makes a Flickr API call and converts the response to HTML. We'll use two important sets of techniques that I will elaborate on in some detail, HTTP clients and XML processing, after which I describe how to use these techniques to make the call to Flickr. Here I focus on PHP, but you can apply these ideas to your language of choice.

Tip When debugging web services, I have found it helpful to use a network protocol analyzer such as Wireshark (http://en.wikipedia.org/wiki/Wireshark). Properly formulating a web service call often requires trial and error. Through its support of HTTP, Wireshark lets you see exactly what was sent and what was received, including HTTP headers, response codes, and entity bodies.

HTTP Clients

Let's consider first the issue of how to perform an HTTP `GET` request and retrieve the response in PHP. The function `file_get_contents` takes a URL and returns the corresponding content in a string, provided the `allow_url_fopen` option is set to `true` in the system-wide `php.ini`. For example:

```
<?php
// retrieve Atom feed of recent flower-tagged photos in Flickr
$url = "http://api.flickr.com/services/feeds/photos_public.gne?tags=flower&lang=
en-us&format=atom";

$content = file_get_contents($url);
echo $content;
?>
```

If you are using an instance of PHP for which URL access for `file_get_contents` is disabled (which is not uncommon for shared hosting facilities with security concerns), then you might still be able to use the cURL extension for PHP (`libcurl`) to perform the same function. `libcurl` is documented here:

```
http://us3.php.net/curl
```

The following `getResource` function does what `file_get_contents` does. Note the four steps in using the `curl` library: initializing the call, configuring options, executing the call, and closing down the handle:

```
<?php
function getResource($url){
// initialize a handle
        $chandle = curl_init();
// set URL
        curl_setopt($chandle, CURLOPT_URL, $url);
// return results a s string
        curl_setopt($chandle, CURLOPT_RETURNTRANSFER, 1);
// execute the call
        $result = curl_exec($chandle);
        curl_close($chandle);

        return $result;
}
?>
```

The many options you can configure in `libcurl` are documented here:

```
http://us3.php.net/manual/en/function.curl-setopt.php
```

In this book, I use libcurl for HTTP access in PHP. Should you not be able to use libcurl, you can use the libcurl Emulator, a pure-PHP implementation of libcurl:

```
http://code.blitzaffe.com/pages/phpclasses/files/libcurl_emulator_52-7
```

Note I will often use curl to demonstrate HTTP requests in this book. More information is available at http://curl.haxx.se/.

A Refresher on HTTP

So, how would you configure the many options of a library such as libcurl? Doing so requires some understanding of HTTP. Although HTTP is a foundational protocol, it's really quite easy to get along, even as programmers, without knowing the subtleties of HTTP. My goal here is not to describe HTTP in great detail. When you need to understand the protocol in depth, I suggest reading the official specifications; here's the URL for HTTP 1.0 (RFC 1945):

```
http://tools.ietf.org/html/rfc1945
```

And here's the URL for HTTP 1.1:

```
http://tools.ietf.org/html/rfc2616
```

You can also consult the official W3C page:

```
http://www.w3.org/Protocols/
```

Reading and digesting the specification is not the best way to learn HTTP for most of us, however. Instead of formally learning HTTP all in one go in its formal glory, I've learned different aspects of HTTP at different times, and because that partial knowledge was sufficient for the situation at hand, I felt no need to explore the subtleties of the protocol. It was a new situation that prompted me to learn more. My first encounter with HTTP was simply surfing the Web and using URLs that had http for their prefix. For a little while, I didn't even know that http wasn't the only possible scheme in a URI—and that, technically, http:// is not a redundant part of a URI, even if on business cards it might be. (People understand www.apress.com means an address for a page in web browser—and the prefix http:// just looks geeky.)

Later when I learned about HTML forms, I learned that there are two possible values for the method attribute for FORM: get and post.[1] (It puzzled me why it wasn't get and put since put is often the complement to get.) For a long time, the only difference I perceived between get and post was that a form that uses the get method generates URLs that include the name/value pairs of the submitted form elements, whereas post doesn't. The practical upshot for me was that get produces addressable URLs, whereas post doesn't. I thought I had post figured out as a way of changing the state of resources (as opposed to using get for asking for information)—and then I learned about the formal way of distinguishing between safe and idempotent methods (see the "Safe Methods and Idempotent Methods" sidebar for a further explanation of these terms). Even with post, it turns out that there is a difference between two different forms of form encoding stated in the FORM enctype attribute (application/x-www-form-urlencoded vs. multipart/form-data), a distinction that is not technically part of HTTP but that will have a practical effect on how you programmatically make certain HTTP requests.[2]

SAFE METHODS AND IDEMPOTENT METHODS

The HTTP specification defines safe methods and idempotent methods here:

`http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`

Safe methods "*should not* have the significance of taking an action other than retrieval" of a representation of a resource. You shouldn't be changing the resource through a safe method. In HTTP, `GET` and `HEAD` methods are supposed to be safe. Unsafe methods that have the potential of altering the state of the retrieved resource include `POST`, `PUT`, and `DELETE`.

Idempotent methods are those that have the same effect on the resource whether they are performed once or more than one time. It's akin to multiplying a number by zero—the result is the same whether you do it once or more than once. According to the HTTP standard, the `GET`, `HEAD`, `PUT`, and `DELETE` methods should be idempotent operations. Moreover, "the methods `OPTIONS` and `TRACE` *should not* have side effects and so are inherently idempotent."

1. http://www.w3.org/TR/html401/interact/forms.html#h-17.13.1
2. http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4

Formal Structure of HTTP

When I moved from writing HTML to writing basic web applications, I then learned more about the formal structure of HTTP—and how what I had learned fit within a larger structure of what HTTP is capable of doing. For instance, I learned that, in addition to `GET` and `POST`, HTTP defines six other methods, among which was a `PUT` after all. (It's just that few, if any, web browsers support `PUT`.) Let me describe the parts of HTTP 1.1 request and response messages. (I draw some terminology in the following discussion from the excellent presentation of HTTP by Leonard Richardson and Sam Ruby in *Restful Web Services*.)

An HTTP request is composed of the following pieces:

* The *method* (also known as *verb* or *action*). In addition to `GET` and `POST`, there are six others defined in the HTTP specification: `OPTIONS`, `HEAD`, `PUT`, `DELETE`, `TRACE`, and `CONNECT`. `GET` and `POST` are widely used and supported in web browsers and programming libraries.
* The *path*—the part of the URL to the right of the hostname.
* A series of *request headers*. See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html`.
* A request *body*, which may be empty.

The parts of the HTTP response include the following:

* A *response code*. You can find a long list of codes at `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`. Examples include 200 OK, 400 Bad Request, and 500 Internal Server Error.
* *Response headers*. See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html`.
* A response *body*.

Let's consider the following example:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search
&api_key={api-key}&tags=puppy&per_page=3
```

To track the HTTP traffic, I'm using `curl` (with the `verbose` option) to make the call. (You can also use Wireshark to read the parameters of the HTTP request and response):

```
curl --verbose "http://api.flickr.com/services/rest/?method=flickr.photos.search
&api_key={api-key}&tags=puppy&per_page=3"
```

This is an edited version of what I get:

```
* About to connect() to api.flickr.com port 80
*   Trying 68.142.214.24... * connected
* Connected to api.flickr.com (68.142.214.24) port 80
> GET /services/rest/?method=flickr.photos.search&api_key={api-key}&tags=puppy
&per_page=3 HTTP/1.1
User-Agent: curl/7.13.2 (i386-pc-linux-gnu) libcurl/7.13.2 OpenSSL/0.9.7e
zlib/1.2.2
libidn/0.5.13
Host: api.flickr.com
Pragma: no-cache
Accept: */*

