2

IDEAS FOR WEBBOT PROJECTS

It's often more difficult to find applications for new technology than it is to learn the technology itself. Therefore, this chapter focuses on encouraging you to generate ideas for things that you can do with webbots. We'll explore how webbots capitalize on browser limitations, and we'll see a few examples of what people are currently doing with webbots. We'll wrap up by throwing out some wild ideas that might help you expand your expectations of what can be done online.

Inspiration from Browser Limitations

A useful method for generating ideas for webbot projects is to study what *cannot* be done by simply pointing a browser at a typical website. You know that browsers, used in traditional ways, cannot automate your Internet experience. For example, they have these limitations:

- Browsers cannot aggregate and filter information for relevance.
- Browsers cannot interpret what they find online.
- Browsers cannot act on your behalf.

However, a browser may leverage the power of a webbot to do many things that it could not do alone. Let's look at some real-life examples of how browser limitations were leveraged into actual webbot projects.

Webbots That Aggregate and Filter Information for Relevance

TrackRates.com (*http://www.trackrates.com*, shown in Figure 2-1) is a website that deploys an army of webbots to aggregate and filter hotel room prices from travel websites. By identifying room prices for specific hotels for specific dates, it determines the actual market value for rooms up to three months into the future. This information helps hotel managers intelligently price rooms by specifically knowing what the competition is charging for similar rooms. TrackRates.com also reveals market trends by performing statistical analysis on room prices, and it tries to determine periods of high demand by indicating dates on which hotels have booked all of their rooms.

Figure 2-1: TrackRates.com

I wrote TrackRates.com to help hotel managers analyze local markets and provide facts for setting room prices. Without the TrackRates.com webbot, hotel managers either need to guess what their rooms are worth, rely on less current information about their local hotel market, or go through the arduous task of manually collecting this data.

Webbots That Interpret What They Find Online

WebSiteOptimization.com (*http://www.websiteoptimization.com*) uses a webbot to help web developers create websites that use resources effectively. This webbot accepts a web page's URL (as shown in Figure 2-2) and analyzes how each graphic, CSS, and JavaScript file is used by the web page. In the interest of full disclosure, I should mention that I wrote the backend for this web page analyzer.

Figure 2-2: A website-analyzing webbot

The WebSiteOptimization.com webbot analyzes the data it collects and offers suggestions for optimizing website performance. Without this tool, developers would have to manually parse through their HTML code to determine which files are required by web pages, how much bandwidth they are using, and how the organization of the web page affects its performance.

Webbots That Act on Your Behalf

Pokerbots, webbots that play online poker, are a response to the recent growth in online gambling sites, particularly gaming sites with live poker rooms. While the action in these pokers sites is live, not all the players are. Some online poker players are webbots, like Poker Robot, shown in Figure 2-3.

Webbots designed to play online poker not only know the rules of Texas hold 'em but use predetermined business rules to expertly read how others play. They use this information to hold, fold, or bet appropriately. Reportedly, these automated players can very effectively pick the pockets of new and inexperienced poker players. Some *collusion webbots* even allow one virtual

player to play multiple hands at the same table, while making it look like a separate person is playing each hand. Imagine playing against a group of people who not only know each other's cards, but hold, fold, and bet against you as a team!

Figure 2-3: An example pokerbot

Obviously, such webbots that play expert poker (and cheat) provide a tremendous advantage. Nobody knows exactly how prevalent pokerbots are, but they have created a market for anti-pokerbot software.

A Few Crazy Ideas to Get You Started

One of the goals of this book is to encourage you to write new and experimental webbots of your own design. A way to jumpstart this process is to brainstorm and generate some ideas for potential projects. I've taken this opportunity to list a few ideas to get you started. These ideas are not here necessarily because they have commercial value. Instead, they should act as inspiration for your own webbots and what you want to accomplish online.

When designing a webbot, remember that the more specifically you can define the task, the more useful your webbot will be. What can you do with a webbot? Let's look at a few scenarios.

Help Out a Busy Executive

Suppose you're a busy executive type and you like to start your day reading your online industry publication. Time is limited, however, and you only let yourself read industry news until you've finished your first cup of coffee. Therefore, you don't want to be bothered with stories that you've read before or that you know are not relevant to your business. You ask your developer

to create a specialized webbot that consolidates articles from your favorite industry news sources and only displays links to stories that it has not shown you before.

