php|architect's Guide to Web Scraping with PHP

Matthew Turland

nb™ php|architect nanobooks

php|architect’s Guide to Web Scraping with PHP

by Matthew Turland

php|architect's Guide to Web Scraping

First Edition: March 2010
ISBN: **978-0-9810345-1-5**
Produced in Canada
Printed in the United States

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided ”as-is” and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavoured to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

Written by	Matthew Turland
Published by	Marco Tabini & Associates, Inc. 28 Bombay Ave. Toronto, ON M3H 1B7 Canada (416) 630-6202 / (877) 630-6202 info@phparch.com / www.phparch.com
Publisher	Marco Tabini
Technical Reviewer	Luke Giuliani
Layout and Design	Arbi Arzoumani
Managing Editor	Beth Tucker Long
Finance and Resource Management	Emanuela Corso
Cover picture	Vladimir Fofanov

Contents

Credits

I know this section probably isn't why you purchased this book, but I encourage you to read on anyway. One of the things I learned during my time in college is that, even if I don't retain all that is taught to me, having even a vague awareness of that knowledge serves to broaden and improve my perspective. I believe the same is true of knowing more about the author when reading a book they've written, in particular the people who contributed to that person's life and perseverence through the completion of their work. The people described here are good people and they deserve to be remembered.

Whitney

I've spent many nights and weekends writing this book. It was a time sink for months. It was also part of my own personal quest to better myself, show my capabilities, and share something I had to offer. I can't imagine that's an easy thing for a spouse to understand or support for as long as it took to finish the book, but my wife did. That's not something for which I can offer sufficient words to describe, let alone recompense, so I won't try. I'll simply say that I hope you enjoy this book because, were it not for her, it would not exist.

MTA

Before I really started working on the book, I tried pitching the abstract to a few of the bigger names in technical publishing with no success. It ended up being for the better, though, because the PHP community is my target audience. In hindsight, I don't think I could have ended up with a better publisher to that end than php|architect. At the time, I only had a single article on the subject in php|architect Magazine to my name. Liz reviewed the proposal and pitched it to Marco, who made the decision to accept it, and Beth reviewed it once the initial manuscript was completed. Each had a hand in ensuring the book became a polished published work and I'll be thankful for their efforts long after the book is in print.

Luke

I could not have asked for a better technical editor than Luke Giuliani. He has a meticulousness and exceptional attention to detail that occasioanlly surpassed my own, but for the most part is an enjoyable quality we both share. Even when I got stuck in a rut, he was always there to offer motivation and encouragement. His advisement offered not only corrections and suggestions related to the content of the book itself, but also guidance on its organization and presentation to maximize clarity. I severely doubt that my book would be nearly as polished as the copy you see in front of you had it not been for Luke's participation. I owe him a great deal.

Ben

Ben is a man for whom I have immense respect. He is willing to assert his opinions, including the importance of developing and adhering to web standards. Our discussions on topics from the HTTP protocol to interpretation of the Roy Fielding's dissertation on REST have engaged and inspired me. As the light at the end of the tunnel to this book's completion became visible, I began to consider how much it would merit from a foreword by someone with his expertise when it was published. When I approached him about it and asked if he would do the honors, he happily accepted. As wonderful as it felt to finish the project, his valuation of it made the victory that much sweeter.

Friends

The term "friend" is not one that I use lightly. It has long been a belief of mine that family is not something held within a bloodline, but in how people treat one another, and I hold friends in the same regard. There is a special group of people within the PHP community to whom I apply this term. They are too many to name, but each has at one point or another offered critique, suggestion, or moral support whenever I needed it. I think each will know who they are. However small they might consider what they gave, I hope they know its importance and my gratitude for it.

You

While it can take a while longer to write one, time spent reading a book is also a personal investment. This book has made its way through a long journey of development to your hands. You are its audience. I appreciate the time you're taking to read this book and hope that it proves to be time well spent. Regardless of what your opinion of it might be once you've read it, I encourage you to write a review, discuss it on your preferred social networking site, or e-mail me at `me@matthewturland.com` and let me know what your thoughts are.

Foreword

Web scraping is the future of the Web.

I didn't believe that when Matthew first approached me, asking me to write the foreword to this book. In fact, I thought quite the opposite. *Web scraping? Isn't that an old topic — something we used to do in the early days of web development? Why would I want to read about web scraping? Isn't it unethical?*

And you're probably asking yourself some of the same questions.

So, I started to think about it — about what web scraping really is — and the more I considered it, the more it reminded me of Tim Berners-Lee's vision of a web of linked data, of semantic data, connected together and open for all to share and use. Is not web scraping simply the act of getting data from one source and parsing it to use in your own applications? Is this not the goal of the Semantic Web?

When the Web began, its purpose was to share data. The educational and research communities used the Web to display data and link it through hyperlinks to other data. Since XML and, much less, web services and data feeds did not exist in the early days, it became common practice to write scripts to fetch data from other websites, parse the HTML received, process the data in some way, and then display it on one's own website.

One of my earliest experiences with web scraping was in 1998 when I wanted to display up-to-date news headlines on a website. At the time, a particular news website (which shall remain unnamed) provided HTML snippets of news headlines for its customers to embed on their websites. I, however, was not a customer, yet I figured out a way to grab the raw HTML, parse it, and display it on my website. As unethical as this may be — and I don't advocate this behavior at all — I was participating in what I would later find out is called "web scraping."

Today, there are many services out there providing news feeds, and there are plenty of code libraries that can parse this data. So, with the proliferation of web services and public APIs, why is web scraping still so important to the future of the Web? It is important because of the rise of microformats, Semantic Web technologies, the W3C Linking Open Data Community Project, and the Open Data Movement. Just this year at the TED conference, Tim Berners-Lee spoke of linked data saying, "We want the data. We want unadulterated data. We have to ask for raw data now."

The future of the Web is in providing and accessing raw data. How do we access this raw data? Through web scraping.

Yes, there are legal issues to consider when determining whether web scraping is a technique you want to employ, but the techniques this book describes are useful for accessing and parsing raw data of any kind found on the Web, whether it is from a web service API, an XML feed, RDF data embedded in a web page, microformats in HTML, or plain old HTML itself.

There is no way around it. To be a successful web programmer, you must master these techniques. Make them part of your toolbox. Let them inform your software design decisions. I encourage you to bring us into the future of the Web. Scrape the Web within the bounds of the law, publish your raw data for public use, and demand raw data now!

Ben Ramsey
Atlanta, Georgia
June 28, 2009

Chapter 1

Introduction

If you are looking to get an idea of what this book is about or whether you are a member of its intended audience, this chapter is for you. It will provide some background on the material to be covered in the pages to follow and address some common related concerns that may be pertinent. If you have a good high-level idea of what web scraping is and would like to jump straight into the more technical content in this book, you can safely skip on ahead to the next chapter.

Intended Audience

This book is targeted at developers of an intermediate or advanced level who already have a fair amount of comfort programming with PHP 5. You should be aware of object-oriented programming principles such as inheritance, abstraction, and encapsulation as well as how these principles relate to the PHP 5 object model. The book will detail general concepts and use PHP as a (very qualified) means to the end of illustrating these concepts with practical examples. Knowledge of the HTTP protocol, XML concepts and related PHP extensions, or JavaScript will also prove particularly helpful.

How to Read This Book

If you come across a PHP function with which you are not familiar, try pointing your preferred web browser at `http://php.net/functionname` where `functionname` is the name of the function in question. This will generally direct you to either that function's entry in the manual or the landing page for the manual section that contains that function entry.

Beyond that, the chapters can be read independently, but do interrelate in such a way that reading the book from cover to cover may be useful to form a more complete understanding of the material.

Web Scraping Defined

Web scraping is a process involving the retrieval a semi-structured document from the internet, generally a web page in a markup language such as HTML or XHTML, and analysis of that document in order to extract specific data from it for use in another context. It is commonly (though not entirely accurately) also known as screen scraping. Web scraping does not technically fall within the field of data mining because the latter implies an attempt to discern semantic patterns or trends in large data sets that have already been obtained. Web scraping applications (also called intelligent, automated, or autonomous agents) are concerned only with obtaining the data itself through retrieval and extraction and can involve data sets of significantly varied sizes.

You might be saying to yourself that web scraping sounds a lot like acting as a client for a web service. The difference is in the intended audience of the document and, by proxy, the document's format and structure. Web services, because of their intended purpose, are inherently bound by the requirement to generate valid markup in order to remain useful. They must maintain consistent standard formatting in order for machines to be capable of parsing their output.

Web browsers, on the other hand, are generally a lot more forgiving about handling visual rendering of a document when its markup is not valid. As well, web browsers are intended for human use and the methods in which they consume information do not always fall parallel to the way machines would consume it when using an equivalent web service. This can make development of web scraping applications difficult in some instances. Like the obligation of a web service to generate valid markup, a web browser has certain responsibilities. These include respecting server requests to not index certain pages and keeping the number of requests sent to servers within a reasonable amount.

In short, web scraping is the subset of a web browser's functionality necessary to obtain and render data in a manner conducive to how that data will be used.

Applications of Web Scraping

Though it's becoming more common for web sites to expose their data using web services, the absence of a data source that is tailored to machines and offers all the data of a corresponding web site is still a common situation. In these instances, the web site itself must effectively become your data source, and web scraping can be employed to automate the consumption of the data it makes available. Additionally, web services are also used to transfer information into external data systems. In their absence, web scraping can also be used to integrate with such systems that don't offer web services, but do offer a web-based interface for their users.

Another application of web scraping that is likely more well-known is the development of automated agents known as crawlers, which seek out resources for storage and analysis that will eventually comprise the search results they deliver to you. In the earliest days of the internet, this type of data was sought out manually by human beings, a slow and tedious process which limited how quickly a search engine could expand its offerings. Web scraping provided an alternative to allow computers to do the grunt work of finding new pages and extracting their content.

Lastly, web scraping is one way – not the only way or necessarily the recommended way, but certainly a way – to implement integration testing for web applications. Using its abilities to act as a client in extracting and transmitting data, a web scraping application can simulate the browser activity of a normal user. This can help to ensure that web application output complies with its expected response with respect to the application's requirements.

Appropriate Use of Web Scraping

Some data providers may only offer a web site while others may offer APIs that do not offer equivalent data in a conducive format or at all. Some web services may involve an extensive authentication process or be unavailable for public consumption. Unreliability of web service endpoints compared to web sites may also make them unfeasible to use. It is in situations like these that web scraping becomes a desirable alternative.

It is common practice to avoid making changes that break backward-compatibility with existing applications that use web service APIs, or to version them to allow application vendors time to transition to new revisions. As such, web services are significantly less prone to be altered than the markup structure of pages on a web site, especially without advance warning. This is especially true of sites that change frequently, which can drastically affect the stability of applications that employ web scraping.

In a nutshell, web scraping should be used as a last resort when no other feasible options for acquiring the needed data are available.

