

1

Rewriting the Web

Over the past few years, the Web has seen dramatic growth — both in the actual number of users and in the sheer volume of information available to them. As more and more people start making use of the Web, more web sites appear. The volume of available information is overwhelming — and more importantly, the number of ways in which people wish to use the information available to them is increasing rapidly.

Service providers have come to realize that they simply cannot keep up with demand — they cannot hope to satisfy all of the people all of the time. The audience out there is so large and diverse that it is no longer possible to build all the systems their customers desire. Realizing this, some service providers have addressed the problem in a rather innovative way — instead of closely guarding their corporate data and systems, they have taken the bold step of opening them up to the world. Now, if the service you desire isn't available, the tools are there to take an existing service and enhance it yourself. The distinction between service provider and service consumer is blurring. Consumers are taking control of what they see and sometimes even becoming providers themselves. Companies like Google, Yahoo!, Amazon.com, and Flickr are among the first to embrace this new, open Internet, but where they tread, others will surely follow. This first chapter takes a brief wander through some of the different technologies that have made this shift possible.

Web 2.0: Power to the People

Web 2.0 is the new, open Internet — the latest, and indeed greatest, incarnation of the World Wide Web. But what does Web 2.0 actually mean? The version number suggests a new iteration of technology, a new software release, but Web 2.0 isn't actually a technological advancement. There is no new technology involved — all those Web 2.0 systems that you hear people talking could have been built several years ago. So what has changed?

The change has been a social one. At the core of Web 2.0 is the idea of collaboration, the common thread across Web 2.0 systems being the online sharing of information and the networking of human beings as well as computers. In the world of Web 2.0, web sites are no longer stand-alone entities. Instead of simply displaying their wares to passing visitors, they become data centers — feeding information to other applications on the web. The information is not only shared, it is enriched. Users of shared data are encouraged to add to it, to annotate it. They identify points of

interest on Google Maps, add tags to photos on Flickr, and write book reviews on Amazon.com. Users help identify connections between pieces of data — they place their photos on maps of the world, they tag related links in del.icio.us and they create lists of related items on Amazon.com. Each connection made is stored away — an extra data point is created.

By encouraging both the sharing and the enhancement of data, the overall value of those data is increased. The idea that the whole is greater than the sum of its parts is central to Web 2.0, and at the very heart of it is the key element, the user community itself — the very people who enrich the data and give them value.

Remixes and Mashups

The words *remix* and *mashup* are regularly bandied about, but the two are often confused and used interchangeably. Both refer to the presentation of third-party content in a form that is different from the original, but each refers to a slightly different approach.

A *remix* is a piece of content taken from somewhere and presented in a different form. There's typically no significant modification to the content itself — only to the mode of presentation. A very simple example of creating a remix might be applying a user-defined style sheet to change the background color of a web site you are viewing, and perhaps hiding the banner advertisements at the same time. A more complex example might be building a custom web site to display photographs stored on Flickr in your own online gallery. In each case, the content is being derived from a single source, but a custom mode of presentation used.

A *mashup*, on the other hand, involves content from a variety of sources mashed together to produce a brand new dataset — one that is richer than any of the original sources on their own. For example, taking your photographs from Flickr and displaying them as an overlay onto Google Maps would be creating a mashup of Flickr content (the photos and the location information held against them) and Google's mapping data.

Remixes and mashups aren't necessarily distinct — all mashups, for example, could well be considered to be remixes, but remixes may or may not also be mashups, depending on the number of data sources they use. In practice, the distinction between the two terms is minor. In this book, when I'm discussing the projects, I'll use the term that seems most appropriate based on the number of data sources. In more general passages, it quickly becomes tedious to keep having to use the phrase "remix or mashup," so for ease of reading I'll interchangeably use one or the other of the terms. If ever the distinction is important, it will be made explicitly clear.

What Makes a Mashable System?

So, is any source of content out on the Internet mashable? Well, in theory, yes — if you can see it, you can mash it. There are, however, a number of things to consider.

Are You Allowed to Mash the Content?

If the content is owned by somebody else — and this is true in many cases — then you most likely need permission from the copyright owner before you can reuse the content. The content owner may explicitly make content available, although usually there will be restrictions on how you can use it.

On Flickr, and some other systems, the matter is slightly more complicated. The content made available by Flickr isn't typically owned by Flickr itself — it's actually the property of Flickr's users. The photos you see on Flickr are owned by millions of people across the globe. Obviously, use of your own photos is not a problem — you are free to use those where you choose — but you should be careful about how you use photos that do not belong to you. Flickr allows its members to assign different copyright licenses to photos from the Creative Commons set of licenses, and so explicitly allow their use in other applications. These licenses are discussed in more detail in Chapter 2.

When deciding what content to include in your mashup, always check that you will be using the content in ways that are permitted by the copyright owner.

How Easy Is It to Get the Content?