The webbot could ignore articles that contain certain key phrases you previously entered in an *exclusion list*[1] and highlight articles that contain references to you or your competitors. With such an application, you could quickly scan what's happening in your industry and only spend time reading relevant articles. You might even have more time to enjoy your coffee.

Save Money by Automating Tasks

It's possible to design a webbot that automatically buys inventory for a store, given a predetermined set of buying criteria. For example, assume you own a store that sells used travel gear. Some of your sources for inventory are online auction websites.[2] Say you are interested in bidding on under-priced Tumi suitcases during the closing minute of their auctions. If you don't use a webbot of some sort, you will have to use a web browser to check each auction site periodically.

Without a webbot, it can be expensive to use the Internet in a business setting, because repetitive tasks (like procuring inventory) are time consuming without automation. Additionally, the more mundane the task, the greater the opportunity for human error. Checking online auctions for products to resell could easily consume one or two hours a day—up to 25 percent of a 40-hour work week. At that rate, someone with an annual salary of $80,000 would cost a company $20,000 a year to procure inventory (without a webbot). That cost does not include the cost of opportunities lost while the employee manually monitors auction sites. In scenarios like this, it's easy to see how product acquisition with a webbot saves a lot of money—even for a small business with small requirements. Additionally, a webbot may uncover bargains missed by someone manually searching the auction site.

Protect Intellectual Property

You can write a webbot to protect your online intellectual property. For example, suppose you spent many hours writing a JavaScript program. It has commercial value, and you license the script for others to use for a fee. You've been selling the program for a few months and have learned that some people are downloading and using your program without paying for it. You write a webbot to find websites that are using your JavaScript program without your permission. This webbot searches the Internet and makes a list of URLs that reference your JavaScript file. In a separate step, the webbot does a *whois* lookup on the domain to determine the owner from the domain registrar.[3]

---

[1] An *exclusion list* is a list of keywords or phrases that are ignored by a webbot.

[2] Some online auctions actually provide to1ols to help you write webbots that manage auctions. If you're interested in automating online auctions, check out eBay's Developers Program (*http://developer.ebay.com*).

[3] *whois* is a service that returns information about the owner of a website. You can do the equivalent of a *whois* from a shell script or from an online service.

If the domain is not one of your registered users, the webbot compiles contact information from the domain registrar so you can contact the parties who are using unlicensed copies of your code.

Monitor Opportunities

You can also write webbots that alert you when particular opportunities arise. For example, let's say that you have an interest in acquiring a Jack Russell Terrier.[4] Instead of devoting part of each day to searching for your new dog, you decide to write a webbot to search for you and notify you when it finds a dog meeting your requirements. Your webbot performs a daily search of the websites of local animal shelters and dog rescue organizations. It parses the contents of the sites, looking for your dog. When the webbot finds a Jack Russell Terrier, it sends you an email notification describing the dog and its location. The webbot also records this specific dog in its database, so it doesn't send additional notifications for the same dog in the future. This is a fairly common webbot task, which could be modified to automatically discover job listings, sports scores, or any other timely information.

Verify Access Rights on a Website

Webbots may prevent the potentially nightmarish situation that exists for any web developer who mistakenly gives one user access to another user's data. To avoid this situation, you could commission a webbot to verify that all users receive the correct access to your site. This webbot logs in to the site with every viable username and password. While acting on each user's behalf, the webbot accesses every available page and compares those pages to a list of appropriate pages for each user. If the webbot finds a user is inadvertently able to access something he or she shouldn't, that account is temporarily suspended until the problem is fixed. Every morning before you arrive at your office, the webbot emails a report of any irregularities it found the night before.

Create an Online Clipping Service

Suppose you're very vain, and you'd like a webbot to send an email to your mother every time a major news service mentions your name. However, since you're not vain enough to check all the main news websites on a regular basis, you write a webbot that accomplishes the task for you. This webbot accesses a collection of websites, including CNN, Forbes, and Fortune. You design your webbot to look only for articles that mention your name, and you employ an exclusion list to ignore all articles that contain words or phrases like *shakedown*, *corruption*, or *money laundering*. When the webbot finds an appropriate article, it automatically sends your mother an email with a link to the article. Your webbot also blind copies you on all emails it sends so you know what she's talking about when she calls.

[4] I actually met my dog online.

Plot Unauthorized Wi-Fi Networks

You could write a webbot that aids in maintaining network security on a large corporate campus. For example, suppose that you recently discovered that you have a problem with employees attaching unauthorized wireless access points to your network. Since these unauthorized access points occur inside your firewalls and proxies, you recognize that these unauthorized Wi-Fi networks pose a security risk that you need to control. Therefore, in addition to a new security policy, you decide to create a webbot that automatically finds and records the location of all wireless networks on your corporate campus.