< HTTP/1.1 200 OK
< Date: Tue, 21 Aug 2007 20:42:54 GMT
< Server: Apache/2.0.52
< Set-Cookie: cookie_l10n=en-us%3Bus; expires=Friday, 20-Aug-10 20:42:54 GMT;
path=/; domain=flickr.com
< Set-Cookie: cookie_intl=deleted; expires=Monday, 21-Aug-06 20:42:53 GMT;
path=/;
domain=flickr.com
< Content-Length: 570
< Connection: close
< Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page="1" pages="97168" perpage="3" total="291503">
        <photo id="1196703288" owner="69161261@N00" secret="d4e5a75664"
server="1412" farm="2" title="Pomeranian" ispublic="1" isfriend="0" isfamily="0"
/>
        <photo id="1196707012" owner="58944004@N00" secret="9d88253b87"
server="1200" farm="2" title="Fraggle" ispublic="1" isfriend="0" isfamily="0" />
        <photo id="1195805641" owner="21877391@N00" secret="311d276ec7"
server="1177" farm="2" title="Blue" ispublic="1" isfriend="0" isfamily="0" />
</photos>
</rsp>
```

Let's break down the specifics of this request/response exchange, as shown in Table 6-2 and Table 6-3, respectively.

Table 6-2. The HTTP Request Parameters in a Flickr Call

Parameter	Value
`method`	`GET`
`path`	`/services/rest/?method=flickr.photos.search&api_key={api-key}&tags=puppy&per_page=3`
`headers`	Four headers of the following types: `User-Agent`, `Host` (which identifies `api.flickr.com`), `Pragma`, and `Accept`
`response`	Empty (typical of `GET` requests)

Table 6-3. The HTTP Response Parameters in a Flickr Call

Parameter	Value
Response code	`200 OK`
Response headers	Seven headers of the following types: `Date`, `Server`, `Set-Cookie` (twice), `Content-Length`, `Connection`, `Content-Type`
Response body	The XML document representing the photos that match the query

Keep this example in mind to see how the HTTP request and response are broken down as you continue through this chapter. Notice the structure of having a document (in the body) and a set of headers in both the HTTP request and response structure.

Even though you now understand the basic structure of HTTP, the point is to not have to understand the intricacies of the protocol. You can shield yourself from the details while still taking advantage of the rich functionality of HTTP with the right choice of tools and libraries. Richardson and Ruby provide a helpful shopping list of desirable features in an HTTP client library:

* Support for HTTPS and SSL certificate validation.
* Support for what they consider to be the five main HTTP methods: `GET`, `HEAD`, `POST`, `PUT`, and `DELETE`. Some give you only `GET`. Others let you use `GET` and `POST`.
* Lets you customize the request body of `POST` and `PUT` requests.
* Lets you customize the HTTP request headers.
* Gives you access to the response code and HTTP response headers—not just the body of the response.
* Lets you communicate through an HTTP proxy.

They list the following features as nice options:

* Lets you request and handle data *compression*. The relevant HTTP request/response headers are `Accept-Encoding` and `Encoding`.
* Lets you deal with *caching*. The relevant HTTP headers are `ETag` and `If-Modified-Since` and `ETag` and `Last-Modified`.
* Lets you deal with the most common forms of HTTP *authentication*: `Basic`, `Digest`, and `WSSE`.
* Lets you deal with HTTP *redirects*.

* Helps you deal with HTTP *cookies*.

They also make specific recommendations for what to use in various languages, including the following:

* The `httplib2` library (`http://code.google.com/p/httplib2/`) for Python
* `HttpClient` in the Apache Jakarta project (`http://jakarta.apache.org/commons/httpclient/`)
* `rest-open-uri`, a modification of Ruby's `open-uri` to support more than the `GET` method (`http://rubyforge.org/projects/rest-open-uri/`)

XML Processing

Once you have made the HTTP request to the Flickr API, you are left with the second big task of processing the XML document contained in the response body. The topic of how to process XML is a large subject, especially when you consider techniques in multiple languages. What I show you here is one way of parsing XML in PHP 5 through an example involving a reasonably complicated XML document (with namespaces and attributes).

The `simpleXML` library is built into PHP 5, which is documented here:

`http://us3.php.net/simplexml`

I found the following article particularly helpful to me in understanding how to handle namespaces and mixed content in `simpleXML`:

`http://devzone.zend.com/node/view/id/688`

In the following example, I parse an Atom feed (an example from Chapter 4) and print various parts of the document. I access XML elements as though they are PHP object properties (using `->element-name`) and the attributes as though they are members of an array (using `["attribute-name"]`), for example, `$xml->title` and `$entry->link["href"]`. First I list the code and then the output from the code:

```
<?php
// An example to show how to parse an Atom feed (with multiple namespaces)
// with SimpleXML
# create the XML document in the $feed string
$feed=<<<EOT
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
      xmlns:dc="http://purl.org/dc/elements/1.1/">
  <title>Apress :: The Expert's Voice</title>
  <subtitle>Welcome to Apress.com. Books for Professionals,
    by Professionals(TM)...with what the
    professional needs to know(TM)</subtitle>
  <link rel="alternate" type="text/html" href="http://www.apress.com/"/>
  <link rel="self"
        href="http://examples.mashupguide.net/ch06/Apress.Atom.with.DC.xml"/>
  <updated>2007-07-25T12:57:02Z</updated>
  <author>
    <name>Apress, Inc.</name>
    <email>support@apress.com</email>
```

```
  </author>
  <id>http://apress.com/</id>
  <entry>
    <title>Excel 2007: Beyond the Manual</title>
    <link href="http://www.apress.com/book/bookDisplay.html?bID=10232"/>
    <id>http://www.apress.com/book/bookDisplay.html?bID=10232</id>
    <updated>2007-07-25T12:57:02Z</updated>
    <dc:date>2007-03</dc:date>
    <summary type="html"
      >&lt;p&gt;&lt;i&gt;Excel 2007: Beyond the Manual&lt;/i&gt; will introduce
those who are already familiar with Excel basics to more advanced features, like
consolidation, what-if analysis, PivotTables, sorting and filtering, and some
commonly used functions. You'll learn how to maximize your efficiency at
producing
professional-looking spreadsheets and charts and become competent at analyzing
data
using a variety of tools. The book includes practical examples to illustrate
advanced features.&lt;/p&gt;</summary>
  </entry>
  <entry>
    <title>Word 2007: Beyond the Manual</title>
    <link href="http://www.apress.com/book/bookDisplay.html?bID=10249"/>
    <id>http://www.apress.com/book/bookDisplay.html?bID=10249</id>
    <updated>2007-07-25T12:57:10Z</updated>
    <dc:date>2007-03-01</dc:date>
    <summary type="html"
      >&lt;p&gt;&lt;i&gt;Word 2007: Beyond the Manual&lt;/i&gt; focuses on new
features of Word 2007 as well as older features that were once less accessible
than
they are now. This book also makes a point to include examples of practical
applications for all the new features. The book assumes familiarity with Word
2003
or earlier versions, so you can focus on becoming a confident 2007
user.&lt;/p&gt;</summary>
  </entry>
</feed>
EOT;

# instantiate a simpleXML object based on the $feed XML
$xml = simplexml_load_string($feed);

# access the title and subtitle elements
print "title: {$xml->title}\n";
print "subtitle: {$xml->subtitle}\n";

# loop through the two link elements, printing all the attributes for each link.

print "processing links\n";
foreach ($xml->link as $link) {
  print "attribute:\t";
  foreach ($link->attributes() as $a => $b) {
    print "{$a}=>{$b}\t";
  }
  print "\n";
```

```
}
print "author: {$xml->author->name}\n";

# let's check out the namespace situation

$ns_array = $xml->getDocNamespaces(true);

# display the namespaces that are in the document
print "namespaces in the document\n";
foreach ($ns_array as $ns_prefix=>$ns_uri) {
  print "namespace: ${ns_prefix}->${ns_uri}\n";
}
print "\n";

# loop over all the entry elements
foreach ($xml->entry as $entry) {
  print "entry has the following elements in the global namespace: \t";

  // won't be able to access tags that aren't in the global namespace.
  foreach ($entry->children() as $child) {
    print $child->getName(). " ";
  }
  print "\n";
  print "entry title: {$entry->title}\t link: {$entry->link["href"]}\n";

  // show how to use xpath to get date
  // note dc is registered already to $xml.
  $date = $entry->xpath("./dc:date");
  print "date (via XPath): {$date[0]}\n";

  // use children() to get at date
  $date1 = $entry->children("http://purl.org/dc/elements/1.1/");
  print "date (from children()): {$date[0]}\n";

  }

# add <category term="books" /> to feed -- adding the element will work
# but the tag is in the wrong place to make a valid Atom feed.
# It is supposed to go before the entry elements
$category = $xml->addChild("category");
$category->addAttribute('term','books');

# output the XML to show that category has been added.
$newxmlstring = $xml->asXML();
print "new xml (with category tag): \n$newxmlstring\n";
?>
```

The output from the code is as follows:

```
title: Apress :: The Expert's Voice
subtitle: Welcome to Apress.com. Books for Professionals,
by Professionals(TM)...with what the professional needs to know(TM)
processing links
attribute: rel=>alternate type=>text/html href=>http://www.apress.com/
```