Legality of Web Scraping

The answer to this question is a bit extensive and veers off into “legal land.” As such, it is included in Appendix A to avoid detracting from the primary purpose of the book.

Topics Covered

You’re obviously reading chapter 1 now, which provides a brief introduction to web scraping, answers common questions, and leads into the meat of the book.

- Chapter 2 deals with relevant details of the HTTP protocol, as HTTP clients are used in the process of document retrieval. This includes how requests and responses are structured and various headers that are used in each to implement features such as cookies, HTTP authentication, redirection, and more.

- Chapters 3-7 cover specific PHP HTTP client libraries and their features, usage, and advantages and disadvantages of each.
- Chapter 8 goes into developing a custom client library and common concerns when using any library including prevention of throttling, access randomization, agent scheduling, and side effects of client-side scripts.
- Chapter 9 details use of the tidy extension for correcting issues with retrieved markup prior to using other extensions to analyze it.
- Chapters 10-12 review various XML extensions for PHP, compare and contrast the two classes of XML parsers, and provide a brief introduction to XPath.
- Chapter 13 is a study of CSS selectors, comparisons between them and XPath expressions, and information on available libraries for using them to query markup documents.
- Chapter 14 explores regular expressions using the PCRE extension, which can be useful in validating scraped data to ensure the stability of the web scraping application.
- Chapter 15 outlines several general high-level strategies and best practices for designing and developing your web scraping applications.

Chapter 2

HTTP

The first task that a web scraping application must be capable of performing is the retrieval of documents containing the information to be extracted. If you have used a web browser without becoming aware of all that it does “under the hood” to render a page for your viewing pleasure, this may sound trivial to you. However, the complexity of a web scraping application is generally proportional to the complexity of the application it targets for retrieving and extracting data.

For targets consisting of multiple pages or requiring retention of session or authentication information, some level of reverse-engineering is often required to develop a corresponding web scraping application. Like a complex mathematics problem with a very simple answer, the development of web scraping applications can sometimes involve more analysis of the target than work to implement a script capable of retrieving and extracting data from it.

This sort of reconnaisance requires a decent working knowledge of the **HyperText Transfer Protocol** or **HTTP**, the protocol that powers the internet. The majority of this chapter will focus on familiarization with that protocol. The end goal is that you become capable of performing the necessary research to learn how a target application works such that you are capable of writing an application to extract the data you want.

Requests

The HTTP protocol is intended to give two parties a common method of communication: **web clients** and **web servers**. Clients are programs or scripts that send requests to servers. Examples of clients include web browsers, such as Internet Explorer and Mozilla Firefox, and crawlers, like those used by Yahoo! and Google to expand their search engine offerings. Servers are programs that run indefinitely and do nothing but receive and send responses to client requests. Popular examples include Microsoft IIS and the Apache HTTP Server.

You must be familiar enough with the anatomy and nuances of HTTP requests and responses to do two things. First, you must be able to configure and use your preferred client to view requests and responses that pass between it and the server hosting the target application as you access it. This is essential to developing your web scraping application without expending an excessive amount of time and energy on your part.

Second, you must be able to use most of the features offered by a PHP HTTP client library. Ideally, you would know HTTP and PHP well enough to build your own client library or fix issues with an existing one if necessary. In principle, however, you should resort to finding and using an adequate existing library first and constructing one that is reusable as a last resort. We will examine some of these libraries in the next few chapters.

Supplemental References

This book will cover HTTP in sufficient depth as it relates to web scraping, but should not in any respect be considered a comprehensive guide on the subject. Here are a few recommended references to supplement the material covered in this book.

- RFC 2616 HyperText Transfer Protocol – HTTP/1.1 (`http://www.ietf.org/rfc/rfc2616.txt`)
- RFC 3986 Uniform Resource Identifiers (URI): Generic Syntax (`http://www.ietf.org/rfc/rfc3986.txt`)
- “HTTP: The Definitive Guide” (ISBN 1565925092)

- "HTTP Pocket Reference: HyperText Transfer Protocol" (ISBN 1565928628)
- "HTTP Developer's Handbook" (ISBN 0672324547)
- Ben Ramsey's blog series on HTTP (`http://benramsey.com/http-status-codes`)

GET Requests

Let's start with a very simple HTTP request, one to retrieve the main landing page of the Wikipedia web site in English.

```
GET /wiki/Main_Page HTTP/1.1
Host: en.wikipedia.org
```

The individual components of this request are as follows.

- `GET` is the **method** or **operation**. Think of it as a verb in a sentence, an action that you want to perform on something. Other examples of methods include POST and HEAD. These will be covered in more detail later in the chapter.
- `/wiki/Main_Page` is the **Uniform Resource Identifier** or **URI**. It provides a unique point of reference for the **resource**, the object or target of the operation.
- `HTTP/1.1` specifies the HTTP **protocol version** in use by the client, which will be detailed further a little later in this chapter.
- The method, URL, and HTTP version collectively make up the **request line**, which ends with a <CR><LF> (carriage return-line feed) sequence, which corresponds to ASCII characters 13 and 10 or Unicode characters U+000D and U+000A respectively. (See RFC 2616 Section 2.2 for more information.)
- A single **header** `Host` and its associated value `en.wikipedia.org` follow the request line. More header-value pairs may follow.
- Based on the resource, the value of the `Host` header, and the protocol in use (HTTP, as opposed to HTTPS or HTTP over SSL),

`http://en.wikipedia.org/wiki/Main_Page` is the resulting full URL of the requested resource.

> **URI vs URL**
> URI is sometimes used interchangeably with URL, which frequently leads to confusion about the exact nature of either. A URI is used to uniquely identify a resource, indicate how to locate a resource, or both. URL is the subset of URI that does both (as opposed to either) and is what makes them usable by humans. After all, what's the use of being able to identify a resource if you can't access it! See sections 1.1.3 and 1.2.2 of RFC 3986 for more information.

GET is by far the most commonly used operation in the HTTP protocol. According to the HTTP specification, the intent of GET is to request a representation of a resource, essentially to "read" it as you would a file on a file system. Common examples of formats for such representations include HTML and XML-based formats such as XHTML, RSS, and Atom.

In principle, GET should not modify any existing data exposed by the application. For this reason, it is considered to be what is called a **safe operation**. It is worth noting that as you examine your target applications, you may encounter situations where GET operations are used incorrectly to modify data rather than simply returning it. This indicates poor application design and should be avoided when developing your own applications.

Anatomy of a URL

If you aren't already familiar with all the components of a URL, this will likely be useful in later chapters.

```
http://user:pass@www.domain.com:8080/path/to/file.ext?query=&var=value#anchor
```

- `http` is the protocol used to interact with the resource. Another example is `https`, which is equivalent to `http` on a connection using an SSL certificate for encryption.

- `user:pass@` is an optional component used to instruct the client that Basic HTTP authentication is required to access the resource and that `user` and `pass` should be used for the username and password respectively when authenticating. HTTP authentication will be covered in more detail toward the end of this chapter.

- `:8080` is another optional segment used to instruct the client that `8080` is the port on which the web server listens for incoming requests. In the absence of this segment, most clients will use the standard HTTP port 80.
- `/path/to/file.ext` specifies the resource to access.
- `query=&var=value` is the query string, which will be covered in more depth in the next section.
- `#anchor` is the fragment, which points to a specific location within or state of the current resource.

Query Strings

Another provision of URLs is a mechanism called the **query string** that is used to pass **request parameters** to web applications. Below is a GET request that includes a query string and is used to request a form to edit a page on Wikipedia.

```
GET /w/index.php?title=Query_string&action=edit
Host: en.wikipedia.org
```

There are a few notable traits of this URL.

- A question mark denotes the end of the resource path and the beginning of the query string.
- The query string is composed of key-value pairs where each pair is separated by an ampersand.
- Keys and values are separated by an equal sign.

Query strings are not specific to GET operations and can be used in other operations as well. Speaking of which, let's move on.

Query String Limits
Most mainstream browsers impose a limit on the maximum character length of a query string. There is no standardized value for this, but Internet Explorer 7 appears to hold the current least common denominator of 2,047 bytes at the time of this writing. Querying a search engine should turn up your preferred browser's limit. It's rare for this to become an issue during development, but it is a circumstance worth knowing.

POST Requests

The next most common HTTP operation after GET is POST, which is used to submit data to a specified resource. When using a web browser as a client, this is most often done via an HTML form. POST is intended to add to or alter data exposed by the application, a potential result of which is that a new resource is created or an existing resource is changed. One major difference between a GET request and a POST request is that the latter includes a **body** following the request headers to contain the data to be submitted.

```
POST /w/index.php?title=Wikipedia:Sandbox&action=submit HTTP/1.1
Host: en.wikipedia.org

wpAntispam=&wpSection=&wpStarttime=20080719022313&wpEdittime=200807190
22100&&wpScrolltop=&wpTextbox1=%7B%7BPlease+leave+this+line+alone+%28s
andbox+heading%29%7D%7D+%3C%21--+Hello%21+Feel+free+to+try+your+format
ting+and+editing+skills+below+this+line.+As+this+page+is+for+editing+e
xperiments%2C+this+page+will+automatically+be+cleaned+every+12+hours.+
--%3E+&wpSummary=&wpAutoSummary=d41d8cd98f00b204e9800998ecf8427e&wpSav
e=Save+page&wpPreview=Show+preview&wpDiff=Show+changes&wpEditToken=%5C
%2B
```

A single blank line separates the headers from the body. The body should look familiar, as it is formatted identically to the query string with the exception that it is not prefixed with a question mark.

> **URL Encoding**
> One trait of query strings is that parameter values are encoded using percent-encoding or, as it's more commonly known, **URL encoding**. The PHP functions `urlencode` and `urldecode` are a convenient way to handle string values encoded in this manner. Most HTTP client libraries handle encoding request parameters for you. Though it's called URL encoding, the technical details for it are actually more closely associated with the URI as shown in section 2.1 of RFC 3986.

HEAD Requests

Though not common when accessing target web applications, HEAD requests are useful in web scraping applications in several ways. They function in the same way as a GET request with one exception: when the server delivers its response, it will not deliver the resource representation that normally comprises the response body. The reason this is useful is that it allows a client to get at the data present in the response headers without having to download the entire response, which is liable to be significantly larger. Such data can include whether or not the resource is still available for access and, if it is, when it was last modified.

```
HEAD /wiki/Main_Page HTTP/1.1
Host: en.wikipedia.org
```

Speaking of responses, now would be a good time to investigate those in more detail.

Responses

Aside from the first response line, called the **status line**, responses are formatted very similarly to requests. While different headers are used in requests and responses, they are formatted the same way. A blank line separates the headers and the body in both requests and responses. The body may be absent in either depending on what the request operation is. Below is an example response.

```
HTTP/1.0 200 OK
Date: Mon, 21 Jul 2008 02:32:52 GMT
```

```
Server: Apache
X-Powered-By: PHP/5.2.5
Cache-Control: private, s-maxage=0, max-age=0, must-revalidate
Content-Language: en
Last-Modified: Mon, 21 Jul 2008 02:06:27 GMT
Content-Length: 53631
Content-Type: text/html; charset=utf-8
Connection: close

[body...]
```

Aside from headers, the main difference in content between requests and responses is in the contents of the request line versus the status line.