You notice that your mail room uses a small metal cart to deliver mail. Because this cart reaches every corner of the corporate campus on a daily basis, you seek and obtain permission to attach a small laptop computer with a webbot and Global Positioning System (GPS) card to the cart. As your webbot hitches a ride through the campus, it looks for open wireless network connections. When it finds a wireless network, it uses the open network to send its GPS location to a special website. This website logs the GPS coordinates, IP address, and date of uplink in a database. If you did your homework correctly, in a few days your webbot should create a map of all open Wi-Fi networks, authorized and unauthorized, in your entire corporate campus.

Track Web Technologies

You could write webbots that use *web page headers*, the information that servers send to browsers so they may correctly render websites, to maintain a list of web technologies used by major corporations. Headers typically indicate the type of webserver (and often the operating system) that websites use, as shown in Figure 2-4.

```
C:\>curl --head http://www.chrysler.com

Server: IBM_HTTP_Server/2
Date: Sun, 04 Dec 2011 20:28:47 GMT
Connection: keep-alive
```

Figure 2-4: A web page header showing server technology

Your webbot starts by accessing the headers of each website from a list that you keep in a database. It then parses web technology information from the header. Finally, the webbot stores that information in a database that is used by a graphing program to plot how server technology choices change over time.

Allow Incompatible Systems to Communicate

In addition to creating human-readable output, you could design a webbot that only talks to other computers. For example, let's say that you want to synchronize two databases, one on a local private network and one that's behind a public website. In this case, *synchronization* (ensuring that both

databases contain the same information) is difficult because the systems use different technologies with incompatible synchronization techniques. Given the circumstances, you could write a webbot that runs on your private network and, for example, analyzes the public database through a password-protected web service every morning. The webbot uses the Internet as a common protocol between these databases, analyzes data on both systems, and exchanges the appropriate data to synchronize the two databases.

Final Thoughts

Studying browser limitations is one way to uncover ideas for new webbot designs. You've seen some real-world examples of webbots in use and read some descriptions of conceptual webbot designs. But, enough with theory—let's head to the lab!

The next five chapters describe the basics of webbot development: downloading pages, parsing data, emulating form submission, and managing large amounts of data. Once you master these concepts, you can move on to actual webbot projects.

3

DOWNLOADING WEB PAGES

The most important thing a webbot does is move web pages from the Internet to your computer. Once the web page is on your computer, your webbot can parse and manipulate it.

This chapter will show you how to write simple PHP scripts that download web pages. More importantly, you'll learn PHP's limitations and how to overcome them with *PHP/CURL*, a special binding of the cURL library that facilitates many advanced network features. cURL is used widely by many computer languages as a means to access network files with a number of protocols and options.

NOTE *While web pages are the most common targets for webbots and spiders, the Web is not the only source of information for your webbots. Later chapters will explore methods for extracting data from newsgroups, email, and FTP servers, as well.*

Prior to discovering PHP, I wrote webbots in a variety of languages, including Visual Basic, Java, and Tcl/Tk. But due to its simple syntax, in-depth string parsing capabilities, networking functions, and portability, PHP proved ideal for webbot development. However, PHP is primarily a server language, and its chief purpose is to help webservers interpret incoming requests and send

the appropriate web pages in response. Since webbots don't serve pages (they request them), this book supplements PHP built-in functions with PHP/CURL and a variety of libraries, developed specifically to help you learn to write webbots and spiders.

Think About Files, Not Web Pages

To most people, the Web appears as a collection of web pages. But in reality, the Web is collection of files that form those web pages. These files may exist on servers anywhere in the world, and they only create web pages when they are viewed together. Because browsers simplify the process of downloading and rendering the individual files that make up web pages, you need to know the nuts and bolts of how web pages are put together before you write your first webbot.

When your browser requests a file, as shown in Figure 3-1, the webserver that fields the request sends your browser a *default* or *index file*, which maps the location of all the files that the web page needs and tells how to render the text and images that comprise that web page.

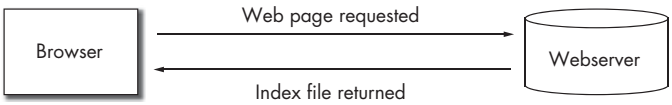

Figure 3-1: When a browser requests a web page, it first receives an index file.