```
attribute: rel=>self
href=>http://examples.mashupguide.net/ch06/Apress.Atom.with.DC.xml
author: Apress, Inc.
namespaces in the document
namespace: ->http://www.w3.org/2005/Atom
namespace: dc->http://purl.org/dc/elements/1.1/

entry has the following elements in the global namespace: title link id
updated summary
entry  title: Excel 2007: Beyond the Manual link:
http://www.apress.com/book/bookDisplay.html?bID=10232
date (via XPath): 2007-03
date (from children()): 2007-03
entry has the following elements in the global namespace: title link id
updated summary
entry  title: Word 2007: Beyond the Manual link:
http://www.apress.com/book/bookDisplay.html?bID=10249
date (via XPath): 2007-03-01
date (from children()): 2007-03-01
new xml (with category tag):
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
       xmlns:dc="http://purl.org/dc/elements/1.1/">
  <title>Apress :: The Expert's Voice</title>
  <subtitle>Welcome to Apress.com. Books for Professionals,
by Professionals(TM)...with what the professional needs to know(TM)</subtitle>
  <link rel="alternate" type="text/html" href="http://www.apress.com/"/>
  <link rel="self"
        href="http://examples.mashupguide.net/ch06/Apress.Atom.with.DC.xml"/>
  <updated>2007-07-25T12:57:02Z</updated>
  <author>
    <name>Apress, Inc.</name>
    <email>support@apress.com</email>
  </author>
  <id>http://apress.com/</id>
  <entry>
    <title>Excel 2007: Beyond the Manual</title>
    <link href="http://www.apress.com/book/bookDisplay.html?bID=10232"/>
    <id>http://www.apress.com/book/bookDisplay.html?bID=10232</id>
    <updated>2007-07-25T12:57:02Z</updated>
    <dc:date>2007-03</dc:date>
    <summary type="html">&lt;p&gt;&lt;i&gt;Excel 2007: Beyond the
Manual&lt;/i&gt; will introduce those who are already familiar with Excel basics
to
more advanced features, like consolidation, what-if analysis, PivotTables,
sorting
and filtering, and some commonly used functions. You'll learn how to maximize
your
efficiency at producing professional-looking spreadsheets and charts and become
competent at analyzing data using a variety of tools. The book includes
practical
examples to illustrate advanced features.&lt;/p&gt;</summary>
  </entry>
  <entry>
```

```
    <title>Word 2007: Beyond the Manual</title>
    <link href="http://www.apress.com/book/bookDisplay.html?bID=10249"/>
    <id>http://www.apress.com/book/bookDisplay.html?bID=10249</id>
    <updated>2007-07-25T12:57:10Z</updated>
    <dc:date>2007-03-01</dc:date>
    <summary type="html">&lt;p&gt;&lt;i&gt;Word 2007: Beyond the
Manual&lt;/i&gt; focuses on new features of Word 2007 as well as older features
that
were once less accessible than they are now. This book also makes a point to
include
examples of practical applications for all the new features. The book assumes
familiarity with Word 2003 or earlier versions, so you can focus on becoming a
confident 2007 user.&lt;/p&gt;</summary>
  </entry>
<category term="books"/></feed>
```

There are certainly alternatives to `simpleXML` for processing XML in PHP 5, but it provides a comfortable interface for a PHP programmer to XML documents.

Note When trying to figure out the structures of PHP objects, consider using one of the following functions: `print_r`, `var_dump`, or `var_export`.

Pulling It All Together: Generating Simple HTML Representations of the Photos

Now we have the two pieces of technology to send an HTTP request to Flickr and parse the XML in the response:

* The `getResource` function I displayed earlier that uses the `libcurl` library of PHP 5
* The `simpleXML` library to parse the XML response

I'll now show you a PHP script that uses these two pieces of functionality to prompt a user for a tag and that returns the list of five HTML-formatted photos for that tag.

Here's a breakdown of the logical steps that take place in the following script:

1. It displays the total number of pictures (`$xml->photos['total']`).
2. It iterates through the array of photos through an elaboration of the following loop:

```
foreach ($xml->photos->photo as $photo) {
  $id = $photo['id'];
}
```

3. It forms the URL of the thumbnail and the URL of the photo page through the logic contained in the following line:

```
$thumb_url =
  "http://farm{$farmid}.static.flickr.com/{$serverid}/{$id}_{$secret}_t.jpg";
```

The following is one possible version of such a script.[3] (Note the `Content-Type` HTTP response header of `text/html` to keep Internet Explorer happy with XHTML, but the output is XHTML 1.0 Strict.)

3. http://examples.mashupguide.net/ch06/flickrsearch.php

```
<?php
header("Content-Type:text/html");
echo '<?xml version="1.0" encoding="utf-8"?>';
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <title>flickrsearch.php</title>
  </head>
  <body>
<?php
if (isset($_GET['tag'])) {
   do_search($_GET['tag']);
} else {
?>
     <form action="<?php echo $_SERVER['PHP_SELF']?>" method="get">
     <p>Search for photos with the following tag:
    <input type="text" size="20" name="tag"/> <input type="submit"
value="Go!"/></p>
     </form>
<?php
}
?>
<?php

# uses libcurl to return the response body of a GET request on $url
function getResource($url){
  $chandle = curl_init();
  curl_setopt($chandle, CURLOPT_URL, $url);
  curl_setopt($chandle, CURLOPT_RETURNTRANSFER, 1);
  $result = curl_exec($chandle);
  curl_close($chandle);

  return $result;
}

function do_search($tag) {
  $tag = urlencode($tag);

#insert your own Flickr API KEY here

  $api_key = "[API-Key]";
  $per_page="5";
  $url = "http://api.flickr.com/services/rest/?method=flickr.photos.search
&api_key={$api_key}&tags={$tag}&per_page={$per_page}";
```

```
  $feed = getResource($url);
  $xml = simplexml_load_string($feed);
  print "<p>Total number of photos for {$tag}: {$xml->photos['total']}</p>";

# http://www.flickr.com/services/api/misc.urls.html
# http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}.jpg
foreach ($xml->photos->photo as $photo) {
  $title = $photo['title'];
  $farmid = $photo['farm'];
  $serverid = $photo['server'];
  $id = $photo['id'];
  $secret = $photo['secret'];
  $owner = $photo['owner'];
  $thumb_url = "http://farm{$farmid}.static.flickr.com/{$serverid}/
{$id}_{$secret}_t.jpg";
  $page_url = "http://www.flickr.com/photos/{$owner}/{$id}";
  $image_html= "<a href='{$page_url}'><img alt='{$title}'
src='{$thumb_url}'/></a>";
  print "<p>$image_html</p>";
}

} # do_search
?>
  </body>
</html>
```

Where Does This Leave Us?

This code allows you to search and display some pictures from Flickr. More important, it is an example of a class of Flickr methods: those that require neither signing nor authorization to be called. You will see in the next section how to determine which of the Flickr API methods fall in that category. In the following sections, you'll look at generalizing the techniques you have used in studying `flickr.photos.search` to the other capabilities of the Flickr API.

The Flickr API in General

What are some approaches to learning the Flickr API? My first suggestion is to look around the documentation and glance through the list of API methods here:

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

While you are doing so, you should think back to all the things you know about Flickr as an end user (aspects I discussed in Chapter 2) and see whether they are reflected in the API. For example, can you come up with an API call to calculate the NSID of your own account? What is a URL to return that information? Hint: `flickr.people.findByUsername`.

Perhaps the best way to learn about the API is to have a specific problem in mind and then let that problem drive your learning of the API. Don't try to learn commit the entire API to memory—that's what the documentation is for.

As I argued earlier, calls that require neither signing nor authorization (such as `flickr.photos.search`) are the easiest place to start. How would you figure out which calls those are? You can make pretty good guesses from the names of methods. For instance, you won't be surprised that the method `flickr.photos.geo.setLocation` would need authorization: you would be using it to change the geolocation of a photo, an act that would require Flickr to determine whether you have the permission to do so. On the other hand, the method `flickr.groups.pools.getPhotos` allows you to retrieve photos for a given group. A reasonably proficient Flickr user knows that there are public groups whose photos would be visible to everybody, including those who are not logged in to Flickr at all. Hence, it's not surprising that this method would not require signing or authorization.

Using flickr.reflection Methods

You can get fairly far by eyeballing the list of Flickr methods for ones that do not require any permission to execute. (Recall the levels of permissions within the Flickr API: none, `read`, `write`, and `delete`.) It turns out that the Flickr API has a feature that you won't find in too many other web APIs: *the Flickr API has methods that return information about the API itself*. `flickr.reflection.getMethods` returns a list of all the Flickr methods available. `flickr.reflection.getMethodInfo` takes a given method name and returns the following:

* A description of the method
* Whether the method needs to be signed
* Whether the method needs to be authorized
* The minimal permission level needed by the method (0 = none, 1 = `read`, 2= `write`, 3=`delete`)
* The list of arguments for the method, including a description of the argument and whether it is optional
* The list of possible errors arising from calling the method

For example, let's look at what the Flickr API tells us about `flickr.photos.geo.setLocation`. You can use this format:

```
http://api.flickr.com/services/rest/?method= flickr.reflection.getMethodInfo
&api_key={api-key}&method_name={method-name}
```

Specifically, you can use this:

```
http://api.flickr.com/services/rest/?method=flickr.reflection.getMethodInfo
&api_key={api-key}&method_name=flickr.photos.geo.setLocation
```

to generate this:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<method name="flickr.photos.geo.setLocation" needslogin="1" needssigning="1"
        requiredperms="2">
  <description>Sets the geo data (latitude and longitude and, optionally,
the accuracy level) for a photo.
```

Before users may assign location data to a photo they must define who, by
default,
may view that information. Users can edit this preference at <a
href="http://www.flickr.com/account/geo/privacy/">http://www.flickr
.com
/account/geo/privacy/. If a user has not set this preference, the API
method will return an error.</description>
</method>
<arguments>
  <argument name="api_key" optional="0">Your API application key. <a
href="/services/api/misc.api_keys.html">See here for more
details.</argument>
  <argument name="photo_id" optional="0">The id of the photo to set location
data for.</argument>
  <argument name="lat" optional="0">The latitude whose valid range is -90 to
90. Anything more than 6 decimal places will be truncated.</argument>
  <argument name="lon" optional="0">The longitude whose valid range is -180
to 180. Anything more than 6 decimal places will be truncated.</argument>
  <argument name="accuracy" optional="1">Recorded accuracy level of the
location information. World level is 1, Country is ~3, Region ~6, City ~11,
Street
~16. Current range is 1-16. Defaults to 16 if not specified.</argument>
</arguments>
<errors>
  <error code="1" message="Photo not found">The photo id was either invalid
or was for a photo not viewable by the calling user.</error>
  <error code="2" message="Required arguments missing.">Some or all of the
required arguments were not supplied.</error>
  <error code="3" message="Not a valid latitude.">The latitude argument
failed validation.</error>
  <error code="4" message="Not a valid longitude.">The longitude argument
failed validation.</error>
  <error code="5" message="Not a valid accuracy.">The accuracy argument
failed validation.</error>
  <error code="6" message="Server error.">There was an unexpected problem
setting location information to the photo.</error>
  <error code="7" message="User has not configured default viewing settings
for location data.">Before users may assign location data to a photo they must
define who, by default, may view that information. Users can edit this
preference at

http://www.flickr.com/account/geo/privacy/</error>
  <error code="96" message="Invalid signature">The passed signature was
invalid.</error>
  <error code="97" message="Missing signature">The call required signing but
no signature was sent.</error>
  <error code="98" message="Login failed / Invalid auth token">The login
details or auth token passed were invalid.</error>
  <error code="99" message="User not logged in / Insufficient
permissions">The method requires user authentication but the user was not logged
in,
or the authenticated method call did not have the required permissions.</error>
  <error code="100" message="Invalid API Key">The API key passed was not

```
valid or has expired.</error>
  <error code="105" message="Service currently unavailable">The requested
service is temporarily unavailable.</error>
  <error code="111" message="Format &quot;xxx&quot; not found">The requested
response format was not found.</error>
  <error code="112" message="Method &quot;xxx&quot; not found">The requested
method was not found.</error>
  <error code="114" message="Invalid SOAP envelope">The SOAP envelope send in
the request could not be parsed.</error>
  <error code="115" message="Invalid XML-RPC Method Call">The XML-RPC request
document could not be parsed.</error>
</errors>
</rsp>
```

Note specifically that the following:

```
<method name="flickr.photos.geo.setLocation" needslogin="1" needssigning="1"
        requiredperms="2">
```

confirms what we had surmised—that it needs authorization and signing because it requires a minimum permission level of `write`. Compare that to what we would get for `flickr.photos.search`, which is the method that we have used throughout this chapter as an easy place to start in the API:

```
<method name="flickr.photos.search" needslogin="0" needssigning="0"
        requiredperms="0">
```

These reflection methods give rise to many interesting possibilities, especially to those of us interested in the issue of automating and simplifying the way we access web APIs. Methods in the API are both similar and different from the other methods. It would be helpful to be able to query the API with the following specific questions:

* What are all the methods that do not require any permissions to be used?
* Which methods need to be signed?
* What is an entire list of all arguments used in the Flickr API? Which method uses which argument? Which methods have in common the same arguments?

Caution These reflection methods in the Flickr API are useful only if they are kept up-to-date and provide accurate information. In working with the reflection APIs, I have run into some problems (for example, `http://tech.groups.yahoo.com/group/yws-flickr/message/3263`) that make me wonder the degree to which the reflection methods are a first-class member of the APIs.

Querying the Flickr Reflection Methods with PHP

As a first step toward building a database of the Flickr API methods that would support such queries, I wrote the following PHP script to generate a summary table of the API methods. First there is a `flickr_methods.php` class that has functions to read the list of methods using `flickr.methods.getMethods` and, for each method, convert the data from

flickr.reflection.getMethodInfo into a form that can be serialized and unserialized from a local file.

```
<?php
# flickr_methods.php
# can use this class to return a $methods (an array of methods) and
$methods_info --
# directly from the Flickr API or via a cached copy

class flickr_methods {

  protected $api_key;

  public function __construct($api_key) {
    $this->api_key = $api_key;
  }

  public function test() {
    return $this->api_key;
  }

# generic method for retrieving content for a given url.
  protected function getResource($url){
    $chandle = curl_init();
    curl_setopt($chandle, CURLOPT_URL, $url);
    curl_setopt($chandle, CURLOPT_RETURNTRANSFER, 1);
    $result = curl_exec($chandle);
    curl_close($chandle);

    return $result;
  }

# return simplexml object for $url if successful with specified number of
retries
  protected function flickrCall($url,$retries) {
    $success = false;
    for ($retry = 0; $retry < $retries; $retry++) {
      $rsp = $this->getResource($url);
      $xml = simplexml_load_string($rsp);
      if ($xml["stat"] == 'ok') {
        $success = true;
        break;
      }
    } // for
    if ($success) {
      return $xml;
    } else {
      throw new Exception("Could not successfully call Flickr");
    }
  }

# go through all the methods and list

  public function getMethods() {
```

```
    // would be useful to return this as an array (later on, I can have another
    // method to group them under common prefixes.)

    $url =
"http://api.flickr.com/services/rest/?method=flickr.reflection.getMethods
&api_key={$this->api_key}";
    $xml = $this->flickrCall($url, 3);
    foreach ($xml->methods->method as $method) {
      //print "${method}\n";
      $method_list[] = (string) $method;
    }
    return $method_list;
  }

# get info about a given method($api_key, $method_name)

  public function getMethodInfo($method_name) {

    $url =
    "http://api.flickr.com/services/rest/?method=flickr.reflection.getMethodInfo
&api_key={$this->api_key}&method_name={$method_name}";
    $xml = $this->flickrCall($url, 3);
    return $xml;
  }

# get directly from Flickr the method data
# returns an array with data
  public function download_flickr_methods () {

    $methods = $this->getMethods();

    // now loop to grab info for each method

# this counter lets me limit the number of calls I make -- useful for testing
    $limit = 1000;
    $count = 0;

    foreach ($methods as $method) {

      $count += 1;
      if ($count > $limit) {
        break;
      }

      $xml = $this->getMethodInfo($method);
      $method_array["needslogin"] = (integer) $xml->method["needslogin"];
      $method_array["needssigning"] = (integer) $xml->method["needssigning"];
      $method_array["requiredperms"] = (integer) $xml->method["requiredperms"];
      $method_array["description"] = (string) $xml->method->description;
      $method_array["response"] = (string) $xml->method->response;
    // loop through the arguments
      $args = array();
      foreach ($xml->arguments->argument as $argument) {
```

```
        $arg["name"] = (string) $argument["name"];
        $arg["optional"] = (integer) $argument["optional"];
        $arg["text"] = (string) $argument;
        $args[] = $arg;
      }
      $method_array["arguments"] = $args;

    // loop through errors
      $errors = array();
      foreach ($xml->errors->error as $error) {
        $err["code"] = (string) $error["code"];
        $err["message"] = (integer) $error["message"];
        $err["text"] = (string) $error;
        $errors[] = $err;
      }
      $method_array["errors"] = $errors;

      $methods_info[$method] = $method_array;
    }

    $to_store['methods'] = $methods;
    $to_store['methods_info'] = $methods_info;
    return $to_store;

  } // download_Flickr_API

# store the data
  public function store_api_data($fname, $to_store) {

    $to_store_str = serialize($to_store);
    $fh = fopen($fname,'wb') OR die ("can't open $fname!");
    $numbytes = fwrite($fh, $to_store_str);
    fclose($fh);
  }

# convenience method for updating the cache
  public function update_api_data($fname) {

    $to_store = $this->download_flickr_methods();
    $this->store_api_data($fname,$to_store);
  }

# restore the data

  public function restore_api_data($fname) {

    $fh = fopen($fname,'rb') OR die ("can't open $fname!");
    $contents = fread($fh, filesize($fname));
    fclose($fh);
    return unserialize($contents);

  }

} //flickr_methods
```