- `1.0` is the minimum HTTP protocol version under which the response can be correctly interpreted by the client.
- `200` is a response **status code** and `OK` is its corresponding human-readable description. It indicates the result of the server attempting to process the request, in this case that the request was successful.

Status codes are divided into five classes distinguished by the first digit of the code. Below is a brief summary of each class. See section 10 of RFC 2616 for further descriptions of circumstances under which specific status codes may be received.

- 1xx Informational: Request received, continuing process.
- 2xx Success: Request received, understood, and accepted.
- 3xx Redirection: Client must take additional action to complete the request.
- 4xx Client Error: Request could not be fulfilled because of a client issue.
- 5xx Server Error: Request was valid but the server failed to process it.

Moving right along, let us examine headers in more depth.

Headers

An all-purpose method of communicating a variety of information related to requests and responses, headers are used by the client and server to accomplish a number of things including retention of state using cookies and identity verification using HTTP authentication. This section will deal with those that are particularly applicable to web scraping applications. For more information, see section 14 of RFC 2616.

Cookies

HTTP is designed to be a stateless protocol. That is, once a server returns the response for a request, it effectively "forgets" about the request. It may log information about the request and the response it delivered, but it does not retain any sense of state for the same client between requests. Cookies are a method of circumventing this using headers. Here is how they work.

- The client issues a request.
- In its response, the server includes a `Set-Cookie` header. The header value is comprised of name-value pairs each with optional associated attribute-value pairs.
- In subsequent requests, the client will include a `Cookie` header that contains the data it received in the `Set-Cookie` response header.

Cookies are frequently used to restrict access to certain content, most often by requiring some form of identity authentication before the target application will indicate that a cookie should be set. Most client libraries have the capability to handle parsing and resending cookie data as appropriate, though some require explicit instruction before they will do so. For more information on cookies, see RFC 2109 or its later (though less widely adopted) rendition RFC 2965.

One of the aforementioned attributes, "expires," is used to indicate when the client should dispose of the cookie and not persist its data in subsequent requests. This attribute is optional and its presence or lack thereof is the defining factor in whether or not the cookie is what's called a **session cookie**. If a cookie has no expiration value set, it will persist for the duration of the client session. For normal web browsers, this is generally when all instances of the browser application have been closed.

Redirection

The `Location` header is used by the server to redirect the client to a URI. In this scenario, the response will most likely include a 3xx class status code (such as 302 Found), but may also include a 201 code to indicate the creation of a new resource. See subsection 14.30 of RFC 2616 for more information.

It is hypothetically possible for a malfunctioning application to cause the server to initiate an infinite series of redirections between itself and the client. For this reason, client libraries often implement a limit on the number of consecutive redirections it will process before assuming that the application being accessed is behaving inappropriately and terminating. Libraries generally implement a default limit, but allow you to override it with your own.

Referring URLs

It is possible for a requested resource to refer to other resources in some way. When this happens, clients traditionally include the URL of the referring resource in the `Referer` header. Yes, the header name is misspelled there and intentionally so. The commonality of that particular misspelling caused it to end up in the official HTTP specification, thereby becoming the standard industry spelling used when referring to that particular header.

There are multiple situations in which the specification of a referer can occur. A user may click on a hyperlink in a browser, in which case the full URL of the resource containing the hyperlink would be the referer. When a resource containing markup with embedded images is requested, subsequent requests for those images will contain the full URL of the page containing the images as the referer. A referer is also specified when redirection occurs, as described in the previous section.

The reason this is relevant is because some applications depend on the value of the `Referer` header by design, which is less than ideal for the simple fact that the header value can be spoofed. In any case, it is important to be aware that some applications may not function as expected if the provided header value is not consistent with what is sent when the application is used in a browser. See subsection 14.36 of RFC 2616 for more information.

Persistent Connections

The standard operating procedure for an HTTP request is as follows.

- A client connects to a server.
- The client sends a request over the established connection.
- The server returns a response.
- The connection is terminated.

When sending multiple consecutive requests to the same server, however, the first and fourth steps in that process can cause a significant amount of overhead. HTTP 1.0 established no solution for this; one connection per request was normal behavior. Between the releases of the HTTP 1.0 and 1.1 standards, a convention was informally established that involved the client including a `Connection` header with a value of `Keep-Alive` in the request to indicate to the server that a persistent connection was desired.

Later, 1.1 was released and changed the default behavior from one connection per request to persist connections. For a non-persistent connection, the client could include a `Connection` header with a value of `close` to indicate that the server should terminate the connection after it sent the response. The difference between 1.0 and 1.1 is an important distinction and should be a point of examination when evaluating both client libraries and servers hosting target applications so that you are aware of how they will behave with respect to persistent connections. See subsection 8.1 of RFC 2616 for more information.

There is an alternative implementation that gained significantly less support in clients and servers involving the use of a `Keep-Alive` header. Technical issues with this are discussed in subsection 19.7.1 of RFC 2068, but explicit use of this header should be avoided. It is mentioned here simply to make you aware that it exists and is related to the matter of persistent connections.

Content Caching

Two methods exist to allow clients to query servers in order to determine if resources have been updated since the client last accessed them. Subsections of RFC 2616 section 14 detail related headers.

The first method is time-based where the server returns a `Last-Modified` header (subsection 29) in its response and the client can send that value in an `If-Modified-Since` header (subsection 25) in a subsequent request for the same resource.

The other method is hash-based where the server sends a hash value in its response via the `ETag` header (subsection 19) and the client may send that value in an `If-None-Match` header (subsection 26) in a subsequent request for the same resource.

If the resource has not changed in either instance, the server simply returns a 304 Not Modified response. Aside from checking to ensure that a resource is still available (which will result in a 404 response if it is not), this is an appropriate situation in which to use a HEAD request.

Alternatively, the logic of the first method can be inverted by using an `If-Unmodified-Since` header (subsection 28), in which case the server will return a 412 Precondition Failed response if the resource has in fact been modified since the provided access time.

User Agents

Clients are sometimes referred to as user agents. This refers to the fact that web browsers are agents that act on behalf of users in order to require minimal intervention on the user's part. The `User-Agent` header enables the client to provide information about itself, such as its name and version number. Crawlers often use it to provide a URL for obtaining more information about the crawler or the e-mail address of the crawler's operator. A simple search engine query should reveal a list of user agent strings for mainstream browsers. See subsection 14.43 of RFC 2616 for more information.

Unfortunately, some applications will engage in a practice known as **user agent sniffing** or **browser sniffing** in which they vary the responses they deliver displayed based on the user agent string provided by the client. This can include completely disabling a primary site feature, such as an e-commerce checkout page that uses ActiveX (a technology specific to Windows and Internet Explorer).

One well-known application of this technique is the robots exclusion standard, which is used to explicitly instruct crawlers to avoid accessing individual resources or the entire web site. More information about this is available at `http://www.robotstxt.org`. The guidelines detailed there should definitely be accounted for when developing a web scraping application so as to prevent it from exhibiting behavior inconsistent with that of a normal user.

In some cases, a client practice called **user agent spoofing** involving the specification of a false user agent string is enough to circumvent user agent sniffing, but not always. An application may have platform-specific requirements that legitimately warrant it denying access to certain user agents. In any case, spoofing the user agent is a practice that should be avoided to the fullest extent possible.

Ranges

The `Range` request header allows the client to specify that the body of the server's response should be limited to one or more specific byte ranges of what it would normally be. Originally intended to allow failed retrieval attempts to resume from their stopping points, this feature can allow you to minimize data transfer between your application and the server to reduce bandwidth consumption and runtime of your web scraping application.

This is applicable in cases where you have a good rough idea of where your target data is located within the document, especially if the document is fairly large and you only need a small subset of the data it contains. However, using it does add one more variable to the possibility of your application breaking if the target site changes and you should bear that in mind when electing to do so.

While the format of the header value is being left open to allow for other range units, the only unit supported by HTTP/1.1 is bytes. The client and server may both use the `Accept-Ranges` header to indicate what units they support. The server will include the range (in a slightly different format) of the full response body in which the partial response body is located using the `Content-Range` header.

In the case of bytes, the beginning of the document is represented by 0. Ranges use inclusive bounds. For example, the first 500 bytes of a document would be specified as 0-499. To specify from a point to the end of the document, simply exclude the later bound. The portion of a document beginning from the byte 500 going to its end is represented as 500-.

If a range is specified that is valid with respect to the resource being requested, the server should return a 206 Partial Content response. Otherwise, it should return a 416 Requested Range Not Satisfiable response. See sections 14.35 and 14.16 of RFC 2616 for more information on the `Range` and `Content-Range` headers respectively.

Basic HTTP Authentication

Another less frequently used method of persisting identity between requests is HTTP authentication. Most third-party clients offer some form of native support for it. It's not commonly used these days, but it's good to be aware of how to derive the appropriate header values in cases where you must implement it yourself. For more information on HTTP authentication, see RFC 2617.

HTTP authentication comes in several "flavors," the more popular two being Basic (unencrypted) and Digest (encrypted). Basic is the more common of the two, but the process for both goes like this.

- A client sends a request without any authentication information.
- The server sends a response with a 401 status code and a `WWW-Authenticate` header.
- The client resends the original request, but this time includes an `Authorization` header including the authentication credentials.
- The server either sends a response indicating success or one with a 403 status code indicating that authentication failed.

In the case of Basic authentication, the value of the `Authorization` header will be the word `Basic` followed by a single space and then by a Base64-encoded sequence derived from the username-password pair separated by a colon. If, for example, the username is `bigbadwolf` and the password is `letmein` then the value of the header would be `Basic YmlnYmFkd29sZjpsZXRtZWlu` where the Base64-encoded version of the string `bigbadwolf:letmein` is what follows `Basic`.

Digest HTTP Authentication

Digest authentication is a bit more involved. The `WWW-Authenticate` header returned by the server will contain the word “Digest” followed by a single space and then by a number of key-value pairs in the format `key=“value”` separated by commas. Below is an example of this header.

```
WWW-Authenticate: Digest realm="testrealm@host.com",
                         qop="auth,auth-int",
                         nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                         opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

The client must respond with a specific response value that the server will verify before it allows the client to proceed. To derive that value requires use of the MD5 hash algorithm, which in PHP can be accessed using the `md5` or `hash` functions. Here is the process.

- Concatenate the appropriate username, the value of the realm key provided by the server, and the appropriate password together separated by colons and take the MD5 hash of that string. We’ll call this HA1. It shouldn’t change for the rest of the session.