As a rule, this index file also contains references to the other files required to render the complete web page,[1] as shown in Figure 3-2. These may include images, JavaScript, style sheets, or complex media files like Flash, QuickTime, or Windows Media files. The browser downloads each file separately, as it is referenced by the index file.

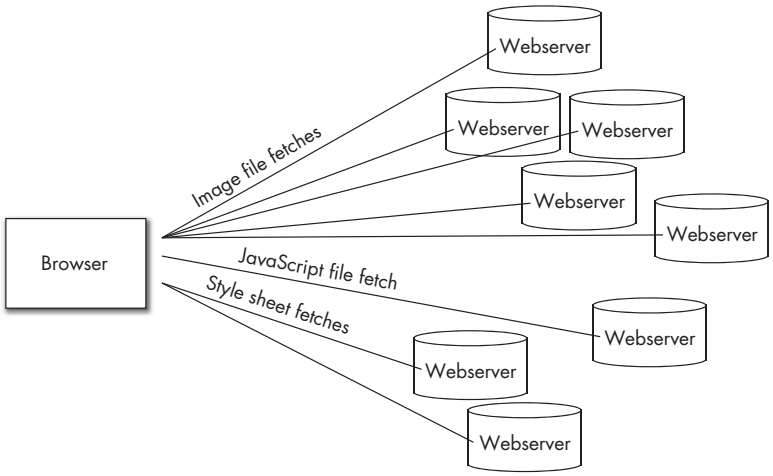

Figure 3-2: Downloading files, as they are referenced by the index file

[1] Some very simple websites consist of only one file.

For example, if you request a web page with references to eight items your single web page actually executes nine separate file downloads (one for the web page and one for each file referenced by the web page). Usually, each file resides on the same server, but they could just as easily exist on separate domains, as shown in Figure 3-2.

Downloading Files with PHP's Built-in Functions

Before you can appreciate PHP/CURL, you'll need to familiarize yourself with PHP's built-in functions for downloading files from the Internet.

Downloading Files with fopen() and fgets()

PHP includes two simple built-in functions for downloading files from a network—`fopen()` and `fgets()`. The `fopen()` function does two things. First, it creates a *network socket*, which represents the link between your webbot and the network resource you want to retrieve. Second, it implements the HTTP *protocol*, which defines how data is transferred. With those tasks completed, `fgets()` leverages the networking ability of your computer's operating system to pull the file from the Internet.

Creating Your First Webbot Script

Let's use PHP's built-in functions to create your first webbot, which downloads a "Hello, world!" web page from this book's companion website. The short script is shown in Listing 3-1.

```
# Define the file you want to download
$target      = "http://www.WebbotsSpidersScreenScrapers.com/hello_world.html";
$file_handle = fopen($target, "r");

# Fetch the file
while (!feof($file_handle))
    echo fgets($file_handle, 4096);
fclose($file_handle);
```

Listing 3-1: Downloading a file from the Web with `fopen()` and `fgets()`

As shown in Listing 3-1, `fopen()` establishes a network connection to the *target*, or file you want to download. It references this connection with a *file handle*, or network link called `$file_handle`. The script then uses `fopen()` to fetch and echo the file in 4,096-byte chunks until it has downloaded and displayed the entire file. Finally, the script executes an `fclose()` to tell PHP that it's finished with the network handle.

Before we can execute the example in Listing 3-1, we need to examine the two ways to execute a webbot: You can run a webbot either in a browser or in a command shell.[2]

[2] See Chapter 22 for more information on executing webbots as scheduled events.

Executing Webbots in Command Shells

If you have a choice, it is usually better to execute webbots from a shell or command line. Webbots generally don't care about web page formatting, so they will display exactly what is returned from a webserver. Browsers, in contrast, will interpret HTML tags as instructions for rendering the web page. For example, Figure 3-3 shows what Listing 3-1 looks like when executed in a shell.

```
C:\>php script_3_1.php
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
        <title>Hello, world!</title>
</head>

<body>
Congratulations! If you can read this, <br>
you successfully downloaded this file.
</body>
</html>
```

Figure 3-3: Running a webbot script in a shell

Executing Webbots in Browsers

To run a webbot script in a browser, simply load the script on a webserver and execute it by loading its URL into the browser's location bar as you would any other web page. Contrast Figure 3-3 with Figure 3-4, where the same script is run within a browser. The HTML tags are gone, as well as all of the structure of the returned file; the only things displayed are two lines of text. Running a webbot in a browser only shows a partial picture and often hides important information that a webbot needs.