It's often said that if you can see the content on the web, you can write software to retrieve it — after all, that's all your web browser is doing. Strictly speaking, that's absolutely correct. The real question, however, isn't whether or not you can get the content, but rather how *easy* it is to get the content. Suppose, for example, that the piece of text you need to include in your mashup is buried in the middle of somebody else's web page. You need to retrieve and parse the HTML on the page and try to locate the piece you're interested in. If the content of the page is regularly updated, this may in itself be a fairly tricky task. Add to that the prospect of having to deal with cookies, web server authentication, and a whole bunch of other things that happen quietly behind the scenes when you are browsing the Web, and the seemingly straightforward task of getting your piece of text can quickly become a nightmare.

On the other hand, if the content is explicitly made available to you by the content provider, perhaps via a defined programmatic interface or web service, then retrieving it can be extremely straightforward.

When planning your remix or mashup, you should always consider just how you will obtain the content you will be using. In particular, you should check that it is made available in ways that will help rather than hinder you.

Mashup Architectures

From an architectural viewpoint, the most fundamental decision you have to make when starting to design your remix or mashup is where the remixing occurs. Does it occur on the server, or does it occur on the client? Quite possibly, it's a combination of the two. So how do you decide where to do the remixing? As you work through this book, you'll see many examples of both server-side and client-side mashups in action, but let's first take a look at the pros and cons of each.

For the sake of simplicity, we'll assume here that all of the remixes and mashups we discuss are web-based systems. Of course that doesn't have to be the case — you could just as easily write a native Microsoft Windows application that mashes different kinds of data together. The basic mashing principles are, however, basically the same, no matter how you choose to build and deploy your mashup.

Mashing on the Client

Building your mashup so that it executes on the user's client machine is often an attractive option. For a start, it means that you don't need any special software running on your own web server. You may not

Part I: Building a New Internet

even need a web server of your own at all. The downside, however, is clear — you are completely dependent on the end user to be using a computer capable of running your mashup. You can see a typical client-side configuration in Figure 1-1.

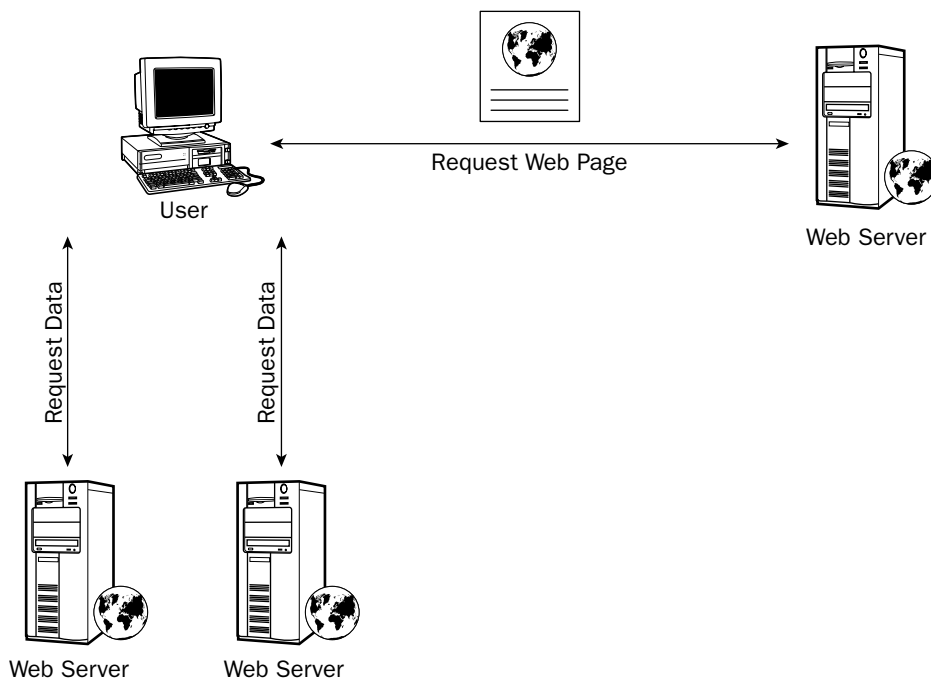


Figure 1-1

In a client-side mashup, the user first retrieves the web page containing the mashup. Typically, other code within this page then executes within the client browser and retrieves further data from other web servers on the Internet, and then integrates it into the original page, finally displaying the resulting page to the user.

For client-side mashups to work, you need to either have the end user install all the required software, or rely on it already being there. In most cases you can reasonably expect a user to have certain software available — a relatively up-to-date web browser, JavaScript, maybe even the Flash plug-in. You should remember, however, that the more demands you place on the client, the smaller the potential audience for your mashup becomes. Many users won't install extra software just so they can see your mashup — no matter how good it is. Ideally, all they should need to do is point their web browsers in the right direction.

Mashing on the Server

For maximum control and flexibility, mashing on the server is usually by far the best option. You will often have much more control as to the software available to you, installing anything that you need. The server takes control, retrieving content from the various different sources, assembling it all, and then sending the finished result back to the client. You can see all this in action in Figure 1-2.