```
<?php
$ha1 = md5($username . ’:testrealm@host.com:’ . $password);
?>
```

- Concatenate the method and URI of the original request separated by a colon and take the MD5 hash of that string. We’ll call this HA2. This will obviously vary with your method or URI.

```
<?php
$ha2 = md5(’GET:/wiki/Main_Page’);
?>
```

- Initialize a request counter that we'll call nc with a value of 1. The value of this counter will need to be incremented and retransmitted with each subsequent request to uniquely identify it. Retransmitting a request counter value used in a previous request will result in the server rejecting it. Note that the value of this counter will need to be expressed to the server as a hexadecimal number. The `dechex` PHP function is useful for this.

```
<?php
$nc = 1;
?>
```

- Generate a random hash using the aforementioned hashing functions that we'll call the client nonce or cnonce. The `time` and `rand` functions may be useful here. This can (and probably should) be regenerated and resent with each request.

```
<?php
$cnonce = md5($_SERVER['REMOTE_ADDR'] . microtime(true));
?>
```

- Take note of the value of the nonce key provided by the server, also known as the server nonce. We'll refer to this as simply the nonce. This is randomly generated by the server and will expire after a certain period of time, at which point the server will respond with a 401 status code. It will modify the `WWW-Authenticate` header it returns in two noteworthy ways: 1) the key-value pair `stale=TRUE` will be added; 2) the `nonce` value will be changed. When this happens, simply rederive the response code as shown below with the new nonce value and resubmit the original request (not forgetting to increment the request counter).

- Concatenate HA1, the server nonce (nonce), the current request counter (nc) value, the client nonce you generated (cnonce), an appropriate value (most likely “auth”) from the comma-separated list contained in the qop (quality of protection) key provided by the server, and HA2 together separated by colons and take the MD5 hash of that string. This is the final response code.

```
<?php
$response = implode(’:’, array(
    $ha1,
    $nonce,
    dechex($nc),
    $cnonce,
    ’auth’,
    $ha2
));
?>
```

- Lastly, send everything the server originally sent in the `WWW-Authenticate` header, plus the response value and its constituents (except the password obviously), back to the server in the usual `Authorization` header.

```
Authorization: Digest username="USERNAME",
                      realm="testrealm@host.com",
                      nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                      uri="/wiki/Main_Page",
                      qop="auth",
                      nc=00000001,
                      cnonce="0a4f113b",
                      response="6629fae49393a05397450978507c4ef1",
                      opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Some third-party clients implement this, some don’t. Again, it’s not commonly used, but it’s good to be aware of how to derive the appropriate header value in cases where you must implement it yourself. For more information on HTTP authentication, see RFC 2617.

Wrap-Up

At this point, you should feel comfortable looking at HTTP requests and responses and be capable deducing information about them by analyzing their individual components. The next few chapters will expound upon this information by reviewing several commonly used PHP HTTP client implementations.

Chapter 3

HTTP Streams Wrapper

At this point, you should be fairly well-acquainted with some of the general concepts involved in using an HTTP client. The next few chapters will review some of the more popular mainstream client libraries, particularly common use cases and the advantages and disadvantages of each. This client covered in this chapter will be the HTTP streams wrapper.

PHP 4.3 saw the addition of the Streams extension to the core. According to the related section of the PHP manual, the intention was to provide “a way of generalizing file, network, data compression, and other operations which share a common set of functions and uses.” Streams introduced several concepts, one of which is a **wrapper**. The job of a wrapper is to define how a stream handles communications in a specific protocol or using a specific encoding. One such protocol for which a wrapper is available is HTTP.

The advantage to the HTTP streams wrapper is that there is very little to learn in terms of the API. It’s fairly easy to get something simple working quickly. The disadvantage is that it’s very minimalistic in terms of the feature set offered. It gives you the ability to send HTTP requests without having to construct them entirely on your own (by specifying the body and optionally any headers you want to add) and access data in the response. That’s about it. The ability to debug requests is one example of a feature that it does not include at the time of writing.

The fact that the wrapper is written in C is a bit of a double-edged sword. On the positive side, there is a substantial performance difference between C code and PHP code (though it is more noticeable in a high load environment). On the negative side, you have to either know C or depend on the community to deliver patches in a timely fashion for any issues that may arise. This also applies to extensions written in C that will be covered in subsequent sections.

The streams wrapper is part of the PHP core and as such has no installation requirements beyond that of PHP itself.

Simple Request and Response Handling

Here's a simple example of the HTTP streams wrapper in action.

```
<?php
$response = file_get_contents('http://localhost.example');
print_r($http_response_header);
?>
```

There are a few things to note.

- The `allow_url_fopen` PHP configuration setting must be enabled for this to work, which it is in most environments.
- In this example, the `file_get_contents` function call is equivalent to making a GET request for the specified URL '`http://localhost.example`'.
- `$response` will contain the response body after the call to the `file_get_contents` function completes.
- `$http_response_header` is implicitly populated with the HTTP response status line and headers after the `file_get_contents` call because it uses the HTTP streams wrapper *within the current scope*.

While this example does work, it violates a core principle of good coding practices: no unexpected side effects. The origin of `$http_response_header` is not entirely obvious because PHP populates it implicitly. Additionally, it's more restrictive because the variable is only populated within the scope containing the call to `file_get_contents`. Here's a better way to get access to the same data from the response headers.

```
<?php
$handle = fopen('http://localhost.example', 'r');
$response = stream_get_contents($handle);
$meta = stream_get_meta_data($handle);
print_r($meta['wrapper_data']);
?>
```

Let's step through this.

- The resource `$handle` is created to read from the URL `http://localhost.example`.
- The `stream_get_contents` function is called to read the remaining data on the stream pointed to by the `$handle` resource into `$response`.
- The `stream_get_meta_data` function is called to read metadata for the stream pointed to by the `$handle` resource into `$meta`.
- The `wrapper_data` index of the `$meta` array outputs the same array as `$http_response_header` would within the current scope. So long as `$handle` is accessible within the current scope, `stream_get_meta_data()` can be called on it. This makes it more flexible than `$http_response_header`.

Stream Contexts and POST Requests

Another concept introduced by streams is the **context**, which is basically a set of configuration options used in a streams operation. A context is created by passing an associative array of context options and their corresponding values to the `stream_context_create` function. One use of contexts with regard to the HTTP streams wrapper is making POST requests, as the wrapper uses the GET method by default.

```
<?php
$context = stream_context_create(array(
    'http' => array(
        'method' => 'POST',
        'header' => implode("\r\n", array(
            'Content-Type: application/x-www-form-urlencoded',
                'Referer: http://localhost.example'
            )),
            'content' => http_build_query(array(
                'param1' => 'value1',
                'param2' => 'value2'
            ))
        )
    ));

    $response = file_get_contents(
        'http://localhost.example/process',
        false,
        $context
    );
    ?>
```

Here is a walk-through of this example.

- `'http`' is the streams wrapper being used.
- `'POST`' is the HTTP method of the request.
- The `'header`' stream context setting is populated with a string containing HTTP header key-value pairs, in this case for the `Content-Type` and `Referer` HTTP headers. The `Content-Type` header is used to indicate that the request body data is URL-encoded. When multiple custom headers are needed, they must be separated by a carriage return-line feed (`"\r\n"` also known as CRLF) sequence. The `implode` function is useful for this if key-value pairs for headers are stored in an enumerated array.
- The `http_build_query` function is being used to construct the body of the request. This function can also be used to construct query strings of URLs for GET requests.
- `http://localhost.example/process` is the URL of the resource being requested.
- `file_get_contents` is called to execute the request, the options for which are passed via the context `$context` created using `stream_context_create`.
- The body of the response is returned and stored in the variable `$response`.

Error Handling

Before PHP 5.3.0, an HTTP streams wrapper operation resulting in an HTTP error response (i.e. a 4xx or 5xx status code) causes a PHP-level warning to be emitted. This warning will only contain the HTTP version, the status code, and the status code description. The function calls for such operations generally return `false` as a result and leave you without a stream resource to check for more information. Here's an example of how to get what data you can.

```
<?php
function error_handler($errno, $errstr, $errfile, $errline,
    array $errcontext) {

    // $errstr will contain something like this:
    // fopen(http://localhost.example/404): failed to open stream:
    // HTTP request failed! HTTP/1.0 404 Not Found

    if ($httperr = strstr($errstr, 'HTTP/')) {

        // $httperr will contain HTTP/1.0 404 Not Found in the case
        // of the above example, do something useful with that here
    }
}

set_error_handler('error_handler', E_WARNING);

// If the following statement fails, $stream will be assigned false
// and error_handler will be called automatically
$stream = fopen('http://localhost.example/404', 'r');

// If error_handler() does not terminate the script, control will
// be returned here once it completes its execution
restore_error_handler();
?>
```

This situation has been improved somewhat in PHP 5.3 with the addition of the `ignore_errors` context setting. When this setting is set to `true`, operations resulting in errors are treated the same way as successful operations. Here's an example of what that might look like.

```
<?php
```

```
$context = stream_context_create(
    array(
        'http' => array(
            'ignore_errors' => true
        )
    )
);

$stream = fopen('http://localhost.example/404', 'r', false, $context);

// $stream will be a stream resource at this point regardless of
// the outcome of the operation
$body = stream_get_contents($stream);
$meta = stream_get_meta_data($stream);

// $meta['wrapper_data'][0] will equal something like HTTP/1.0 404
// Not Found at this point, with subsequent array elements being
// other headers
$response = explode(' ', $meta['wrapper_data'][0], 3);
list($version, $status, $description) = $response;
switch (substr($status, 0, 1)) {
    case '4':
    case '5':
        $result = false;
    default:
        $result = true;
}
?>
```

HTTP Authentication

The HTTP stream wrapper has no context options for HTTP authentication credentials. However, it is possible to include the credentials as part of the URL being accessed. See the example below. Note that credentials are not pre-encoded; this is handled transparently when the request is made. Also, note that this feature only works when Basic HTTP authentication is used; Digest authentication must be handled manually.

```
<?php
$response = file_get_contents(
    'http://username:password@localhost.example'
);
?>
```

A Few More Options

Below are a few other stream context options for the HTTP streams wrapper that may prove useful.

- `'user_agent'` allows you to set the user agent string to use in the operation. This can also be set manually by specifying a value for the `User-Agent` header in the `'header'` context option value.
- `'max_redirects'` is used to set the maximum number of redirections that the operation will process prior to assuming that the application is misbehaving and terminating the request. This option is only available in PHP 5.1.0 and up and uses a default value of 20.
- `'timeout'` is used to set a maximum limit on the amount of time in seconds that a read operation may be allowed to execute before it is terminated. It defaults to the value of the `default_socket_timeout` PHP configuration setting.

All other features utilizing headers must be implemented manually by specifying request headers in the `'header'` context option and checking either `$http_response_header` or the `'wrapper_data'` index of the array returned by the `stream_get_meta_data` function for response headers.

Wrap-Up

For more information about the HTTP streams wrapper itself, see `http://www.php.net/manual/en/wrappers.http.php`.