NOTE *To display HTML tags within a browser, surround the output with* `<xmp>` *and* `</xmp>` *tags.*

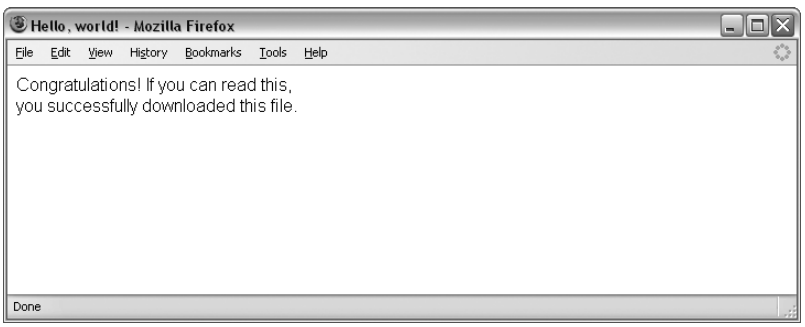

Figure 3-4: Browser "rendering" the output of a webbot

Browser buffering is another complication you might run into if you try to execute a webbot in a browser. Buffering is useful when you're viewing web pages because it allows a browser to wait until it has collected enough of a web page before it starts *rendering* or displaying the web page. However, browser buffering is troublesome for webbots because they frequently run for extended periods of time—much longer than it would take to download a typical web page. During prolonged webbot execution, status messages written by the webbot may not be displayed by the browser while it is buffering the display.

I have one webbot that runs continuously; in fact, it once ran for seven months before stopping during a power outage. This webbot could never run effectively in a browser because browsers are designed to render web pages with files of finite length. Browsers assume short download periods and may buffer an entire web page before displaying anything—therefore, never displaying the output of your webbot.

NOTE *Browsers can still be very useful for creating interfaces that set up or control the actions of a webbot. They can also be useful for displaying the results of a webbot's work.*

Downloading Files with file()

An alternative to `fopen()` and `fgets()` is the function `file()`, which downloads formatted files and places them into an array. This function differs from `fopen()` in two important ways: One way is that, unlike `fopen()`, it does not require you to create a file handle, because it creates all the network preparations for you. The other difference is that it returns the downloaded file as an array, with each line of the downloaded file in a separate array element. The script in Listing 3-2 downloads the same web page used in Listing 3-1, but it uses the `file()` command.

```
<?
// Download the target file
$target = "http://www.WebbotsSpidersScreenScrapers.com/hello_world.html";
$downloaded_page_array = file($target);

// Echo contents of file
for($xx=0; $xx<count($downloaded_page_array); $xx++)
      echo $downloaded_page_array[$xx];
?>
```

Listing 3-2: Downloading files with `file()`

The `file()` function is particularly useful for downloading *comma-separated value (CSV) files*, in which each line of text represents a row of data with columnar formatting (as in an Excel spreadsheet). Loading files line-by-line into an array, however, is not particularly useful when downloading HTML files because the data in a web page is not defined by rows or columns; in a CSV file, however, rows and columns have specific meaning.

Introducing PHP/CURL

While PHP is capable when it comes to simple file downloads, most real-life applications require additional functionality to handle advanced issues such as form submission, authentication, redirection, and so on. These functions are difficult to facilitate with PHP's built-in functions alone. Forunately, every PHP install should include a library called PHP/CURL, which automatically takes care of these advanced topics. Most of this book's examples exploit the benefit of PHP/CURL's ability to download files.

The open source cURL project is the product of Swedish developer Daniel Stenberg and a team of developers. The cURL library is available for use with nearly any computer language you can think of. When cURL is used with PHP, it's known as PHP/CURL.

The name *cURL* is either a blend of the words *client* and *URL* or an acronym for the words *client URL Request Library*—you decide. cURL does everything that PHP's built-in networking functions do and a lot more. Appendix A expands on PHP/CURL's features, but here's a quick overview of the things PHP/CURL can do for you, a webbot developer.

Multiple Transfer Protocols

Unlike the built-in PHP network functions, PHP/CURL supports multiple transfer protocols, including FTP, FTPS, HTTP, HTTPS, Gopher, Telnet, and LDAP. Of these protocols, the most important is probably HTTPS, which allows webbots to download from encrypted websites that employ the Secure Sockets Layer (SSL) protocol.

Form Submission

PHP/CURL provides easy ways for a webbot to emulate browser form submission to a server. PHP/CURL supports all of the standard *methods*, or form submission protocols, as you'll learn in Chapter 6.