This form of serialization in the `flickr_method` class provides some basic caching so that you don't have to make more than 100 calls (one for each method) each time you want to display a summary table—which is what the following code does:

```
<?php

  require_once("flickr_methods.php");
  $API_KEY = "[API_KEY]";

  $fname = 'flickr.methods.info.txt';

  $fm = new flickr_methods($API_KEY);

  if (!file_exists($fname)) {
    $fm->update_api_data($fname);
  }
  $m = $fm->restore_api_data($fname);

    $methods = $m["methods"];
    $methods_info = $m["methods_info"];

    header("Content-Type:text/html");
    echo '<?xml version="1.0" encoding="utf-8"?>';
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Flickr methods</title>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
  </head>
  <body>
    <table>
      <tr>
        <th>method name</th>
        <th>description</th>
        <th>needs login</th>
        <th>needs signing</th>
        <th>permissions</th>
        <th>args (mandatory)</th>
        <th>args (optional)</th>
      </tr>
<?php
  foreach ($methods_info as $name=>$method) {
    $description = $method["description"];
# calc mandatory and optional arguments
    $m_args = "";
    $o_args = "";
    foreach ($method["arguments"] as $arg){
      //print "arg: {$arg['name']}\n";
      //print_r ($arg);
      // don't list api_key since it is mandatory for all calls
      if ($arg['name'] != 'api_key') {
        if ($arg["optional"] == '1') {
```

```
          $o_args .= " {$arg['name']}";
        } else {
          $m_args .= " {$arg['name']}";
        }
      } //if
    }
    print <<<EOT
      <tr>
        <td>
          <a href="http://www.flickr.com/services/api/{$name}.html">{$name}</a>
        </td>
        <td>{$description}</td>
        <td>{$method["needslogin"]}</td>
        <td>{$method["needssigning"]}</td>
        <td>{$method["requiredperms"]}</td>
        <td>{$m_args}</td>
        <td>{$o_args}</td>
      </tr>
EOT;
  }
?>
    </table>
  </body>
</html>
```

What Else Can Be Done with Reflection?

There's certainly a lot more you can do with this code. Let me suggest a few ideas:

* Store the data in a database (relational or XML) that can support a query language for making the queries that I listed earlier. (A poor person's approach is to copy the output of the script into a spreadsheet and work from there.)
* Create your own version of the Flickr API Explorer, perhaps as a desktop application, to help you learn about pieces of the API as you have specific questions.
* Use the `reflection` methods as the basis of a new third-party API wrapper that is able to update itself as the API changes.

Note In all the examples I have shown of the Flickr API, I have used HTTP `GET` because none of the examples so far has required any `write` or `delete` permissions. If your calls do require `write` or `delete` permissions, you must issue your Flickr call with HTTP `POST`.

Request and Response Formats

So far in this chapter, I have limited myself to one particular way of formulating a request to call a Flickr API method and the corresponding default format for the response. The Flickr API actually supports three different ways of packaging a request and five different formats for the response. In this section, I describe the choices you

have with respect to request and response formats. Understanding these choices will help you make sense of APIs other than Flickr's since you will face similar choices in working with them.

Regardless of the request or response format used, the Flickr API rests on HTTP. Hence, we need to remember that making a Flickr API call involves two steps, which is a reflection of the request and response pattern of the underlying HTTP protocol of the API:

* You formulate an HTTP *request* corresponding to API method and parameters you want to use. With Flickr, you have a choice of three formats for the request format: REST (what we have used so far), XML-RPC, and SOAP.
* You process the HTTP *response* that includes a payload that is by default XML but that also can contain JavaScript (JSON) or PHP (input for the PHP `unserialize` method).

Note Although web services are not necessarily tied to HTTP (for instance, SOAP can be bound to SMTP), HTTP is the only transport protocol supported for the Flickr API. However, the vast majority of web services used, especially for mashups, are made over HTTP. Hence, I don't cover the use of transport protocols other than HTTP in this book.

Flickr supports three different request formats to call the methods of the API (REST, SOAP, and XML-RPC):

* The REST request format, the simplest one to work with, is similar conceptually and practically to submitting a request through an HTML form. (That is, you submit a request through either HTTP `GET` or `POST` and use named parameters.) In the simplest cases, that could be equivalent to setting parameters for a URL to which you get back some XML that you can parse. Think about the examples I have presented so far to confirm that this is what has been happening. For Flickr, I recommend starting with its REST request format.

Note In Chapter 7, I revisit and refine the term REST. What Flickr calls the REST approach is a commonly used pattern of structuring web services but is more accurately described as a REST-RPC hybrid.

---

* SOAP has an envelope around the request and enables higher levels of abstraction, but it is more complicated and typically takes more specialized libraries and tools to deal with than REST. We will return to this subject in the next chapter, both in the context of Flickr's SOAP request format and in other APIs' SOAP interfaces. SOAP is an important web services technique, especially among folks who use web services for enterprise work.[4]

Note In version 1.1 of SOAP, SOAP is an acronym for Simple Object Access Protocol. Version 1.2 of the SOAP specification indicates that SOAP is no longer an acronym.

* XML-RPC was, in many ways, the proto-SOAP. It's most convenient to use XML-RPC from a library, of which there are many in a variety of languages.

There are current five different formats for Flickr responses: the three corresponding default response formats (REST, XML-RPC, SOAP) and two specialized response formats (`json` and `php_serial`). In other words, a REST-formatted request generates by default a REST-formatted response. You can change the format for the response by using the `format` parameter.

The default behavior of tying the request and request format is typical for web APIs. With the exception of the REST-to-JSON pairing, which we will return to in our discussion of Ajax programming in Chapter 8, the ability to decouple the request format from the response format is unusual. For instance, with the Flickr API, you issue a SOAP-formatted request that asks for a REST-formatted response. I'm not aware of any standard SOAP libraries that can handle such a pairing.

4. http://en.wikipedia.org/wiki/SOAP

You can see for yourself these five formats in action through a simple REST-formatted request:

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

where `format` = `rest`, `xmlrpc`, `soap`, `json`, `php_serial`, or blank (for the default response format).

Flickr Authorization

Authentication is a bit tricky to follow, and ultimately you may want to leave the details to one of the Flickr API kits (covered later in the chapter). However, you may still be interested in working through the details at least once so that you know what's going on below the hood before you use someone else's library. Besides, there will be other authentication schemes out there besides Flickr's that you will want to use. Getting a solid handle on Flickr's authentication scheme is good preparation for more quickly understanding those other authentication schemes.

You can find the specification for Flickr authentication here:

```
http://www.flickr.com/services/api/auth.spec.html
```

There are three types of authentication cases to handle with Flickr:

* Web applications
* Desktop applications
* Mobile applications

Each scheme is different because of the differing natures of each type of application. For example, the author of a web application can configure it to have a URL through

which Flickr would be able to communicate. It's hard to guarantee that a desktop application would have a URL through which such communication could happen. In this book, I cover only the specific case of authentication for web applications. Once you understand this case, you will be able to understand the others without much problem.

Three parties are involved in the authentication dance:

* Flickr
* A third-party application that is using the Flickr API and that I refer to as the *app*
* A person who is both a user of the app and a Flickr user

Authorization is required when an app is calling a Flickr method that requires a permission level of `read`, `write`, or `delete`—anything but none. Through authorization, the app is granted a specific *permission* level by the *user* to access the Flickr API on the user's behalf. Flickr creates a *token* that ties a specific app with a specific user and a specific permission level to embody this authorization act. The authentication dance is all about how that token gets created, used in conjunction with specific API calls, and can be managed and possibly revoked by the user. The details are a bit complicated because this process must also fulfill certain design criteria, which you can surmise from how the authorization scheme is designed:

* The three parties need to be reliably and securely identified and associated in the process of authorization.
* The user must be able to undo an authorization act given to a specific app.
* The protocol must be done using HTTP and not HTTPS. That is, all the parameters being passed are visible to potential third-party interlopers. In other words, knowledge of the token itself should not allow another app to have the token's permissions.
* The app should not need to know anything *a priori* about a person's Flickr identity to secure permission.