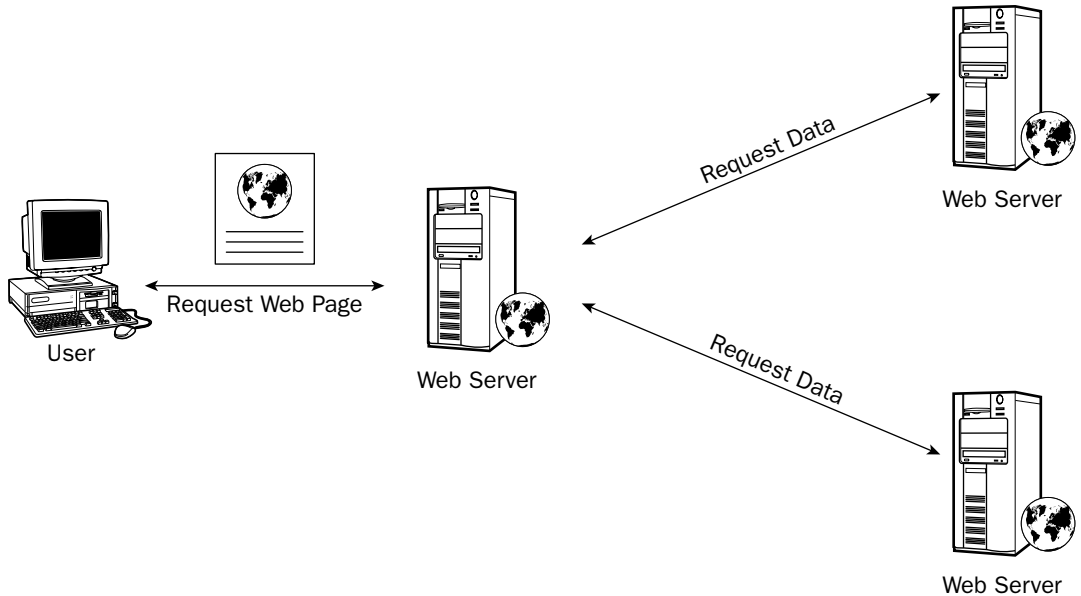


Figure 1-2

The creation of server-side mashups can be a little more complicated — you need to set up and configure all the elements you need on a server first. Despite this, the advantage of server-side mashups is clear — your users don't need any special software. All they need to view your creation is a web browser.

Understanding Mashup Tools and Technologies

Many different tools and technologies are useful in the masher's toolkit. In this next section, we take a brief look at a few of these and see how they might be of use. In a book of this size, it clearly isn't possible to provide full tutorials on each of the subjects mentioned here — each one could easily warrant a whole book in its own right, so here we can do little more than give them a passing mention. Some of these topics are also discussed in more detail in later sections of the book where they are put to use, while Appendix D contains many useful references if you want more detailed information.

HTML, XHTML, and CSS

You almost certainly already have at least a passing familiarity with this family of technologies, starting with HTML, or *Hypertext Markup Language*, which provides the foundation for the entire Web. HTML is a simple, text-based markup language used to structure documents published on the World Wide Web. The HTML standards are defined by the World Wide Web Consortium (W3C) and can all be found on the W3C web site at <http://www.w3.org/>.

Part I: Building a New Internet

XHTML, or *Extensible Hypertext Markup Language*, is a variant of HTML represented as XML (discussed later) and is the successor to HTML, which it is now starting to replace. For maximum flexibility, most new documents should be created with XHTML rather than HTML.

CSS, or *cascading style sheets*, is a mechanism for attaching style information to HTML and XHTML documents. Style information includes such things as font, color, and page layout information. The first incarnation of HTML provided very little ability to specify much in the way of style information, and decisions about how to render the document were left entirely to the browser. Before long, support for additional HTML elements and attributes was added, so that content creators can make documents that specify various elements of style.

```
<html>
  <head>
    <title>My Document</title>
  </head>
  <body bgcolor="#FF33FF">
    <h1><font face="Arial" size="2" color="#0000FF">Welcome to My
      Web Site</font></h1>
    <p>
      <font face="Arial" size="4" color="#00FF00">Hello, world!</font>
    </p>
  </body>
</html>
```

As you can imagine, maintenance of web sites made up of documents styled in this way was a nightmare—every time you wanted to change a font size, for example, you would have to go through each document, changing the size attribute everywhere it occurred. This approach also increased the complexity of the document structure enormously, resulting in a great deal of document bloat. Fortunately, CSS came along to save the day.

CSS provides complete separation of document content and document presentation. With CSS, it is possible to take the same HTML document and present it in different ways and via different media. For example, you might use one set of CSS to render an HTML document for the Web, and a different set to format your document in a way that allows it to be printed easily.

Used properly, CSS can reduce document bloat enormously. For example, with CSS, the HTML above becomes:

```
<html>
  <head>
    <title>My Document</title>
    <link href="styles.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <h1>Welcome to My Web Site</h1>
    <p>Hello, world!</p>
  </body>
</html>
```

Here is the associated CSS file:

```
body {
  color: #00FF00;
```

```
background-color: #FF33FF;
font-family: Arial;
font-size: 14pt;
}

h1 {
color: #0000FF;
font-family: Times New Roman;
font-size: 24pt;
}
```

When you consider that many pages across a web site will share the same CSS, and that the CSS files will normally only be loaded by the browser on first access (subsequent page accesses will use the CSS then stored in the browser's cache), it's easy to see that the reduced page bloat can result in dramatically improved page load times — especially on large and complex pages.