Basic Authentication

PHP/CURL allows webbots to enter password-protected websites that use basic authentication. You've encountered authentication if you've seen this familiar gray box, shown in Figure 3-5, asking for your username and password. PHP/CURL makes it easy to write webbots that enter and use password-protected websites.

Figure 3-5: A basic authentication prompt

Cookies

Without PHP/CURL, it is difficult for webbots to read and write *cookies*, those small bits of data that websites use to create session variables that track your movement. Websites also use cookies to manage shopping carts and authenticate users. PHP/CURL makes it easy for your webbot to interpret the cookies that webservers send it; it also simplifies the process of showing webservers all the cookies your webbot has written. Chapters 20 and 21 have much more to say on the subject of webbots and cookies.

Redirection

Redirection occurs when a web browser looks for a file in one place, but the server tells it that the file has moved and that it should download it from another location. For example, the website *www.company.com* may use redirection to force browsers to go to *www.company.com/spring_sale* when a seasonal promotion is in place. Browsers handle redirections automatically, and PHP/CURL allows webbots to have the same functionality.

Agent Name Spoofing

Every time a webserver receives a file request, it stores the requesting agent's name in a log file called an *access log file*. This log file stores the time of access, the IP address of the requester, and the *agent name*, which identifies the type of program that requested the file. Generally, agent names identify the browser that the web surfer was using to view the website.

Some agent names that a server log file may record are shown in Listing 3-3. The first four names are browsers; the last is the Google spider.

```
Mozilla/5.0 (Windows NT 6.1;) Gecko/20100921 Firefox/4.0b7pre
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1;)
Mozilla/5.0 (Windows NT 5.1) AppleWebKit/534.25 Chrome/12.0.706.0
Googlebot/2.1 (+http://www.google.com/bot.html)
```

Listing 3-3: Agent names as seen in a file access log[3]

A webbot using PHP/CURL can assume any appropriate (or inappropriate) agent name. For example, sometimes it is advantageous to identify your webbots, as Google does. Other times, it is better to make your webbot look like a browser. If you write webbots that use the `LIB_http` library (described later), your webbot's agent name will be *Test Webbot*. If you download a file from a webserver with PHP's `fopen()` or `file()` functions, your agent name will be the version of PHP installed on your computer.

[3] A more complete list of known user agent names is found at *http://www.useragentstring.com/pages/useragentstring.php*.

Referer Management

PHP/CURL allows webbot developers to change the *referer*, which is the reference that servers use to detect which link the web surfer clicked. Sometimes webservers use the referer to verify that file requests are coming from the correct place. For example, a website might enforce a rule that prevents downloading of images unless the referring web page is also on the same webserver. This prohibits people from *bandwidth stealing*, or writing web pages using images on someone else's server. PHP/CURL allows a webbot to set the referer to an arbitrary value.

Socket Management

PHP/CURL also gives webbots the ability to recognize when a webserver isn't going to respond to a file request. This ability is vital because, without it, your webbot might hang (forever) waiting for a server response that will never happen. With PHP/CURL, you can specify how long a webbot will wait for a response from a server before it gives up and moves on.

Installing PHP/CURL

Since PHP/CURL is tightly integrated with PHP, installation should be unnecessary, or at worst, easy. You probably already have PHP/CURL on your computer; you just need to enable it in *php.ini*, the PHP configuration file. If you're using Linux, FreeBSD, OS X, or another Unix-based operating system, you may have to recompile your copy of Apache/PHP to enjoy the benefits of PHP/CURL. Installing PHP/CURL is similar to installing any other PHP library. If you need help, you should reference the PHP website (*http://www.php.net*) for the instructions for your particular operating system and PHP version.

LIB_http

Since PHP/CURL is very flexible and has many configurations, it is often handy to use it within a *wrapper function*, which simplifies the complexities of a code library into something easier to understand. For your convenience, this book uses a library called `LIB_http`, which provides wrapper functions to the PHP/CURL features you'll use most. The remainder of this chapter describes the basic functions of the `LIB_http` library.

`LIB_http` is a collection of PHP/CURL routines that simplify downloading files. It contains defaults and abstractions that facilitate downloading files, managing cookies, and completing online forms. The name of the library refers to the HTTP protocol used by the library. Some of the reasons for using this library will not be evident until we cover its more advanced features. Even simple file downloads, however, are made easier and more robust with `LIB_http` because of PHP/CURL. The most recent version of `LIB_http` is available at this book's website.