Why Passing Passwords Around Doesn't Work Too Well

The current Flickr authorization scheme is not the first one it used. In the early days of Flickr, users granted the power to an app to act on their behalf by giving the apps their Flickr username and password. Doing so meant that in order to revoke an app's permission, users would have to change their Flickr password. Of course, doing that would also instantly revoke permissions of other third-party apps with knowledge of the user's password. The new authorization scheme is meant to correct obvious problems with the old scheme. Why should you as a user have to use your Flickr password for anything other than your dealings with Flickr? Why should revoking permissions to one app mean revoking permissions to other apps?

Authorization for Web Apps

Let's now look at the authorization scheme used with Flickr. We first need to set up some permissions.

Setting Up the Example of Lois and ContactLister

Let's now get down to the details of the process of authentication for web-based applications, keeping the authorization design criteria in mind. Let's have a specific example in mind. The app, which I will call ContactLister, displays the list of contacts for a given Flickr user. It specifically uses the `flickr.contacts.getList` method, which requires authorization with `read` permission. (A Flickr user's contacts list is private.) Let's also make up a hypothetical user called Lois.

Basic Lesson: Flickr Needs to Mediate the Authorization Dance

For ContactLister to get permission from Lois, why couldn't ContactLister just directly display a screen asking Lois to give it permission to read her contact list—and then relay that fact to Flickr? For starters, how does ContactLister prove to Flickr that Lois did in fact give ContactLister permission to access her photos? In the old days of Flickr, ContactLister would have Lois's Flickr username and password. At that time, ContactLister might as well have been Lois since it's the Flickr username/password that Flickr used to authenticate a user.

The solution that Flickr came up with is based on that Flickr needs to establish unambiguously that Lois is with full knowledge of (that is, not being tricked into) giving ContactLister (and not some other third-party app) `read` (and not some other) permission. To do that, Flickr needs to mediate communication between Lois and ContactLister.

Step 1: ContactLister Directs Flickr to Ask Lois for Permission

So instead of ContactLister directly prompting Lois for permission, ContactLister directs Flickr to prompt Lois for `read` permission by formulating the following Flickr URL that it directs Lois to:

```
http://flickr.com/services/auth?api_key={api_key}&perms={perms}&api_sig={api_sig
}
```

Let's look at the various arguments. You are familiar with the `api_key`; `perms` would be set to `read` in this circumstance.

Signing a Call: How Does ContactLister Create and Send One?

The part that is new in this chapter is the `api_sig`. It is the act of calculating the `api_sig` and attaching it to method calls in the Flickr API that we refer to as *signing* the call. The purpose of signing a call is to reliably establish the identity of the signer, the one formulating the URL. Why isn't the `api_key` enough to establish the identity of the caller? In some circumstances, it would be if no one but the author of ContactLister and Flickr knew this `api_key`. On another level, Flickr API keys are sent unencrypted every time a call is made to the Flickr API, akin to passwords being sent in plain text. Hence, the `api_key` alone is an insufficient foundation for signing this call. You shouldn't be able to easily fake a signature.

When you sign up for a Flickr API key, in addition to getting a `key`, you get a corresponding string: `secret`. As the name implies, you are supposed to keep `secret` secret so that in theory only you and Flickr know it. Go to `http://www.flickr.com/services/api/keys/` to see your own keys and secrets.

ContactLister has to use this secret to calculate the `api_sig` and thereby sign the call. The `api_sig` is calculated according to the following algorithm for any Flickr API call:

* Make a signature string that starts with the secret followed by a concatenation of all the name/value pairs of the arguments to be passed to Flickr, sorted alphabetically by name—excluding the `api_sig` but including `method`. The values need to UTF-8 encoded but not URL-encoded.
* The `api_sig` is then the hexadecimal digest of the md5 hash of the signature string.

The following is a Python function that takes a secret and a dictionary of name/value pairs and returns the corresponding `api_sig`:

```
def calcSig(secret,params):
    import md5
    l = params.keys()
    l.sort()
    hash = ''
    for key in l:
        hash += str(key) + params[key].encode('utf-8')
    hash = secret + hash
    api_sig = md5.new(hash).hexdigest()
    return api_sig
```

Let's first run through a concrete example and then discuss how this process constitutes signing the call. Consider the following sample key and secret:

* Key: `020338ddabd2f41ae7ce9413a8d51429`
* Shared secret: `f0fc085289c7677a`

The signature string is then as follows:

```
{secret}api_key{api_key}perms{perms}
```

which is as follows:

```
f0fc085289c7677aapi_key{api_key}permsread
```

The md5 hexadecimal digest of the string is then as follows:

```
f9258a76e4ad3cb5fa40bd8b0098d119
```

Therefore, the signed call is as follows:

```
http://flickr.com/services/auth?api_key={api_key}&perms=read
&api_sig=f9258a76e4ad3cb5fa40bd8b0098d119
```

What Flickr Makes of the Signed Call

So when ContactLister directs Lois to go to this URL, Flickr first determines the integrity of this call by performing the same signature calculation as ContactLister did in the first place: find the secret that corresponds to the `api_key`, sort all the parameters by key (except for the `api_sig` parameter), form the signature string, and then compare it to the value of the `api_sig` parameter. If two match up, then Flickr can conclude the call

did indeed come from ContactLister because presumably the author of ContactLister is the only one other than Flickr who knows the key/secret combination.

You might ask, why can't someone take the `api_sig` from the call and reverse the md5 calculation to derive the secret? Although it's straightforward to calculate the md5 hash of a string, it's much more difficult computationally to go in the other direction. For the purposes here, you should think of this reverse direction for md5 as practically—but not theoretically—impossible. Moreover, using md5 makes it difficult to change the parameters of the call. If you change, say, `perms=read` to `perms=delete`, you get a different `api_sig`, which is very hard to calculate without knowing `secret`.

Note md5, it turns out, does have limitations as a cryptographic hash function. Researchers have demonstrated how to take an md5 hash and create another string that will give you the same md5 hash. Can this weakness be used to issue fake Flickr calls? I don't know; see `http://en.wikipedia.org/wiki/MD5` for more information.

Step 2: Flickr Asks Lois for Permission on Behalf of ContactLister

At any rate, assuming a properly signed call to `http://flickr.com/services/auth`, Flickr now knows reliably that it is indeed ContactLister asking for `read` permission. Remember, though, that the end goal, a token, ties three things together: an app, a permission level, and a user. The call reliably ties the app and permission together for Flickr. However, the call has no explicit mention of a user at all. There's no parameter for `user_id`, for instance.

That ContactLister doesn't have to pass to Flickr anything about Lois's Flickr account is a virtue—not a problem. Why should a third-party app have to know anything *a priori* about a person's relationship to Flickr? So, how does Flickr figure out the user to tie to the request by ContactLister for the `read` permission? The fact is that it's Lois—and not someone else—who uses the authorization URL:

```
http://flickr.com/services/auth?api_key={api_key}&perms=read
&api_sig=f9258a76e4ad3cb5fa40bd8b0098d119
```

When Lois loads the authorization URL in her browser, Flickr then determines the user in question. If Lois is logged in, then Flickr knows the user in question is Lois. If no one is logged in to Flickr, then Lois will be sent through the login process. In either case, it's Flickr that is figuring out Lois's identity as a Flickr user and taking care of her authenticating to Flickr. In that way, Flickr can establish to its own satisfaction the identity of the user involved in the authorization dance—rather than trusting ContactLister to do so.

Now that Flickr knows for sure the identity of the app, the permission level requested, and the user involved, it still needs to actually ask Lois whether it's OK to let ContactLister have the requested `read` permission. If Lois had not already granted ContactLister such permission, then Flickr presents to Lois a screen that clearly informs her of ContactLister's request. The fact that such a display comes from Flickr instead of ContactLister directly should give Lois some confidence that Flickr can track what ContactLister will do with any permissions she grants to it and thereby hold the authors of ContactLister accountable.

Step 3: Flickr Lets ContactLister Know to Pick Up a Token

Assuming that Lois grants ContactLister read permission, Flickr must now inform ContactLister of this fact. (Remember, the permission granting is happening on the Flickr site.) Flickr communicates this authorization act by sending the HTTP `GET` request to the `callback-URL` for ContactLister with what Flickr calls a *frob*. Flickr knows the `callback-URL` to use because part of registering a web application to handle authorization is specifying a callback URL at the following location:

```
http://www.flickr.com/services/api/keys/{api-key}/
```

where the `api-key` is that for the app. In other words, ContactLister must handle a call from Flickr of the following form:

```
callback-URL?frob={frob}
```

A frob is akin to a session ID. It lets ContactLister know that some form of authorization has been granted to ContactLister. To actually get the token that ContactLister needs to use the requested `read` permission, ContactLister needs to use `flickr.auth.getToken` to exchange the frob for the token. Frobs aren't meant to be the permanent representation of an authorization act. Frobs expire after 60 minutes or after `flickr.auth.getToken` is used to redeem the frob for a token. This exchange ensures that ContactLister receives a token and that Flickr knows that ContactLister has received the token. Note that `flickr.auth.getToken` is also a signed call with two mandatory arguments: `api_key` and `frob`—in addition to `api_sig`, of course. The returned token is expressed in the following form (quoting from `http://www.flickr.com/services/api/flickr.auth.getToken.html`):

```
<auth>
  <token>976598454353455</token>
  <perms>write</perms>
  <user nsid="12037949754@N01" username="Bees" fullname="Cal H" />
</auth>
```

Note that it's the token that tells ContactLister the details of what is being authorized: the Flickr user and the permission granted. Now, ContactLister knows the Flickr identity of Lois—without ever needing Lois to tell ContactLister directly.