For details about the context options specific to the wrapper, see `http://www.php.net/manual/en/context.http.php`.

Chapter 4

cURL Extension

The cURL PHP extension, available since PHP 4.0.2, wraps a library called libcurl that implements client logic for a variety of internet protocols including HTTP and HTTPS. Its API is fairly small and applications of it consist mostly of calls to set configuration options and their respective values.

cURL assumes fewer default configuration values than the HTTP streams wrapper, like whether or not to process redirections and how many of them to process. The disadvantage to this is that, combined with the rich feature set of the extension, PHP code to use cURL is often more verbose than equivalent code using other client libraries.

Like the streams wrapper, the cURL extension is written in C and has the same pros and cons in that respect. cURL uses a **session handle** (of the PHP resource data type) with which configuration settings can be associated in a similar fashion to how they are associated with contexts for stream wrappers. Also like stream contexts, cURL session handles can be used multiple times to repeat the same operation until passed to the `curl_close` function.

The cURL extension is included in the PHP core, but must either be compiled into PHP or installed as a separate extension. Depending on the runtime environment's operating system, this may involve installing a package in addition to the OS PHP package.

Simple Request and Response Handling

```
<?php
$ch = curl_init('http://localhost.example/');
curl_setopt($ch, CURLOPT_RETURNTRANSER, true);
$response = curl_exec($ch);
curl_close($ch);
?>
```

Let's look at this line by line.

- `curl_init` is called and passed '`http://localhost.example/path/to/form`' as the URL for the request. Note this parameter is optional and can also be specified by calling `curl_setopt` with the cURL session handle (`$ch` in this case), the `CURLOPT_URL` constant, and the URL string.

- `curl_setopt` is called to set the configuration setting represented by the `CURLOPT_RETURNTRANSFER` parameter to have a value of `true`. This setting will cause `curl_exec` to return the HTTP response body in a string rather than outputting it directly, the latter being the default behavior.

- `curl_exec` is called to have it execute the request and return the response body.

- `curl_close` is called to explicitly close the cURL session handle, which will no longer be reusable after that point.

A useful setting worth mentioning early on is `CURLOPT_VERBOSE`, which outputs debugging information when set to `true`. This output is sent to either stderr (the default) or the file referenced by the value of the `CURLOPT_STDERR`.

Contrasting GET and POST

Obviously the cURL extension has other functions, but by and large most HTTP requests made using the cURL extension will follow the sequence of operations shown in the above example. Let's compare this with a POST request.

```
<?php
```

```
$data = array(
    'param1' => 'value1',
    'param2' => 'value2',
    'file1' => '@/path/to/file',
    'file2' => '@/path/to/other/file'
);

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'http://localhost.example/process');
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);
curl_close($ch);
?>
```

Here are the differences between this example and the previous one.

- The URL is passed using the `curl_setopt` function this time, just to show how to do it without passing it to the `curl_init` function. This is important when reusing cURL session handles with different URLs.

- `CURLOPT_POST` is set to `true` to change the request method to POST.

- `CURLOPT_POSTFIELDS` is an associative array or preformatted query string to be used as the data for the request body. Files can be uploaded by specifying a value where the first character is `@` and the remainder of the value is a filesystem path to the file intended for upload.

Here are a few other cURL configuration setting constants related to the request method.

- `CURLOPT_HTTPGET`: Boolean that, when set to `true`, explicitly resets the request method to GET if it's been changed from the default.

- `CURLOPT_NOBODY`: Boolean that, when set to `true`, excludes the body from the response by changing the request method to HEAD.

Setting Multiple Options

If you're working with PHP 5.1.3+ you have access to the `curl_setopt_array` function, which allows you to pass in an associative array of setting-value pairs to set with a single function call. If you're working with an older version of PHP, the function is relatively easy to write.

Using this function results in not only less and cleaner code, but in the case of the native C function it also results in fewer function calls and by proxy improved performance.

```
<?php
if (!function_exists('curl_setopt_array')) {
    function curl_setopt_array($ch, $options) {
        foreach ($options as $setting => $value) {
            curl_setopt($ch, $setting, $value);
        }
    }
}

$ch = curl_init();
$opts = array(
    CURLOPT_URL => 'http://localhost.example',
    CURLOPT_RETURNTRANSFER => true
);
curl_setopt_array($ch, $opts);
$response = curl_exec($ch);
curl_close($ch);
?>
```

Handling Headers

`CURLOPT_HEADER` holds a boolean flag that, when set to `true`, will cause headers to be included in the response string returned by `curl_exec`.

Another option for getting at some of the data included in the response headers, such as the HTTP response code, is to use the `curl_getinfo` function as shown in the following example. For more on what other information this function offers, see its entry in the PHP manual.

```
<?php
$ch = curl_init();
// ...
$response = curl_exec($ch);
$info = curl_getinfo($ch);
$responsecode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
?>
```

CURLOPT_HTTPHEADER holds an enumerated array of custom request header name-value pairs formatted like so.

```
<?php
$ch = curl_init();
curl_setopt($ch, CURLOPT_HTTPHEADER, array(
    'Accept-Language: en-us,en;q=0.5',
    'Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7',
    'Keep-Alive: 300',
    'Connection: keep-alive'
));
?>
```

Debugging

Previously mentioned in the last section on handling headers, the curl_getinfo function also enables you to view requests being sent by cURL. This can be quite useful when debugging. Below is an example of this feature in action.

```
<?php
$ch = curl_init();
curl_setopt_array(array(
    CURLOPT_RETURNTRANSFER => true,
    CURLINFO_HEADER_OUT => true
));
curl_exec($ch);
$request = curl_getinfo($ch, CURLINFO_HEADER_OUT);
?>
```

- `CURLOPT_RETURNTRANSFER` is set to `true` in the `curl_setopt_array` call even though the return value of `curl_exec` isn’t captured. This is simply to prevent unwanted output.

- `CURLINFO_HEADER_OUT` is set to `true` in the `curl_setopt_array` call to indicate that the request should be retained because it will be extracted after the request is made.

- `CURLINFO_HEADER_OUT` is specified in the `curl_getinfo` call to limit its return value to a string containing the request that was made.

Cookies

```
<?php
$cookiejar = ’/path/to/file’;

$ch = curl_init();

$url = ’http://localhost.example’;
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_COOKIEJAR, $cookiejar);
curl_exec($ch);

$url = ’http://localhost.example/path/to/form’;
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_COOKIEFILE, $cookiejar);
curl_exec($ch);

curl_close($ch);
?>
```

Here is a quick list of pertinent points.

- After the first `curl_exec` call, cURL will have stored the value of the the `Set-Cookie` response header returned by the server in the file referenced by `'/path/to/file`’ on the local filesystem as per the `CURLOPT_COOKIEJAR` setting. This setting value will persist through the second `curl_exec` call.

- When the second `curl_exec` call takes place, the `CURLOPT_COOKIEFILE` setting will also point to `'/path/to/file`’. This will cause cURL to read the contents of that file and use it as the value for the Cookie request header when the request is constructed.

- If $cookiejar is set to an empty string, cookie data will persist in memory rather than a local file. This improves performance (memory access is faster than disk) and security (file storage may be more open to access by other users and processes than memory depending on the server environment).

In some instances it may be desirable for the CURLOPT_COOKIEJAR value to have a different value per request, such as for debugging. In most cases, however, CURLOPT_COOKIEJAR will be set for the first request to receive the initial cookie data and its value will persist for subsequent requests. In most cases, CURLOPT_COOKIEFILE will be assigned the same value as CURLOPT_COOKIEJAR after the first request. This will result in cookie data being read to include in the request, followed by cookie data from the response being written back (and overwriting any existing data at that location) for use in subsequent requests. On a related note, if you want cURL to begin a new session in order to have it discard data for session cookies (i.e. cookies without an expiration date), you can set the CURLOPT_COOKIESESSION setting to true.

If you want to handle cookie data manually for any reason, you can set the value of the Cookie request header via the CURLOPT_COOKIE setting. To get access to the response headers, set the CURLOPT_HEADER and CURLOPT_RETURNTRANSFER settings to true. This will cause the curl_exec call to return the entire response including the headers and the body. Recall that there is a single blank line between the headers and the body and that a colon separates each header name from its corresponding value. This information combined with the basic string handling functions in PHP should be all you need. Also, you'll need to set CURLOPT_FOLLOWLOCATION to false in order to prevent cURL from processing redirections automatically. Not doing this would cause any cookies set by requests resulting in redirections to be lost.

HTTP Authentication

cURL supports both Basic and Digest HTTP authentication methods, among others. The CURLOPT_HTTPAUTH setting controls the method to use and is set using constants such as CURLAUTH_BASIC or CURLAUTH_DIGEST. The CURLOPT_USERPWD setting is a string containing the authentication credentials to use in the format 'username:password'. Note that this has to be set for each request requiring authentication.

Redirection

CURLOPT_FOLLOWLOCATION can be set to true to have cURL automatically place process redirections. That is, it will detect Location headers in the server response and implicitly issue requests until the server response no longer contains a Location header. To set the maximum number of Location headers to have cURL process automatically before terminating, use the CURLOPT_MAXREDIRS setting. To have authentication credentials persist in requests resulting from redirections, set the CURLOPT_UNRESTRICTED_AUTH setting to true.

Referers

CURLOPT_REFERER allows you to explicitly set the value of the Referer header. Setting CURLOPT_AUTOREFERER to true will cause cURL to automatically set the value of the Referer header whenever it processes a Location header.

Content Caching

CURLOPT_TIMECONDITION must be set to either CURL_TIMECOND_IFMODSINCE or CURL_TIMECOND_IFUNMODSINCE to select whether the If-Modified-Since or If-Unmodified-Since header will be used respectively.

CURLOPT_TIMEVALUE must be set to a UNIX timestamp (a date representation using the number of seconds between the UNIX epoch and the desired date) to indicate the last client access time of the resource. The time function can be used to derive this value.

User Agents

CURLOPT_USERAGENT can be used to set the User Agent string to use.

Byte Ranges

CURLOPT_RESUME_FROM can be used to set a single point within the document from which to start the response body. This will cause the Range header to be set with a value of X- where X is the specified starting point.

In order to specify multiple ranges, CURLOPT_RANGE accepts a string in the same format as the Range header (ex: X-Y,Z-).

DNS Caching

You may notice that code using the cURL extension appears to run faster than code using streams. The reason for this is that cURL implements its own DNS cache, which is more likely to be apparent if your operating system or internet service provider does not provide one.

DNS, or Domain Name System, is a system used to derive an IP address for a domain name in a manner similar to how phone directories are used to obtain a phone number for a person using their name. The process of obtaining an IP address for a domain name, called a DNS lookup, can be a costly operation in terms of the time required.