Familiarizing Yourself with the Default Values

To simplify its use, LIB_http sets a series of default conditions for you, as described below:

- Your webbot's agent name is *Test Webbot*.
- Your webbot will time out if a file transfer doesn't complete within 25 seconds.
- Your webbot will store cookies in the file *c:\cookie.txt*.
- Your webbot will automatically follow a maximum of four redirections, as directed by servers in HTTP headers.
- Your webbot will, if asked, tell the remote server that you do not have a local authentication certificate. (This is only important if you access a website employing SSL encryption, which is used to protect confidential information on e-commerce websites.)

These defaults are set at the beginning of the file. Feel free to change any of these settings to meet your specific needs.

Using LIB_http

The LIB_http library provides a set of wrapper functions that simplify complicated PHP/CURL interfaces. Each of these interfaces calls a common routine, http(), which performs the specified task, using the values passed to it by the wrapper interfaces. All functions in LIB_http share a similar format: A target and referring URL are passed, and an array is returned, containing the contents of the requested file, transfer status, and error conditions.

While LIB_http has many functions, we'll restrict our discussion to simply fetching files from the Internet using HTTP. The remaining features are described as needed throughout the book.

http_get()

The function http_get() downloads files with the GET method; it has many advantages over PHP's built-in functions for downloading files from the Internet. Not only is the interface simple, but this function offers all the previously described advantages of using PHP/CURL. The script in Listing 3-4 shows how files are downloaded with http_get().

```
# Usage: http_get()
array http_get (string target_url, string referring_url)
```

Listing 3-4: Using http_get()

These are the inputs for the script in Listing 3-4:

target_url is the fully formed URL of the desired file.

referring_url is the fully formed URL of the referer.

These are the outputs for the script in Listing 3-4:

$array['FILE'] contains the contents of the requested file.

$array['STATUS'] contains status information regarding the file transfer.

$array['ERROR'] contains a textual description of any errors.

http_get_withheader()

When a web agent requests a file from the Web, the server returns the file contents, as discussed in the previous section, along with the *HTTP header*, which describes various properties related to a web page. Browsers and webbots rely on the HTTP header to determine what to do with the contents of the downloaded file.

The data that is included in the HTTP header varies from application to application, but it may define cookies, the size of the downloaded file, redirections, encryption details, or authentication directives. Since the information in the HTTP header is critical to properly using a network file, `LIB_http` configures PHP/CURL to automatically handle the more common header directives. Listing 3-5 shows how this function is used.

```
# Usage: http_get_withheader()
array http_get_withheader (string target_url, string referring_url)
```

Listing 3-5: Using `http_get()`

These are the inputs for the script in Listing 3-5:

target_url is the fully formed URL of the desired file.

referring_url is the fully formed URL of the referer.

These are the outputs for the script in Listing 3-5:

$array['FILE'] contains the contents of the requested file, including the HTTP header.

$array['STATUS'] contains status information about the file transfer.

$array['ERROR'] contains a textual description of any errors.

The example in Listing 3-6 uses the `http_get_withheader()` function to download a file and display the contents of the returned array.

```
# Include http library
include("LIB_http.php");

# Define the target and referer web pages
$target = "http://www.schrenk.com/publications.php";
$ref    = "http://www.schrenk.com";

# Request the header
$return_array = http_get_withheader($target, $ref);

# Display the header
echo "FILE CONTENTS \n";
var_dump($return_array['FILE']);
```

```
echo "ERRORS \n";
var_dump($return_array['ERROR']);

echo "STATUS \n";
var_dump($return_array['STATUS']);
```

Listing 3-6: Using `http_get_withheader()`

The script in Listing 3-6 downloads the page and displays the requested page, any errors, and a variety of status information related to the fetch and download.

Listing 3-7 shows what is produced when the script in Listing 3-6 is executed, with the array that includes the page header, error conditions, and status. Notice that the contents of the returned file are limited to only the HTTP header, because we requested only the header and not the entire page. Also, notice that the first line in a HTTP header is the *HTTP code*, which indicates the status of the request. An HTTP code of 200 tells us that the request was successful. The HTTP code also appears in the status array element.[4]

```
FILE CONTENTS
string(215) "HTTP/1.1 200 OK
Date: Sat, 08 Oct 2011 16:38:51 GMT
Server: Apache/2.0.53 (FreeBSD) mod_ssl/2.0.53 OpenSSL/0.9.7g PHP/5
X-Powered-By: PHP/5
Content-Type: text/html; charset=ISO-8859-1

"
ERRORS
string(0) ""

STATUS
array(20) {
  ["url"]=>
  string(39) "http://www.schrenk.com/publications.php"
  ["content_type"]=>
  string(29) "text/html; charset=ISO-8859-1"
  ["http_code"]=>
  int(200)
  ["header_size"]=>
  int(215)
  ["request_size"]=>
  int(200)
  ["filetime"]=>
  int(-1)
  ["ssl_verify_result"]=>
  int(0)
  ["redirect_count"]=>
  int(0)
```