The separation of style from content provided by CSS makes creating and maintaining web sites dramatically easier. Changes to the look and feel of the entire site can be made simply by the modification of a single CSS file.

XML and XPath

XML, or *Extensible Markup Language*, is a simple, extensible text format, primarily used for transferring and exchanging data structures. Being text-based, it is easily human-readable and is widely used in interfacing with web-based systems. Along with HTML, it is derived from SGML (Standard Generalized Markup Language) and follows a similar tag-based model. XML specifications are managed by the World Wide Web Consortium and may be found on its web site.

Each XML application is free to define its own data structures, which may then be formally described within a DTD (document type definition) or XML schema. A discussion of these documents is beyond the scope of this book, but again, details can be found on the W3C web site.

Creation of XML documents is very simple — here's a sample document describing part of a book publisher's catalog:

```
<?xml version="1.0"?>
<books>
  <book>
    <title>Flickr Mashups</title>
    <author>David Wilkinson</author>
    <isbn>0470097744</isbn>
  </book>
  <book>
    <title>Google Maps Mashups</title>
    <author>Virender Ajmani</author>
    <isbn>0470097752</isbn>
  </book>
  <book>
    <title>del.icio.us Mashups</title>
    <author>Brett O'Connor</author>
    <isbn>0470097760</isbn>
  </book>
```

```
<book>
  <title>Amazon.com Mashups</title>
  <author>Francis Shanahan</author>
  <isbn>0470097779</isbn>
</book>
<book>
  <title>Yahoo! Maps Mashups</title>
  <author>Charles Freedman</author>
  <isbn>0470097787</isbn>
</book>
</books>
```

Every XML document must start with a header that declares it as XML:

```
<?xml version="1.0"?>
```

The header is then followed by a single root element—in this case `<books>`. Within the root element sits the rest of the document. In the example here, the `<books>` element contains a list of individual `<book>` elements. Each `<book>` element contains, in turn, `<title>`, `<author>`, and `<isbn>` elements.

XPath, or *XML Path Language*, enables you to address different parts of an XML document. It is a very powerful tool often used in processing and manipulating XML. XPath treats an XML document as a tree of nodes, with each element and each attribute represented by a node in that tree. An *XPath query* returns a set of nodes from the tree. A typical XPath query consists of an expression representing the path to one or more nodes. So, for example, in the sample XML document here, the XPath query

```
/books
```

will return the root `<books>` node, while the query

```
/books/book
```

returns the set of `<book>` nodes contained within `<books>`, and

```
/books/book/title
```

returns the set of `<title>` nodes.

To return the third book in the list, you would use the following XPath query:

```
/books/book[3]
```

And the last book could be found with

```
/books/book[last()]
```

If you don't want to specify the whole path to an element, you can use the following notation:

```
//title
```

— which in this example will return all the `<title>` nodes within the document—regardless of location.

In this book we only scratch the surface of XPath's capabilities, but if you want to learn how to build more complex XPath expressions, you can read more about the language and its capabilities on the W3C web site.

JavaScript and DOM

JavaScript is another scripting language, widely known for its use in web-based applications. The name JavaScript is something of a misnomer, as it bears only a superficial resemblance to the Java programming language. It first appeared in Netscape's web browser in 1995, and was incorporated into the international standard ECMA-262 as ECMAScript. Most modern browsers incorporate an ECMAScript-compatible scripting language—the one found in Microsoft's Internet Explorer is known as Jscript. JavaScript is actually a registered trademark of Sun Microsystems, but is widely used as a generic reference to the scripting language used within web browsers.

Within the scripting language, web browsers typically implement the W3C standard DOM, or *document object model*. The DOM specifies how HTML and XML documents are represented as a tree structure within the browser. The DOM specification also specifies the programmatic interface by which the various elements within the document tree can be manipulated.

JavaScript used to manipulate the DOM within a browser is often referred to as *dynamic HTML*. Here's a simple example of how you might use dynamic HTML to provide interaction on a web page.

```
<html>
  <head>
    <title>Arithmetic</title>
    <script type="text/javascript">
      function showAnswer()
      {
        var answerNode = document.getElementById('answer');
        answerNode.innerHTML = "<h2>Answer</h2><p>" + (2 + 2) + "</p>";
      }
    </script>
  </head>
  <body>
    <h1>Question</h1>
    <form onsubmit="showAnswer(); return false;">
      <p>
        What is the answer to the sum <code>2 + 2</code>?
      </p>
      <input type="submit" value="Show Answer" />
    </form>

    <div id="answer">
      The answer will appear here
    </div>
  </body>
</html>
```

In this example, the question is placed inside a form, with a button to submit the form and show the answer. A placeholder `<div>` element is added at the end of the document and is where the answer to the question will appear.

Part I: Building a New Internet

In the `<head>` section of the page, a JavaScript function is defined, `showAnswer()`. This function uses the DOM method `getElementById()` to find the HTML element with an `id` of `answer` — that's the placeholder `<div>` at the bottom of the page. Once the element is found, the method replaces the HTML contained within the element with new HTML containing the answer to the sum.