Because the results of DNS lookups don't change often, DNS caching is often used to retain the results of lookups for a certain time period after they are performed. This can be done at multiple levels including the source code level as with cURL, natively at the OS level, or via software like nscd or dnsmasq run either locally or on remote servers such as those used by internet service providers.

cURL DNS caching is enabled by default. Some situations like debugging may warrant disabling it, which can be done by setting CURLOPT_DNS_USE_GLOBAL_CACHE to false. cURL will also by default retain the results of DNS lookups in memory for two minutes. To change this, set the CURLOPT_DNS_CACHE_TIMEOUT setting to the number of seconds a result should remain in the cache before expiring.

Also noteworthy is the fact that cURL DNS caching is not thread-safe. Threading is a particular style of parallel processing. The most common implementation of threading consists of multiple threads of execution contained within a single operating system process that share resources such as memory. Because of this, it may operate unpredictably in a threaded environment such as Windows Server or *NIX running a threaded Apache MPM such as worker.

If you are using the HTTP streams wrapper or either of the PHP-based HTTP client libraries covered in this chapter and you have access to install software on your server, you may want to install a local DNS caching daemon to improve performance. Try nscd or dnsmasq on *NIX. Writing DNS caching into your own client will be covered in a later chapter on writing your own HTTP client.

Timeouts

`CURLOPT_CONNECTTIMEOUT` is a maximum amount of time in seconds to which a connection attempt will be restricted for a cURL operation. It can be set to `0` to disable this limit, but this is inadvisable in a production environment. Note that this time includes DNS lookups. For environments where the DNS server in use or the web server hosting the target application is not particularly responsive, it may be necessary to increase the value of this setting.

`CURLOPT_TIMEOUT` is a maximum amount of time in seconds to which the execution of individual cURL extension function calls will be limited. Note that the value for this setting should include the value for `CURLOPT_CONNECTTIMEOUT`. In other words, `CURLOPT_CONNECTTIMEOUT` is a segment of the time represented by `CURLOPT_TIMEOUT`, so the value of the latter should be greater than the value of the former.

Request Pooling

Because it is written C, the cURL extension has one feature that cannot be replicated exactly in libraries written in PHP: the ability to run multiple requests in parallel. What this means is that multiple requests can be provided to cURL all at once and, rather than waiting for a response to be received for the first request before moving on to sending the second, all requests will be sent and processed as responses are returned. This can significantly shorten the time required to collectively complete all the requests. However, care should be taken not to overload a single host with requests when using this feature.

```
<?php
```

```
$ch1 = curl_init('http://localhost.example/resource1');
curl_setopt($ch1, CURLOPT_RETURNTRANSFER, true);
/* other curl_setopt calls */

$ch2 = curl_init('http://localhost.example/resource2');
curl_setopt($ch2, CURLOPT_RETURNTRANSFER, true);
/* other curl_setopt calls */

$mh = curl_multi_init();
curl_multi_add_handle($mh, $ch1);
curl_multi_add_handle($mh, $ch2);

$running = null;
do {
    curl_multi_exec($mh, $running);
} while ($running > 0);

$ch1_response = curl_multi_getcontent($ch1);
$ch2_response = curl_multi_getcontent($ch2);

curl_multi_remove_handle($mh, $ch1);
curl_close($ch1);

curl_multi_remove_handle($mh, $ch2);
curl_close($ch2);

curl_multi_close($mh);
?>
```

- Two cURL session handles `$ch1` and `$ch2` are initialized and configured normally. Note that more than two can be used in this type of operation; two merely satisfy the purpose of this example.

- A cURL multi handle `$mh` is initialized and the two session handles are added to it.

- A loop is used in conjunction with the flag `$running` to repeatedly check (i.e. poll) the multi handle for completion of all contained operations.

- `curl_multi_getcontent` is used on the two session handles to get the response bodies of each.

- The session handles are individually removed from the multi handle and closed using `curl_multi_remove_handle` and `curl_close` respectively.
- The multi handle is closed using `curl_multi_close`.

Note that this is a fairly simple example that waits until all requests have completed before attempting to process them. If you want to process requests as soon as they return, see the source code at `http://code.google.com/p/rolling-curl/source/browse/#svn/trunk` for an example of how to do this.

Wrap-Up

For more information on the cURL extension, see `http://php.net/manual/en/book.curl.php`.

For information on installing the cURL extension, see `http://php.net/manual/en/curl.installation.php`.

For information on the cURL extension configuration constants, see `http://php.net/manual/en/function.curl-setopt.php`.

Chapter 5

pecl_http PECL Extension

The pecl_http extension became available in PHP 5 and gained a class for HTTP responses in PHP 5.1. Like the cURL extension, it also wraps the libcurl library and has similar advantages as a result, such as its internal DNS cache. While the cURL extension uses resources and configuration options to more closely mimic libcurl's own API, the pecl_http extension offers the same features in the form of procedural and object-oriented APIs. As a result, code that uses pecl_http is generally shorter than equivalent code using the cURL extension.

Another major difference is that the cURL extension is considered to be part of the PHP core, which means that it is more likely to be present in most environments supporting PHP. In contrast, pecl_http must be installed via the pecl installer or compiled manually from source. In either case, the extension requires the PHP and libcurl development headers to be present in order to compile properly. Not technically being part of the PHP core, it is less likely to be present or available in shared hosting environments depending on the hosting provider. This may be a concern for code that is intended to be distributed for use in a variety of environments.

To install pecl_http via the pecl installer, PEAR must be installed (see `http://pear.php.net/manual/en/installation.php`). A package for PEAR may also be available through your OS package manager. Once it is installed, simply issue the command `pecl install pecl_http`. To install from source, download the latest version from `http://pecl.php.net/get/pecl_http`. Note that this requires the header files for the PHP version you are using, which are contained in the PHP source tarballs available for download at `http://www.php.net/downloads.php` or possibly also through your OS package manager.

GET Requests

```
<?php
// Procedural
$response = http_get('http://localhost.example', null, $info);

// Object-oriented
$request = new HttpRequest('http://localhost.example');
$response = $request->send();
?>
```

- After the `http_get` call, `$response` will be a string containing the body of the response and `$info` will contain an associative array with information about the request and response. `null` is used in place of an array of request options that will be covered in more detail shortly.
- The `HttpRequest` block is an object-oriented equivalent of the `http_get` procedural call and offers a more explicit (albeit slightly less concise) API for setting and getting data about the request and response.
- The constructor for the `HttpRequest` class has two additional optional parameters not shown here: a constant representing the request method (the default is `HTTP_METH_GET` for a GET request) and an associative array of request options as included in the `http_get` call.

POST Requests

```
<?php
$data = array(
    'param1' => 'value1',
    'param2' => 'value2'
);
```

```
$files = array(
    array(
        'name' => 'file1',
        'type' => 'image/jpeg',
        'file' => '/path/to/file1.jpg'
    ),
    array(
        'name' => 'file2',
        'type' => 'image/gif',
        'file' => '/path/to/file2.gif'
    )
);

$options = array(
    'referer' => 'http://localhost.example/'
);

// Procedural
$response = http_post_fields(
    'http://localhost.example/process',
    $data,
    $files,
    $options,
    $info
);

// Object-oriented
$request = new HttpRequest;
$request->setUrl('http://localhost.example/process');
$request->setMethod(HTTP_METH_POST);
$request->setPostFields($data);
$request->setPostFiles($files);
$request->setOptions($options);
$response = $request->send();
?>
```

- `http_post_fields`, `setPostFields`, and `setPostFiles` are used to set the request method to POST and to specify that the extension should handle constructing and encoding the POST body with the provided data. If you are handling this aspect of the request yourself, use `http_post_data` or `setRawPostData` instead.

- `setUrl` is used to set the URL of the HttpRequest instance rather than the constructor, just as an example.

- After HttpRequest is instantiated, its request method is explicitly set using the constant HTTP_METH_POST.
- Request options are passed in this time, specifically the value for the Referer header. They are specified as the fourth parameter of http_post_fields and via setOptions.
- As with the GET request, the return values of http_post_fields and send are of the same types (string and HttpMessage) as their respective counterparts in the earlier GET example.
- Because http_post_fields does not have an object to use in retaining state, it instead allows the option to pass in a fifth parameter $info in which it will store information about the request and response. See http://php.net/http_get for an example of the information stored in $info.

Alternative Methods and More Options
For request method constants beyond GET and POST used by the procedural API, see http://php.net/manual/en/http.constants.php. For more on available request options for both APIs, see http://php.net/manual/en/http.request.options.php.

Handling Headers

When using the procedural API, limited request header information is available via the $info parameter of functions to issue requests. Response headers are included in the string returned by those function calls. When the string is passed to http_parse_message, it returns an object with a headers property, which is an associative array of header values indexed by header name.

Custom request headers can be set via the 'headers' request option, which is formatted as an associative array with header names for keys pointing to corresponding header values (as opposed to cURL, which uses an enumerated array with one header name-value pair per array item). Below is an example of custom headers in a request options array.

```php
<?php
$opts = array(
    'headers' => array(
        'User-Agent' => 'Mozilla/5.0 (X11; U; ...',
        'Connection' => 'keep-alive'
    )
);
$request = new HttpRequest;
$request->setOptions($opts);
?>
```

The object-oriented API offers several slightly less cumbersome solutions compared to the procedural API as shown in the examples below.

```php
<?php
$request = new HttpRequest;
// configure $request
$request->send();

// only returns custom set request headers
print_r($request->getHeaders());

// returns all request headers
$requestmessage = $request->getRequestMessage();
print_r($requestmessage->getHeaders());

// returns a specific request header
echo $requestmessage->getHeader('User-Agent');

// both return all response headers
$responsemessage = $request->getResponseMessage();
print_r($responsemessage->getHeaders());
print_r($request->getResponseHeader());

// both return a single response header
echo $responsemessage->getHeader('Content-Length');
echo $request->getResponseHeader('Content-Length');
?>
```

Debugging

Debugging transmitted requests and responses with pecl_http is actually pretty simple. Configure and send a request, then call the `getRawRequestMessage` and `getRawResponseMessage` methods on the request instance as shown below.

```
<?php
$request = new HttpRequest('http://localhost.example/index.php');
// configure $request
$request->send();

// returns a string containing the entire request
echo $request->getRawRequestMessage();

// returns a string containing the entire response
echo $request->getRawResponseMessage();
?>
```

Timeouts

As with cURL, pecl_http has options for handling timeouts.

The `'timeout'` request option, which corresponds to `CURLOPT_TIMEOUT`, is the maximum number of seconds an entire request may take before timing out.

Likewise the `'connecttimeout'` request option, the counterpart for `CURLOPT_CONNECTTIMEOUT`, is the maximum number of seconds that a connection attempt, which includes DNS resolution, may take.

Finally the `'dns_cache_timeout'` request option, which is equivalent to `CURLOPT_DNS_CACHE_TIMEOUT`, is the maximum number of seconds that a DNS cache entry will be retained and defaults to 120 seconds (two minutes).