Step 4: ContactLister Can Now Make an Authorized and Signed Call

ContactLister can now actually make the call to `flickr.contacts.getList`. How so? In addition to signing a call to `flickr.contacts.getList`, ContactLister adds the appropriate authorization information by adding the following argument to the call and signing it appropriately:

```
auth-token={token}
```

We should note moreover that Lois, like all users, can revoke any permission she had previously granted here:

```
http://flickr.com/services/auth/list.gne
```

It's nice for Lois to know that she doesn't have to convince ContactLister to stop accessing her account. She just tells Flickr.

Implementation of Authorization in PHP

That's the narrative of how to do Flickr authorization for web applications. Now let's look at it implemented in PHP. There are two pieces of code. The first generates the authorization URL. (To use it, use your own API key and secret.)

```
<?php
  $api_key = "";
  $secret = "f0fc085289c7677a";
  $perms = "read";

  function login_link($api_key,$secret,$perms) {
    # calculate API SIG
    # sig string = secret + [arguments listed alphabetically name/value --
    # including api_key and perms]

    $sig_string = "{$secret}api_key{$api_key}perms{$perms}";
    $api_sig = md5($sig_string);

    $url = "http://flickr.com/services/auth?api_key={$api_key}&perms={$perms}
&api_sig={$api_sig}";
    return $url;
  }

  $url = login_link($api_key,$secret,$perms);
?>
<html>
  <body><a href="<?php print($url);?>">Login to Flickr</a></body>
</html>
```

To confirm that you have things set up correctly, if you run the app, you should get a prompt from the Flickr site asking for access (see Figure 6-2).[5]

Insert 858Xf0602.tif

Figure 6-2. Flickr authorization screen. (Reproduced with permission of Yahoo! Inc. ® 2007 by Yahoo! Inc. YAHOO! and the YAHOO! logo are trademarks of Yahoo! Inc.)

The second piece of code is the authentication-handling script whose URL is the callback URL registered to the API key. It reads the frob, gets the token, and then lists the contacts of the user (a type of access that demonstrates that authorization is working, since without authorization, an app will not be able to access a user's contact list). To try this yourself, you will need to create this file and then enter its URL in the Callback URL field of your app's key configuration screen at Flickr:[6]

5. http://examples.mashupguide.net/ch06/auth.php

6. http://examples.mashupguide.net/ch06/auth_cb.php

```
<?php
##insert your own Flickr API KEY here
$api_key = "[API_KEY]";
$secret = "[SECRET]";

$perms = "read";
```

```
$frob = $_GET['frob'];

function getResource($url){
  $chandle = curl_init();
  curl_setopt($chandle, CURLOPT_URL, $url);
  curl_setopt($chandle, CURLOPT_RETURNTRANSFER, 1);
  $result = curl_exec($chandle);
  curl_close($chandle);

  return $result;
}

function getContactList($api_key, $secret, $auth_token)  {
  # calculate API SIG
  # sig string = secret + [arguments listed alphabetically name/value --
  # including api_key and perms]; don't forget the method call

  $method = "flickr.contacts.getList";
  $sig_string =
    "{$secret}api_key{$api_key}auth_token{$auth_token}method{$method}";
  $api_sig = md5($sig_string);

  $token_url =
    "http://api.flickr.com/services/rest/?method=flickr.contacts.getList
&api_key={$api_key}&auth_token={$auth_token}&api_sig={$api_sig}";
  $feed = getResource($token_url);
  $rsp = simplexml_load_string($feed);

  return $rsp;
}

function getToken($api_key,$secret,$frob) {
  # calculate API SIG
  # sig string = secret + [arguments listed alphabetically name/value --
  # including api_key and perms]; don't forget the method call

  $method = "flickr.auth.getToken";
  $sig_string = "{$secret}api_key{$api_key}frob{$frob}method{$method}";
  $api_sig = md5($sig_string);

  $token_url =
    "http://api.flickr.com/services/rest/?method=flickr.auth.getToken
&api_key={$api_key}&frob={$frob}&api_sig={$api_sig}";
  $feed = getResource($token_url);
  $rsp = simplexml_load_string($feed);

  return $rsp;
}

$token_rsp = getToken($api_key,$secret,$frob);
$nsid = $token_rsp->auth->user["nsid"];
$username = $token_rsp->auth->user["username"];
$auth_token = $token_rsp->auth->token;
```

```
$perms = $token_rsp->auth->perms;

# display some user info
echo "You are: ", $token_rsp->auth->user["fullname"],"<br>";
echo "Your nsid: ", $nsid, "<br>";
echo "Your username: ", $username,"<br>";
echo "auth token: ", $auth_token, "<br>";
echo "perms: ", $perms, "<br>";

# make a call to getContactList

$contact_rsp = (getContactList($api_key,$secret,$auth_token));
$n_contacts = $contact_rsp->contacts["total"];
$s = "<table>";
foreach ($contact_rsp->contacts->contact as $contact) {
  $nsid = $contact['nsid'];
  $username = $contact['username'];
  $realname = $contact['realname'];
  $s = $s . "<tr><td>{$realname}</td><td>{$username}</td><td>{$nsid}</td></tr>";
}

$s = $s . "</table>";
echo "Your contact list (which requires read permission) <br>";
echo "Number of contacts: {$n_contacts}<br>";
echo $s;
?>
```

Note Uploading photos to Flickr is a major part of the Flickr API that is not covered in this book. I suggest reading the documentation (http://www.flickr.com/services/api/upload.api.html) and using one of the API kits.

Using Flickr API Kits

Once you get the hang of the APIs using REST, you'll likely get tired of using it directly in your programming. The details of authorizing users, uploading photos, and managing a cache of Flickr results (to speed up access) are not things you want to deal with all the time.

API kits in various programming languages have been written to make it more comfortable for you to use the API in your language. These tools often express the Flickr API in terms that are more natural for a given language, by abstracting data, maintaining sessions, and taking care of some of the trickier bits of the API.

You can find a list of API kits for Flickr here:

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

In this section I'll describe briefly some options of API kits for PHP. Currently, three Flickr API kits are publicized on the Flickr services page. This section shows how to set them up to do a simple example of a working program for each of the API kits. You then need to figure out which is the best to use for your given situation.

SETTING UP INCLUDE_PATH AND FLICKR KEYS

Whenever you use third-party libraries, you need to ensure that your PHP path (the `include_path` variable) is set properly so that your PHP code can find your libraries. If you have access to `php.ini`, by all means use it. You can also use the `ini_set()` function in PHP to set your `include_path` variable within your code. In the following code, I assume that `include_path` is properly set.