Finally, the form defines an `onsubmit` event handler that calls the `showAnswer()` function. This event handler is invoked immediately before the form is submitted to the server (when the button is pressed). Because the JavaScript function does all that is needed here, `return false` is added to the event handler to prevent the submission to the server from actually happening.

AJAX

AJAX, or *Asynchronous Javascript and XML*, is simply a new name for a set of tools that have actually been around for a long time. AJAX has become almost synonymous with Web 2.0, but the techniques that form the basis for AJAX have actually been around since the late 1990s.

Traditional web applications are *page-based*. Every time a new piece of information is requested, or data are updated, a request is sent to the server and a new page is retrieved. This results in many web applications being somewhat laborious to use.

AJAX is intended to overcome this inherent clunkiness of web applications by enabling developers to display or manipulate data within a web page, but without performing a page refresh. This is achieved by means of a JavaScript object called `XMLHttpRequest`. This object allows an HTTP request to be sent to a web server, and the response processed. All this happens behind the scenes of the current page. The requests are carried out *asynchronously* — in other words, the user can continue interacting with the current page while the asynchronous request is being carried out in the background. The result is that web applications can be made to appear much more responsive to a user's actions.

AJAX applications are often much more network-friendly. As usually only part of a page is refreshed in response to a user action, the amount of data transferred across the network is often significantly less than is found in traditional page-based applications.

One of the problems with building AJAX applications has always been supporting the wide variety of web browsers in everyday use. Each web browser has its own individual set of quirks to cope with, and creating an application that works reliably and consistently for the majority of users can be quite a challenge. To this end, a number of toolkits have appeared that take some of the pain out of building AJAX applications. One of the most popular toolkits is Sam Stephenson's Prototype JavaScript framework, available from <http://prototype.conio.net/>, which you will be putting to good use in Chapter 6.

JSON

JSON, or *JavaScript Object Notation*, is a lightweight data-exchange format based on JavaScript syntax. It is designed to be easy for humans to read and write, and also for machines to parse.

It is based around two types of data:

- ❑ **Objects:** a JavaScript object is a set of name/value pairs. The order of the items in the object is not significant.
- ❑ **Lists:** a JavaScript list or array is an ordered set of values.

Objects are defined in JSON like this:

```
{
  "title" : "Flickr Mashups",
  "author" : "David Wilkinson",
  "isbn" : "0470097744"
}
```

Each object is surrounded by braces and within those is a comma-separated list of elements. Each element in the object has a name and a value, separated by a colon.

Lists are comma-separated and enclosed by square brackets:

```
["red", "green", "blue"]

[1, 2, 3, 4, 5]

["cat", 312, true]
```

The values represented in JSON may be strings (enclosed in double-quotes as shown above), numbers, Boolean values (`true` and `false`), or the keyword `null`. Whitespace that isn't part of a string is not significant.

Objects and arrays can be combined to form more complex data structures. For example, the data used in the sample XML document shown earlier in this chapter could be represented in a JSON structure as a list of objects — like this:

```
[
  {
    "title" : "Flickr Mashups",
    "author" : "David Wilkinson",
    "isbn" : "0470097744"
  },
  {
    "title" : "Google Maps Mashups",
    "author" : "Virender Ajmani",
    "isbn" : "0470097752"
  },
  {
    "title" : "del.icio.us Mashups",
    "author" : "Brett O'Connor",
    "isbn" : "0470097760"
  },
  {
    "title" : "Amazon.com Mashups",
    "author" : "Francis Shanahan",
    "isbn" : "0470097779"
  },
  {
    "title" : "Yahoo! Maps Mashups",
    "author" : "Charles Freedman",
    "isbn" : "0470097787"
  }
]
```

You can find out more about JSON at <http://www.json.org>.

Web Servers and HTTP

The key piece of software behind any web-based system is the one that is often taken the most for granted—that is, the web server itself. There are many different web servers available, but the two most commonly encountered are the *Apache Web Server* from the Apache Software Foundation (<http://www.apache.org>) and Microsoft's *Internet Information Services* (<http://www.microsoft.com/iis/>), commonly known as IIS.

The web server's role is to listen for incoming requests and then return an appropriate response. Incoming requests commonly come from web browsers, but a significant number may arrive from other software applications—sites that provide an external interface for developers to use, such as Flickr, usually use a web server to handle incoming requests.

The protocol used by web servers is HTTP, or *Hypertext Transfer Protocol*. HTTP is a simple message-based protocol used to transfer information across the web. HTTP supports a number of different types of message, but the two you are most likely to come across are GET and POST.

Most of the time, when you are using your web browser to access web pages, your browser is sending a series of GET requests to one or more web servers. Every page you access is retrieved via an HTTP GET request, as is every image and graphic, every external style sheet, and every external JavaScript file used by the page—sembling all the different pieces of content required to build up a single page can easily result in a couple of dozen GET requests being sent.