Content Encoding

If you are unfamiliar with content encoding, see "Content Encoding" in Chapter 8 for more information. To enable it when using pecl_http, the `'compress'` request option must be set to `true`.

Note that this requires libz support to be enabled. You can check for this by executing the `phpinfo` function within a PHP script or running `php -ri http` from command line. If libz support is not enabled, the technique detailed in Chapter 8 for handling encoded content can be used on the return value of the request object's `getResponseBody` method.

Cookies

```
<?php
$requestcookies = array(
    'foo' => 'foovalue',
    'bar' => 'barvalue'
);

$cookiejar = '/path/to/cookiejar';

// Procedural
$options = array(
    'cookie' => $requestcookies,
    // - OR -
    'cookiestore' => $cookiejar
);

$response = http_get('http://localhost.example', $options);
$parsedresponse = http_parse_message($response);
$responsecookies = array_map(
    'http_parse_cookie',
    $parsedresponse->headers['Set-Cookie']
);

// Object-oriented
$request = new HttpRequest('http://localhost.example');
$request->enableCookies();
$request->setCookies($cookies);
$request->addCookies(array('baz' => 'bazvalue'));
$request->send();
$responsecookies = $request->getResponseCookies();
?>
```

- Like cURL, pecl_http allows cookie data to be specified manually. Unlike cURL, pecl_http handles most of the formatting for you. Simply specify an associative array of cookie name-value pairs for the 'cookie' request option. If your cookie values are already encoded, set the 'encodecookies' request option to false.

- Also like cURL, pecl_http includes an option to use a file for storing cookie data. Unlike cURL, pecl_http always uses the same data source for both read and write operations. That is, it consolidates the CURLOPT_COOKIEFILE and CURLOPT_COOKIEJAR options into the 'cookiestore' request option.

- Because the procedural API lacks the persistent scope that is a defining characteristic of the object-oriented API, extracting cookie values for uses beyond storage and persistence is somewhat involved. http_parse_message is used to parse the headers and body from a string containing an HTTP response message into an object for easier access. http_parse_cookie is then applied to Set-Cookie header values to parse the cookie data from them.

- In HttpRequest the enableCookies method explicitly sets CURLOPT_COOKIEFILE to an empty string so that cookie data is persisted in memory. setCookies accepts an associative array of cookie name-value pairs just like the 'cookie' request option. addCookies does the same thing, but merges the array contents into any existing cookie data rather than deleting the latter as setCookies does.

- Once the send method is called on $request, cookie data from the response is retrieved by calling the getResponseCookies method.

HTTP Authentication

The 'httpauth' request option is used to set credentials in the format 'username:password'. The type of HTTP authentication to use is specified via the 'httpauthtype' request option using one of the pecl_http HTTP_AUTH_* constants, which are similar to those intended for the same purpose in the cURL extension.

Lastly, the 'unrestrictedauth' request option can be set to true if authentication credentials should be included in requests resulting from redirections pointing to a different host from the current one.

Redirection and Referers

Intuitively, the 'redirect' request option is used to set the maximum number of redirections to process before automatically terminating. Not so intuitively, it defaults to 0, which means that processing of redirections is disabled and you must explicitly set this request option to a value greater than 0 to enable it.

The 'referer' request option can be used to set the value of the Referer request header. Alternatively, it can be set via the 'headers' request option (more on that shortly) when using the procedural API or the setHeaders and addHeaders methods of the HttpRequest class when using the object-oriented API.

Content Caching

The 'lastmodified' request option accepts a UNIX timestamp to be used as the value for the If-Modified-Since or If-Unmodified-Since request header.

Likewise, the 'etag' request option accepts a string to be used as the value for the If-Match or If-None-Match request header.

User Agents

The 'useragent' request option can be used to set a custom user agent string for the request.

Byte Ranges

Like cURL, pecl_http includes request options for a single resume point and multiple byte ranges. The former is 'resume' and accepts an integer for the starting point. The latter is 'range' and is formatted as an array of enumerated arrays each containing a pair of integers representing a single byte range. What follows is an example of setting byte ranges within a request options array.

```
<?php
$opts = array(
    'range' => array(
        array(0, 1023), // first 1024 bytes (KB)
        array(3072, 4095) // fourth 1024 bytes (KB)
    )
);
?>
```

Alternatively, it can be set using custom headers via the `'headers'` request option.

```
<?php
$opts = array(
    'headers' => array(
        'User-Agent' => 'Mozilla/5.0 (X11; U; ...',
        'Connection' => 'keep-alive'
    )
);
?>
```

Request Pooling

pecl_http also inherits cURL's support for request pooling, or sending and processing multiple requests in parallel. Like other features, pecl_http implements it in a more succinct fashion. Oddly enough, it is one feature that is limited to pecl_http's object-oriented API and has no equivalent in its procedural API. It is implemented in the form of the `HttpRequestPool` class.

```
<?php
$request1 = new HttpRequest;
// configure $request1

$request2 = new HttpRequest;
// configure $request2

$request3 = new HttpRequest;
// configure $request3

$pool = new HttpRequestPool($request1, $request2);
$pool->attach($request3);
$pool->send();
?>
```

- The `HttpRequestPool` constructor accepts a variable number of arguments, all of which should be preconfigured `HttpRequest` instances.
- Request instances can also be added via the `attach` method of the `HttpRequestPool` class.
- `send` is called on the pool instance to transmit all requests and blocks until all responses are received. Once complete, all request instances contain their respective sets of response data as if they had been sent individually.

Wrap-Up

For more information on the pecl_http extension, see `http://php.net/http`.

For more information on its request options, `http://php.net/manual/en/http.request.options.php`.

Chapter 6

PEAR::HTTP_Client

The PHP Extension and Application Repository (PEAR) project houses a library of reusable components written in PHP and a package system for distributing them, one of which is the `HTTP_Client` package that will be covered in this section. PEAR spurred the creation of the PECL project, a repository of C extensions from which the pecl_http extension discussed in the previous section originates.

One trait of many components in PEAR is backward compatibility with older versions of PHP. The `HTTP_Client` package itself only requires 4.3.0, while the latest and last version of the PHP 4 branch is 4.4.9. As such, this client may be a good solution when feature requirements are more advanced than what the HTTP streams wrapper natively supports and the intended runtime environment is restricted to an older PHP version. Otherwise, if you still want an HTTP client written in PHP, have a look at Zend_Http_Client as described in the next section. It should go without saying that support for PHP 4.x has ceased and that it is highly advisable to run a more current stable version of PHP in production environments.

PHP source tarballs have a configure option to include PEAR when compiling. Some operating systems may have a package for PEAR apart from PHP itself. Once PEAR is installed, `pear install HTTP_Client` should be sufficient to install `PEAR::HTTP_Client`. There is also the option to manually download tarballs for the necessary PEAR packages (`HTTP_Client`, `HTTP_Request`, `Net_URL`, and `Net_Socket`), though this is a bit more difficult to maintain in the long term as updates are released.

> **PEAR Developments**
> At the time of writing, efforts are being put into PEAR 2, the successor to the current incarnation of the PEAR project. This will include a newer package distribution system and ports or revamps of many existing packages to bring them up-to-date with features available in PHP 5. However, because there is currently little in the way of stable ported code for PEAR 2, coverage in this book will be limited to PEAR 1.

Requests and Responses

There are two ways to perform HTTP requests with the related PEAR components: `HTTP_Request` and `HTTP_Client`. The latter composes the former to add capabilities such as handling redirects and persisting headers, cookie data, and request parameters across multiple requests. Here's how to perform a simple request and extract response information when using `HTTP_Request` directly.

```
<?php
require_once 'HTTP/Request.php';

$request =& HTTP_Request('http://localhost.example');
$request->setMethod(HTTP_REQUEST_METHOD_GET);
$request->setURL('http://localhost.example');
$response = $request->sendRequest();
?>
```

- The `HTTP_Request` constructor has two parameters, both of which are optional. The first is a string containing the URL for the request; note that the `setURL` method is an alternative way to specify a value for this. The second is a parameters array, which will be discussed later.

- By default, the GET method is used. `setMethod` is used to change this using constants, the names for which are formed by prefixing the desired request method with `HTTP_REQUEST_METHOD_` as in `HTTP_REQUEST_METHOD_GET`.

- sendRequest intuitively dispatches the request and obtains the response. It returns either true to indicate that that request succeeded or an error object (of the PEAR_Error class by default) if an issue occurred.

Issues that cause errors include environmental requirements of the component not being met or the target of the request exceeding the redirect limit. PEAR::isError is used to determine if the sendRequest call resulted in an error. If the request is sent successfully, several methods of the request object are available to extract response information.

```
<?php
if (!PEAR::isError($response)) {
    $code = $request->getResponseCode();
    $reason = $request->getResponseReason();
    $body = $request->getResponseBody();
    $cookies = $request->getResponseCookies();
    $headers = $request->getResponseHeader();
    $contentType = $request->getResponseHeader('Content-Type');
}
?>
```

Here are a few specifics about the response information methods.

- getResponseCode returns an integer containing the HTTP status code.

- getResponseReason returns a string containing a description that corresponds to the HTTP status code.

- getResponseCookies returns false if no cookies are set; otherwise it returns an enumerated array of associative arrays each of which contains information for an individual cookie such as its name, value, and expiration date.

- getResponseHeader will return an associative array of all headers indexed by header name (in all lowercase) if no parameter is passed to it; otherwise, it takes a single string parameter containing a specific header name for which it will return the value.

sendRequest does not modify any parameters set by you that are specific to the request (as opposed to the response). That is, request instances can be reused and individual parameters modified only as needed to change what differs between consecutive requests. To “start fresh” with an existing request instance, simply explicitly call its constructor method.

```
<?php
$request =& new HTTP_Request;
// ...
$request->HTTP_Request();
// all parameters previously set will be cleared
?>
```

Juggling Data

`HTTP_Request` makes data management pretty easy and straightforward, so it’s all covered in this section.

```
<?php
$request =& new HTTP_Request('http://localhost.example');

// GET
$request->addQueryString('variable', 'value');
$request->addRawQueryString('foo=bar');

// POST
$request->addPostData('variable', 'value');
$request->clearPostData();

// COOKIE
$request->addCookie('variable', 'value');
$request->clearCookies();

// FILES
$request->addFile('fieldname', '/path/to/file', 'text/plain');
?>
```

- `addRawQueryString` isn’t named very intuitively, as it actually overwrites any query string that you’ve previously set.

- By default, addQueryString and addPostData will URL-encode the variable value. To prevent this if your data is already pre-encoded, pass the value true as the third parameter to either method.

- clearPostData and clearCookies reset internal variables used to store data passed in by addQueryString and addPostData respectively.

- setURL and addQueryString or addRawQueryString can be used together to control the content of the internal variable for the URL of the request. getURL can be used to see the effects of these during development.

- addFile is intended to simulate uploading a file via a file input field in an HTML form. For forms with multiple file upload files with the same name, pass an array of file paths as the second parameter and (optionally) an array of corresponding file MIME types as the third parameter.