Also, it's convenient to store your Flickr key and secret in an external file that you can then include. For the following examples, I have a file named `fickr_key.php` containing the following:

```
<?php
define('API_KEY', '[YOUR_KEY]');
define('API_SECRET', '[YOUR_SECRET]');
?>
```

PEAR::Flickr_API

This kit,[7] written by Cal Henderson, is the earliest and simplest of the API kits. To try it on your hosting platform, make sure you have PEAR installed, and install the library using the following command:

7. http://code.iamcal.com/php/flickr/readme.htm

```
pear install -of http://code.iamcal.com/php/flickr/Flickr_API-Latest.tgz
```

Here's a little code snippet to show you its structure:

```
<?php
  include("flickr_key.php");
  require_once 'Flickr/API.php';
  # create a new api object
  $api =& new Flickr_API(array(
      'api_key'  => API_KEY,
      'api_secret' => API_SECRET
    ));

  # call a method

  $response = $api->callMethod('flickr.photos.search', array(
      'tags' => 'flower',
      'per_page' => '10'
    ));

  # check the response

  if ($response){
    # response is an XML_Tree root object
    echo "total number of photos: ", $response->children[0]-
>attributes["total"];
  }else{
    # fetch the error
    $code = $api->getErrorCode();
    $message = $api->getErrorMessage();
  }
```

```
?>
```

Why might you want to use `PEAR::Flickr_API`? It's a simple wrapper with some defining characteristics:

* There's not much of an abstraction of the method calls. You pass in the method name. The advantage is that the API will not be out-of-date with the addition of new Flickr methods. The disadvantage is that one can imagine abstractions that are more idiomatic PHP.
* You pass in the API key when creating a new `Flickr_API` object.
* The response is an `XML_Tree` root object.[8]

My conclusion is that it makes sense to use one of the newer, richer PHP API kits: `phpFlickr` or `Phlickr`; also, more people are actively working on them.

phpFlickr

You can find Dan Coulter's toolkit at `http://phpflickr.com/`. It is written in PHP 4, which is currently an advantage, since PHP 5 is not always readily available. Moreover, there seems to be a continued active community around `phpFlickr`. To install and test the library, following these steps:

8. http://pear.php.net/package/XML_Tree—this package has been superseded by XML_Serializer (http://pear.php.net/package/XML_Serializer)

1. Follow the detailed instructions at http://phpflickr.com/docs/?page=install. Download the latest ZIP file from http://sourceforge.net/projects/phpflickr. At the time of writing, the latest is the following:9

```
http://downloads.sourceforge.net/phpflickr/phpFlickr-2.1.0.tar.gz
```

or the following:

```
http://downloads.sourceforge.net/phpflickr/phpFlickr-2.1.0.zip
```

2. In theory, PEAR should let me install it, but I was not been able to get it to install `phpFlickr`.[10] Uncompress the file into a directory so that you can include it. I put it in a non-PEAR `phplib` directory and renamed the file to `phpFlickr`.

3. Copy and paste the following code as a demonstration of working code:

```
<?php

include("flickr_key.php");
require_once("phpFlickr/phpFlickr.php");

$api = new phpFlickr(API_KEY, API_SECRET);

#
# Get user's ID
#
$username = 'Raymond Yee';
if (isset($_GET['username']))
    $username = $_GET['username'];
```

```
$user_id = $api->people_findByUsername($username);
$user_id = $user_id['id'];

print $user_id;

?>
```

Let's see how `phpFlickr` works:

* The constructor has three arguments: the mandatory API key and two optional parameters, `secret` and `die_on_error` (a Boolean for whether to die on an error condition). Remember that you can use the `getErrorCode()` and `getErrorMsg()` functions of `$api`.[11]
* This is from the official documentation:
    * Apparently, all of the API methods have been implemented in the `phpFlickr` class.
    * To call a method, remove the `flickr.` part of the name, and replace any periods with underscores. You call the functions with parameters in the order listed in the Flickr documentation—with the exception of `flickr.photos.search`, for which you pass in an associative array.
    * To enable caching, use the `phpFlickr::enableCache()` function.

Because the naming convention of `phpFlickr`, which is closely related to that of the Flickr API, you can translate what you know from working with the API pretty directly into using `phpFlickr`.

9. http://sourceforge.net/project/showfiles.php?group_id=139987&package_id=153541&release_id=488387

10. pear install -of http://downloads.sourceforge.net/phpflickr/phpFlickr-2.1.0.tar.gz gets me "Could not extract the package.xml file from /home/rdhyee/pear/temp/download/phpFlickr-2.1.0.tar.gz."

11. http://phpflickr.com/docs/?page=install

Phlickr

`Phlickr` requires PHP 5 and is not just a facile wrapper around the Flickr API; it provides new classes that significantly abstract the API. There are significant advantages to this approach; if the abstraction is done well, you should be able to program Flickr in a more convenient and natural method in the context of PHP 5 (for example, you can work with objects and not XML, which you can then turn into objects). The downside is that you might need to juggle between the Flickr API's way of organizing Flickr functionality and the viewpoint of the `Phlickr` author. Moreover, if Flickr adds new methods, there is a greater chance of `Phlickr` breaking as a result—or at least not being able to keep up with such changes.

The home page for the project is as follows:

```
http://drewish.com/projects/phlickr/
```

You can get the latest version of `Phlickr` from here:

```
http://sourceforge.net/project/showfiles.php?group_id=129880
```

The following code is a simple demonstration of `Phlickr` in action—it uses the `flickr.test.echo` method:

```
<?php
ini_set(
  'include_path',
  ini_get( 'include_path' ) . PATH_SEPARATOR . "/home/rdhyee/pear/lib/php"
);

require_once 'Phlickr/Api.php';

#insert your own Flickr API KEY here
define('FLICKR_API_KEY', '[API-KEY]''');
define('FLICKR_API_SECRET', '[SECRET]''');

$api = new Phlickr_Api(FLICKR_API_KEY, FLICKR_API_SECRET);
$response = $api->ExecuteMethod(
  'flickr.test.echo',
  array('message' => 'It worked!'));

print "<hi>{$response->xml->message}</h1>";
?>
```

`http://drewish.com/projects/phlickr/docs/` documents the objects of the library. To learn more about `Phlickr`, buy and read *Building Flickr Applications with PHP* by Rob Kunkle and Andrew Morton (Apress, 2006). Andrew Morton is the author of `Phlickr`.

Note `Phlickr` must be in a folder called exactly `Phlickr` for operating systems (such as Linux) whose filenames are case-sensitive.

Limitations of the Flickr API

The Flickr API is extensive. The methods of the Flickr API are a fairly stable, well-supported way for your program to access data about most resources from Flickr. As one would expect, the functionality of the Flickr API overlaps strongly with that of the Flickr UI—but the two are not identical. There are currently things you can do in the UI that you can't do in the API. For example:

* Although you can access a Flickr group's photo pool, you can't read or write to the group discussions with the API (though you can get at the latest comments in a group discussion through Flickr feeds).
* You can't add, delete, and configure a weblog for your Flickr account including layout and settings with the API.
* You can't add or delete contacts via the API.
* You can't delete your Flickr account with the API or do most of the account management elements such as changing your e-mail or using a different Yahoo! ID for this Flickr account.

* There is no support for Flickr collections in the API.
* I don't think there is currently support for tag clusters in the API (`http://tech.groups.yahoo.com/group/yws-flickr/message/1596`).

Some of limitations of the API are probably intentional design decisions that are unlikely to change (such as not being able to programmatically delete your entire account). Other discrepancies reflect that new features in Flickr tend to show up first in the UI and then in the API. I would guess, for instance, that there will eventually be support for Flickr collections in the API.

I will point out another class of differences between the API and UI. There is, however, some information from Flickr that is available from both the UI and the API—but that is easier to derive from screen-scraping the UI and through using the API. Take this, for example:

```
http://www.flickr.com/photos/{user-id}/archives/
```

This lists for every year and month the number of photos that the user has taken or uploaded. Accessing this information from the UI involves one HTTP `GET` and screen-scraping the HTML. In contrast, generating the same dataset using the Flickr API requires calculating Unix timestamps for the beginnings and ends of months (for the time zone of the user, which is not available via the API) so that you can feed those time boundaries to `flickr.photos.getCounts`.

What's the point here? Although the API provides the flexibility to calculate the number of photos taken or uploaded between any two arbitrary times, the UI for the archives provides a count of the photos for a very useful default case (that by month), which turns out to require a bit of work to get from the API. In other words, the UI of an application gives insight into what the mainstream use cases for the API are.

I've found such examples about limitations of APIs with respect to the UI a bit surprising at first. I would have expected a given functionality to be purposely excluded from the API (because of a policy decision) or easier to programmatically access via the UI—but not harder than screen-scraping using the API. Otherwise, there's a disincentive to use the API in that case.

Summary

If you have read this long chapter and studied the examples in depth, you should now be able to see both the conceptual heart of the Flickr API—a bunch of HTTP requests that look like HTML form submissions and responses that by default return nice-to-parse XML—and the complexities that arise when dealing with various cases (different request and response formats, authorization, and the need to abstract the API when using them in practice). I'm a big believer in learning as much as you can from the API before taking on authorization. You can use simple calls to solidify your understanding of HTTP and XML processing. Then you can move on to the more complicated cases when you are ready.

If you want to make sense of the Flickr API as a whole, focus on tackling specific problems that get you into exploring parts of the API. The reflection methods, though, do give you the potential to computationally support your understanding of the API as well as make more robust libraries for interacting with Flickr.

Understanding the underlying details of Flickr authorization is something you don't have to deal with if you don't want to—turn to your favorite API kit for help. However,

understanding it brings not only intellectual satisfaction but also enables you to better understand other authorization schemes you may encounter (such as the one for Amazon S3).

In the next chapter, we'll turn to web APIs other than Flickr. I will use the lens of the Flickr API to show you how to explore the bigger world of APIs in general.