A typical GET request to retrieve the contents of the URL <http://www.flickr.com/photos/> looks something like this:

```
GET /photos/ HTTP/1.1
Host: www.flickr.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.6)
    Gecko/20060728 Firefox/1.5.0.6
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;
    q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Cookie: cookie_epass=b451d297bd19680bea0012147c34c68b; CP=null*;
    cookie_accid=492257; cookie_session=492257%b451d297bd19680bea0012147c34c68b;
    use_master_until=1157985652
```

The first line of an HTTP request consists of the method (in this case, `GET`), then the name of the resource being requested (`/photos/`), and finally the version of the HTTP protocol being used (`HTTP/1.1`). There then follow a series of headers, which provide additional information that the web server can use to decide how best to handle the request. At the end of the headers is a blank line. Most of these headers you can safely ignore unless you are particularly interested in delving into the inner workings of HTTP. A couple, however, are worth explaining.

The `Host` header is used to specify the name of the host machine on which the client is trying to access a resource. At first glance, that seems a somewhat redundant piece of information — after all, surely the host machine knows what it is called? It is specified in the header of the message because a single web server might be responsible for hosting many different sites. Each web site is a *virtual host* configured within the web server. Probably the majority of the web sites on the Internet are, in fact, virtual hosts, each one sharing a web server with many other web sites.

The other header of interest here is the `Cookie` header. Many web sites use cookies to store information within the client browser. This is typically how web sites remember who you are — every time you visit, your browser presents your cookie to the web server, and the web server can use that to identify you. Cookies are sent in the HTTP headers. In this case, the cookie is sent from the client to the server. When the cookie value is initially set, or whenever it changes, a `Set-Cookie` header is sent from the server back to the client in the response.

Let's take a look at a typical HTTP response. The response shown below is to the request above:

```
HTTP/1.1 200 OK
Date: Sun, 10 Sep 2006 15:14:23 GMT
Server: Apache/2.0.52
Cache-Control: private
Connection: close
Content-Type: text/html; charset=utf-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>Flickr: Photos from everyone in Flickr</title>
  ...
```

The first line consists of a string identifying the protocol version being used, a three-digit status code, and finally a human-readable message describing the meaning of the status code. Here the status is `200`, which means that everything is fine and the requested resource is contained in the body of the response. Status codes of the form `2xx` indicate success. `3xx` means that further action is required from the client — perhaps the resource has moved and a new request should be sent to retrieve it from the new location. Status codes of the form `4xx` indicate an error on the part of the client — the most commonly seen is `404`, which means that the requested resource was not found on the server. Finally, `5xx` codes indicate an error on the part of the server.

After the HTTP status, there follow a number of headers. The most important one shown here is the `Content-Type` header — this contains the MIME type of the requested resource. Here it is `text/html`, so the web browser knows to render this as a web page. After the HTTP headers is a blank line, followed by the requested resource itself — here, an HTML page.

The other web server method you are likely to come across is `POST`. Whereas `GET` requests a resource from the web server, `POST` is used to send data to a resource on the web server. The most common use of `POST` is for filling in a form on a web page: the data included in the form are sent in the body of the `POST` request.

Web Browsers

The web browser is, in many ways, the companion to the web server. It is the software that handles the client side of an HTTP transaction. Web browsers have become such a fundamental part of our everyday lives that you might wonder why they even warrant a mention here — for many people, the web browser is one of the primary tools on a computer.

Taking the browser for granted is one of the most common mistakes people make when building web applications. All too often, people fall into the trap of forgetting that the browser they use isn't the only one in existence. Because they are built around a common set of standards, you would expect all web browsers to work with all web sites. Well, by and large, they do — but once you start building more interesting applications, using a wider variety of technologies, you notice that there are, in practice, inconsistencies. Sometimes, there are things that the standards don't actually specify, or that may be open to interpretation, or perhaps a browser only partially implements a particular standard. Some differences may even be caused by bugs in the browser itself.

People use a wide variety of browsers: Internet Explorer, Firefox, Netscape, Safari, and Opera are some of the more common browsers in use — but that list is far from complete. To complicate matters, each browser typically exists in many different versions — each with its own different idiosyncrasies, and each still in use by someone somewhere on the Internet.

To compound the problem, most browsers are capable of running on a variety of platforms — Microsoft Windows, Linux, OS X — and for each platform are usually a number of different variants: Windows XP, Windows 2000, Windows Me, and Vista, for example. Each browser may behave slightly differently on each of those platforms.

If you think that sounds like a daunting set of variations — you're right; it is. What's more, there is no real answer. The total combination of browser/platform combinations is too large for all but the largest development teams to consider testing against. The important thing to remember is that just because your mashup works fine for you, you must not assume that everyone else will see it in exactly the same way.

At the very least, you should always check your mashup under a couple of different browsers. On Windows, try with Internet Explorer and Firefox. On a Mac, try both Firefox and Safari. If you can, try it out with both Mac and Windows. Simply doing this will catch the vast majority of problems. All too often, people are disappointed when they go to check out the latest cool mashup they've heard about, only to find that it doesn't work with their chosen browser. Don't let this happen to you!

PHP and Perl

PHP and Perl are two of the most commonly used scripting languages in use on the Internet today, and most of the examples in this book make use of one of them.