[4] A complete list of HTTP codes can be found in Appendix B.

```
  ["total_time"]=>
  float(0.683)
  ["namelookup_time"]=>
  float(0.005)
  ["connect_time"]=>
  float(0.101)
  ["pretransfer_time"]=>
  float(0.101)
  ["size_upload"]=>
  float(0)
  ["size_download"]=>
  float(0)
  ["speed_download"]=>
  float(0)
  ["speed_upload"]=>
  float(0)
  ["download_content_length"]=>
  float(0)
  ["upload_content_length"]=>
  float(0)
  ["starttransfer_time"]=>
  float(0.683)
  ["redirect_time"]=>
  float(0)
}
```

Listing 3-7: File contents, errors, and the download status array returned by `LIB_http`

The information returned in `$array['STATUS']` is extraordinarily useful for learning how the fetch was conducted. Included in this array are values for download speed, access times, and file sizes—all valuable when writing diagnostic webbots that monitor the performance of a website.

Learning More About HTTP Headers

When a Content-Type line appears in an HTTP header, it defines the *MIME*, or the media type of file sent from the server. The MIME type tells the web agent what to do with the file. For example, the Content-Type in the previous example was *text/html*, which indicates that the file is a web page. Knowing if the file they just downloaded was an image or an HTML file helps browsers know if they should display the file as text or render an image. For example, the HTTP header information for a JPEG image is shown in Listing 3-8.

```
HTTP/1.1 200 OK
Date: Wed, 23 Mar 2011 00:06:13 GMT
Server: Apache/1.3.12 (Unix) mod_throttle/3.1.2 tomcat/1.0 PHP/5
Last-Modified: Wed, 23 Jul 2008 18:03:29 GMT
ETag: "74db-9063-3d3eebf1"
Accept-Ranges: bytes
Content-Length: 36963
Content-Type: image/jpeg
```

Listing 3-8: An HTTP header for an image file request

Examining LIB_http's Source Code

Most webbots in this book will use the library `LIB_http` to download pages from the Internet. If you plan to explore any of the webbot examples that appear later in this book, you should obtain a copy of this library; the latest version is available for download at this book's website. We'll explore some of the defaults and functions of `LIB_http` here.

LIB_http Defaults

At the very beginning of the library is a set of defaults, as shown in Listing 3-9.

```
define("WEBBOT_NAME", "Test Webbot");       # How your webbot will appear in server logs
define("CURL_TIMEOUT", 25);                 # Time (seconds) to wait for network response
define("COOKIE_FILE", "c:\cookie.txt");     # Location of cookie file
```

Listing 3-9: `LIB_http` defaults

LIB_http Functions

The functions shown in Listing 3-10 are available within `LIB_http`. All of these functions return the array defined earlier, containing downloaded files, error messages, and the status of the file transfer.

```
http_get($target, $ref)                                 # Simple get request (w/o header)
http_get_withheader($target, $ref)                      # Simple get request (w/ header)
http_get_form($target, $ref, $data_array)               # Form (method ="GET", w/o header)
http_get_form_withheader($target, $ref, $data_array) # Form (method ="GET", w/ header)
http_post_form($target, $ref, $data_array)              # Form (method ="POST", w/o header)
http_post_withheader($target, $ref, $data_array)     # Form (method ="POST", w/ header)
http_header($target, $ref)                              # Only returns header
```

Listing 3-10: `LIB_http` functions

Final Thoughts

Some of these functions use an additional input parameter, `$data_array`, when form data is passed from the webbot to the webserver. These functions are listed below:

- `http_get_form()`
- `http_get_form_withheader()`
- `http_post_form()`
- `http_post_form_withheader()`

If you don't understand what all these functions do now, don't worry. Their use will become familiar to you as you go through the examples that appear later in this book. Now might be a good time to thumb through Appendix A, which details the features of PHP/CURL that webbot developers are most apt to need.