PHP is a recursive acronym for *PHP: Hypertext Preprocessor*. It is a general-purpose scripting language, although it is most commonly encountered in the generation of web pages — a task to which it is ideally suited. Rather than writing lots of code to output HTML tags, you can embed PHP directly into standard HTML pages:

```
<html>
  <head>
    <title>Arithmetic</title>
```

```

</head>
<body>
  <h1>Question</h1>
  <p>
    What is the answer to the sum <code>2 + 2</code>?
  </p>
  <h2>Answer</h2>
  <?php
    echo 2 + 2 ;
  ?>
</body>
</html>

```

I'll be making a lot of use of PHP in the chapters that follow, and some experience of PHP is assumed throughout most of this book. If you need to brush up on your PHP skills, the main PHP web site at <http://www.php.net/> is always a good place to start. There you will find the PHP documentation and tutorials.

Like PHP, Perl is a scripting language, and it has a very long history. Created by Larry Wall in the mid 1980s, it has become one of the most widely used tools around. System administrators refer to it as the “duct tape of the Internet”: its versatility and almost ubiquitous availability allow it to be easily applied in most situations.

Perl is available for a large variety of computing platforms, and is supported by a comprehensive library of add-on modules available from CPAN—the *Comprehensive Perl Archive Network*—which can be found at <http://cpan.perl.org/>.

Regular Expressions

A *regular expression* is a string that defines a *pattern* that is used to match one or more strings, and is typically used to search for specific patterns within a larger string. The easiest way to understand what this means is by seeing an example or two—let's look at a few examples of typical regular expressions.

Many types of regular expressions exist—in this book, I use only *Perl-style* regular expressions.

Here's a very simple regular expression:

```
/Flickr/
```

The regular expression, often shortened to *regex*, is delimited by forward slashes. Within the slashes is the pattern to be matched. Here the pattern is very simple—it's the word `Flickr`. This pattern will match any string that contains the text `Flickr`. The following are all strings that match the pattern.

```
Flickr Mashups
The Flickr web site
I like Flickr
```

Most regular expressions aren't that straightforward, however. Many special characters can be used within a regex pattern to perform more complex matches. For example, the caret character (^) is used to match the start of a string, and the dollar sign (\$) matches the end of a string. Look at this regex:

```
/^Flickr/
```

Part I: Building a New Internet

The `^` character at the start of the pattern means that only strings that begin with the word `Flickr` will be matched. In the three sample strings, only one matches:

```
Flickr Mashups
```

Similarly, this regexp

```
/Flickr$/
```

only matches the string that ends with `Flickr`:

```
I like Flickr
```

The period (`.`) will match any character, so the following pattern will match any string that has an `F` and an `r` separated by four characters:

```
/F....r/
```

That means it will match all three of the Flickr strings, and will also match these two:

```
Farmer Giles has a herd of cows  
Fly around the world
```

The asterisk (`*`) is a modifier that means “zero or more occurrences” of the preceding character. So, the regexp

```
/che*se/
```

will match

```
cheese  
cheeeeeeeese  
chse
```

Similarly, the plus (`+`) and question mark (`?`) characters are modifiers that mean *one or more occurrences* and *exactly zero or one occurrences*, respectively.

You can use square brackets to enclose a set of alternatives, so the regexp

```
/br[eo]ad/
```

will match both `bread` and `broad`.

You can use `\d` to match a numeric digit (0–9) and `\w` to represent a “word” character (a letter or digit, or the underscore character), so the regexp

```
/\d\d\d\d-\d\d-\d\d/
```

can be used to match dates in the format *yyyy-mm-dd*.

Regular expressions are a very powerful tool, and there's much more to them than can be covered in a brief introduction such as this. You'll be using them in various places in this book, often within PHP. The PHP web site has a handy description of regular expression pattern syntax at <http://www.php.net/manual/en/reference.pcre.pattern.syntax.php>.

REST

REST, or *Representational State Transfer*, is a term that is often misused. Strictly speaking, REST describes an architectural approach to building software systems. The term REST was first used by Roy Fielding, one of the principal authors of the HTTP specification, in his PhD dissertation: "Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

A web application built on REST principles would model each object stored in the system as a unique URL. So, a bookstore might use URLs of the following form:

```
http://www.somebookstore.com/books/0470097744
```

Here, `0470097744` is the ISBN of the book in question. When the URL is accessed, a document representing the book is returned — on the web, this is most likely an HTML page. Of course, there isn't really a different HTML page for every single book in the store — the pages are all dynamically generated based on the information passed in the URL.

The HTML page for the book might include a link to a list of reviews of the book:

```
http://www.somebookstore.com/books/0470097744/reviews
```

Each review would also have its own URL:

```
http://www.somebookstore.com/books/0470097744/reviews/1
```

Each time the user follows a link to new URL, the new URL contains all of the context required to service the request — there is no state required to be stored on the server; it is stored entirely in the URL. This is a key feature of a REST system.

In the Web 2.0 world, the term REST is also often used to refer to any simple HTTP interface that exchanges information (usually XML) without the aid of an additional messaging layer such as SOAP (see the following section). The Flickr API has such a REST interface, which is covered in detail in Chapter 4.

REST interfaces such as the one offered by Flickr are URL-based, and represent a very concise way of sending a request to the remote system. A typical REST request URL might look something like this:

```
http://www.flickr.com/services/rest/?method=flickr.photos.search
&user_id=50317659@N00&per_page=6&api_key=1f2811827f3284b3aa12f7bbb7792b8d
```

This URL performs a search for a user's recent photos on Flickr. Don't worry about the details of what all those parameters mean right now — we'll look at this request in more detail in Chapter 4. You can

Part I: Building a New Internet

see, however, that the whole of the request is found in the URL. One of the big advantages of using REST is that you can try things out simply by typing the URL into your web browser — there are no complicated messaging protocols to conform to. As far as application interfaces go, REST is as close to instant gratification as you are likely to get as a software developer.

If you do try typing this example URL into your browser, it won't actually work. Before you can use Flickr's services, you will need to get an API key from it. We'll explain all about what API keys are, how you get one, and how you use them in the next few chapters.

SOAP

SOAP is an XML-based messaging format, widely used as the means of communication with web services. The SOAP standard is maintained by W3C. Originally, SOAP was an acronym for *Simple Object Access Protocol*, but that particular definition has been dropped in recent years.

SOAP requests and responses are both XML documents, and as such can be very verbose. For example, when converted to a SOAP message, the simple Flickr REST request shown previously becomes something like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<s:Envelope
  xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
>
  <s:Body>
    <x:FlickrRequest xmlns:x="urn:flickr">
      <method>flickr.photos.search</method>
      <api_key>1f2811827f3284b3aa12f7bbb7792b8d</api_key>
      <user_id>50317659@N00</user_id>
      <per_page>6</per_page>
    </x:FlickrRequest>
  </s:Body>
</s:Envelope>
```

Many web services that provide SOAP interfaces use WSDL, or *Web Services Description Language*, as a means of formally describing the services available and defining how to both construct the request-message documents and understand the response documents.

Because of the widespread use of SOAP, many SOAP toolkits are available, and many development environments provide direct support for SOAP. Such toolkits, however, typically rely on the existence of a WSDL description of the web service, and if this is not available, they are of limited use.

XML-RPC

XML-RPC is another commonly used XML-based messaging format and is a mechanism for providing *remote procedure calls* using XML. A remote procedure call is simply a means of executing a service on another computer. XML-RPC was the precursor to SOAP and, as with SOAP, the message format is somewhat verbose. A typical XML-RPC message might look something like this:

```

<?xml version="1.0" encoding="utf-8" ?>
<methodCall>
  <methodName>flickr.photos.search</methodName>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>api_key</name>
            <value>
              <string>1f2811827f3284b3aa12f7bbb7792b8d</string>
            </value>
          </member>
          <member>
            <name>user_id</name>
            <value>
              <string>50317659@N00</string>
            </value>
          </member>
          <member>
            <name>per_page</name>
            <value>
              <int>6</int>
            </value>
          </member>
        </struct>
      </value>
    </param>
  </params>
</methodCall>

```

XML-RPC isn't in quite such widespread use as SOAP, and less toolkit support is available for it within the developer community.

Databases

Most systems of any complexity make use of a database somewhere behind the scenes. For very small amounts of data, and small numbers of users, you can get by with storing any data you need to keep track of in files on disk, or perhaps in cookies, but before long you'll find yourself needing a proper *database management system*. A database management system provides an efficient means of storing, searching, and maintaining large quantities of information.

One of the most popular database systems in use in web systems today is MySQL. MySQL is a freely available open-source database from MySQL AB and is available from <http://www.mysql.com>. MySQL is a relational database, and as might be expected from the name, uses *Structured Query Language* (SQL) to manage and query the data.

SQL and MySQL database administration are a large and complex topic—and one impossible to cover in any depth in a book like this. Chapter 14 contains instructions for setting up a MySQL database, together with a brief introduction to basic SQL commands, and many more resources on the Internet explore these subjects in much more detail. The main MySQL web site has full documentation and includes a brief tutorial on using MySQL.

Curl

Earlier on in this chapter, you saw how easy it was to send a REST-style query to a web server — you can simply type it into the address bar of your web browser. But what if you don't want to use your web browser? Perhaps you want to see exactly what is being returned, or perhaps you want to save the results to a file. What about more complicated messages such as SOAP or XML-RPC? How can you try out sending those kinds of message? The answer is a little application called *curl*.

Curl is a command-line tool for sending and retrieving files using URL syntax. It supports a wide range of protocols including, most importantly, HTTP. Curl is free and available for a very wide variety of platforms. You can download it from <http://curl.haxx.se/>.

When you come to run curl, you have a huge number of command-line options to choose from. These are all documented on the curl web site, but here are a few common ones. To retrieve the contents of a URL such as <http://www.flickr.com/>, simply type the following:

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

To retrieve the page, but save it to a file:

```
curl -o file.html http://www.flickr.com/
```

And to POST the contents of a file to a URL:

```
curl -d @request.xml http://www.flickr.com/services/soap/
```

Curl is a very powerful and versatile tool; it is well worth spending a little time to read through the manual.

Summary

This chapter introduced you to the world of mashups. You saw the difference between client- and server-side mashups and were introduced to many of the technologies that make mashups possible. Now, let's move on and take a closer look at Flickr, the system that will be the focus of our mashups for the rest of this book.