- To send the contents of a file as the entire body of a request, use file_get_contents to retrieve the file contents into a string and pass that to the setBody method of the request instance.

Wrangling Headers

Handling headers is also relatively straightforward. The only header-specific convenience method that's included is for handling Basic HTTP authentication, as shown below.

```
<?php
$request =& new HTTP_Request;
$request->addHeader('name', 'value');
$request->removeHeader('name');
$request->setBasicAuth('username', 'password');
?>
```

Using the Client

HTTP_Client persists explicit sets of headers and requests parameters across multiple requests, which are set using the setDefaultHeader and setRequestParameter methods respectively. The client constructor also accepts arrays of these. Default headers and request parameters can be cleared by calling reset on the client instance.

Internally, the client class actually creates a new instance of HTTP_Request per request. The request operation is set depending on which of the client instance methods are called; get, head, and post are supported.

The capabilities of the client described up to this point can all be accomplished by reusing the same request instance for multiple requests. However, the client also handles two things that the request class does not: cookies and redirects.

By default, cookies are persisted automatically across requests without any additional configuration. HTTP_Client_CookieManager is used internally for this. For custom cookie handling, this class can be extended and an instance of it passed as the third parameter to the client constructor. If this is done, that instance will be used rather than an instance of the native cookie manager class being created by default.

The maximum number of redirects to process can be set using the setMaxRedirects method of the client class. Internally, requests will be created and sent as needed to process the redirect until a non-redirecting response is received or the maximum redirect limit is reached. In the former case, the client method being called will return an integer containing the response code rather than true as the request class does. In the latter case, the client method will return an error instance. Note that the client class will process redirects contained in meta tags of HTML documents in addition to those performed at the HTTP level.

To retrieve information for the last response received, use the currentResponse method of the client instance. It will return an associative array containing the keys 'code', 'headers', and 'body' with values corresponding to the return values of request methods getResponseCode, getResponseHeader, and getResponseBody respectively.

By default, all responses are stored and can be accessed individually as shown below. To disable storage of all responses except the last one, call enableHistory on the client instance and pass it false.

```
<?php
for ($client->rewind(); $client->valid(); $client->next()) {
    $url = $client->key();
    $response = $client->current();
}
?>
```

Observing Requests

Both HTTP_Request and HTTP_Client have attach and detach methods for adding and removing instances of HTTP_Request_Listener. This class implements the observer design pattern and serves to intercept events that occur during the process of transmitting a request and receiving a response. To create an observer, first create a class that extends HTTP_Request_Listener as shown below.

```
<?php
class Custom_Request_Listener extends HTTP_Request_Listener
{
    function Custom_Request_Listener()
    {
        $this->HTTP_Request_Listener();
    }

    function update(&$subject, $event, $data = null)
    {
        switch ($event) {
            // Request events
            case 'connect':      // handle the 'connect' event
            case 'sentRequest':  // ...
            case 'disconnect':   // ...

            // Response events
            case 'gotHeaders':   // ...
            case 'tick':         // ...
            case 'gztick':       // ...
            case 'gotBody':      // ...

            // Client events
            case 'request':      // ...
            case 'httpSuccess':  // ...
            case 'httpRedirect': // ...
            case 'httpError':    // ...
```

```
                default:
                    PEAR::raiseError('Unhandled error: ' . $event);
            }
        }
    }
    ?>
```

- Declare a constructor that calls the parent constructor and performs any needed custom logic.

- Declare the `update` method with the signature shown. `$subject` is the request or client instance to which the listener is attached. `$event` is a string containing one of the events shown in the `switch` statement.

- `$data` is data specific to events related to reception of the server response.

- Request and response events occur within `HTTP_Request` instances. Client events occur within `HTTP_Client` instances.

- Note that not all events need to be handled, only those with which you are concerned.

The `attach` method in `HTTP_Request` takes a single parameter, a listener instance. The equivalent method in `HTTP_Client` takes two parameters, a required listener instance and an optional boolean flag. When the latter is `false` (which it is by default), attached listeners will not be propagated to requests created within the client instance. That is, a listener added will not be notified of request and response events, only client events. To have an added listener receive all events, explicitly specify the `$propagate` parameter to be `true` when calling `attach`.

Wrap-Up

For more information on the PEAR packages covered here, see these resources listed below.

- `http://pear.php.net/manual/en/package.http.http-request.php`

- `http://pear.php.net/manual/en/package.http.http-client.php`
- `http://pear.php.net/manual/en/package.http.http-request.listeners.php`

Chapter 7

Zend_Http_Client

Zend Framework (often abbreviated ZF) was conceived in early 2005. Contrary to PEAR1, ZF tends to remain relatively current in the version of PHP it requires and often takes advantage of features available in newer versions. As of ZF 1.10.1, PHP 5.2.4 is required.

Designed to take full advantage of the new object system introduced in PHP 5, ZF has a fully object-oriented API. The project team strives to maintain an E_STRICT level of compliance and good code coverage via unit testing.

Among the components included in ZF is `Zend_Http_Client`, an HTTP client library with a fairly comparable feature set to the others already covered in previous chapters. With that, let's move into how it works.

Basic Requests

Let's start with a basic request.

```
<?php
$client = new Zend_Http_Client;
$client->setUri('http://localhost.example');
$response = $client->request();

// This does the same thing
$client = new Zend_Http_Client('http://localhost.example');
$response = $client->request('GET');
```

```
// Another way to set the request method
$client->setMethod(Zend_Http_Client::GET);
?>
```

- The URL for the request can either be passed as the first parameter to the `Zend_Http_Client` constructor or via the `setUri` method after the client instance is created. Both methods accept the URI as either a string or an instance of `Zend_Uri_Http`.
- The `request` method of the client is used to dispatch the request, taking an optional parameter of a string containing the HTTP request method to use. GET is used by default. The HTTP request method can also be set via the client `setMethod` method using the client class constant named for that HTTP request method.

Responses

The response returned by the client's `request` method has a number of useful accessor methods.

```
<?php
// Returns an integer containing the status code
var_dump($response->getStatus());

// Returns a string containing a description for the status code
var_dump($response->getMessage());

// Returns a string containing the fully decoded response body
var_dump($response->getBody());

// Returns a string containing the unaltered response body
var_dump($response->getRawBody());

// Returns an associative array of headers
var_dump($response->getHeaders());

// Returns a string or array of values for a single header
var_dump($response->getHeader('Content-Type'));
```

```
// Returns TRUE for 100- and 200-level status codes
var_dump($response->isSuccessul());

// Returns TRUE for 400- and 500-level status codes
var_dump($response->isError());
?>
```

URL Handling

By default, `Zend_Http_Client` uses `Zend_Uri_Http` to validate any URI that is passed into the client. Use of unconventional URLs, particularly those using characters described as disallowed by section 2.4.3 of RFC 2396 (the predecessor to RFC 3986), may cause validation failure. See the code sample below for how to deal with this situation.

```
<?php
// $valid will be false because the URL contains a |
$valid = Zend_Uri::check('http://localhost.example/?q=this|that');

// Force URLs with disallowed characters to be considered valid
Zend_Uri::setConfig(array('allow_unwise' => true));

// $valid will be true because of allow_unwise being enabled
$valid = Zend_Uri::check('http://localhost.example/?q=this|that');

// URLs with disallowed characters will be considered invalid again
Zend_Uri::setConfig(array('allow_unwise' => false));
?>
```

Custom Headers

The `setHeaders` method is the Swiss Army Knife of header management for `Zend_Http_Client`. See the example below for the multitude of ways in which headers can be set.

```
<?php
// Single header, with header name and value separate
$client->setHeaders('Host', 'localhost.example');

// Single header, with header name and value together
$client->setHeaders('Host: localhost.example');

// Single header with multiple values, mainly useful for cookies
$client->setHeaders('Cookie', array(
    'lang=en-US',
    'PHPSESSID=1a0b82148815944c548caef5ccb884c9'
));

// Multiple headers, with header names and values separate
$client->setHeaders(array(
    'Host' => 'localhost.example',
    'User-Agent' => 'Zend_Http_Client 1.7.2'
));

// Multiple headers, with header names and values together
$client->setHeaders(array(
    'Host: localhost.example',
    'User-Agent: Zend_Http_Client 1.7.2'
));
?>
```

Configuration

`Zend_Http_Client` has configuration settings much like the context options of the HTTP streams wrapper, configuration settings of the cURL extension, and request options of the pecl_http extension. As shown in the example below, settings are declared in the form of an associative array of setting name-value pairs and can be passed either as the second parameter to the `Zend_Http_Client` constructor or later via its `setConfig` method.

```
<?php
$config = array('timeout' => 30);

// One way
$client = new Zend_Http_Client('http://localhost.example', $config);
```

```
// Another way
$client->setConfig($config);
?>
```

Connectivity

The timeout configuration setting is an integer value specifying the number of seconds for which the client should attempt to connect to the server before timing out. In the event of a timeout, an instance of the exception Zend_Http_Client_Adapter_Exception will be thrown.

By default, the client assumes that only one request will be performed on the connection it establishes. That is, it will automatically include a Connection header with a value of close. When sending multiple requests to the same host, the keepalive configuration can be set to true to have all requests sent on the same connection for improved performance.

Debugging

The last request sent by the client can be obtained in the form of a string via its getLastRequest method. For the last response received, the corresponding method getLastResponse can be called. This returns an instance of Zend_Http_Response rather than a string. To convert this object to a string, call its asString method. See below for examples of both.

```
<?php
$requestString = $client->getLastRequest();
$responseObject = $client->getLastResponse();
$responseString = $responseObject->asString();
?>
```

Note that the storeresponse configuration setting affects how getLastResponse behaves. When set to true (the default), it causes the last response received by the client to be stored for later retrieval. When it is set to false, the response is not stored and is only available as the return value of the client's request method. In this case, getLastResponse would return null. If you don't needed the additional availability of the response, turning this off can lessen resource usage.

Cookies

Zend_Http_Client will accept manually specified cookie name-value pairs via its setCookie method, but by default will not automatically retain response cookies and resend them in subsequent requests. To have it do so, simply call setCookieJar with no parameters. This will cause an instance of the default cookie handling class, Zend_Http_CookieJar, to be implicitly instantiated.

If you need access to cookie data for something other than propagating it to subsequent requests, there are a few ways to do so. Cookies can be accessed individually via the cookie jar's getCookie method, the required parameters for which are a URI and a cookie name.

```
<?php
$cookie = $client->getCookieJar()->getCookie(
    'http://localhost.example/',
    'cookiename'
);
?>
```

Note that the URI includes a scheme (http://), a domain (localhost.example), and a path (/). A single cookie jar instance can store cookie data for multiple domains and multiple paths on the same domain. In cases where the latter capability is not used, the path / can be specified so that all cookies set on the specified domain are available to all paths under that domain. The getMatchingCookies method of Zend_Http_CookieJar allows cookies to be accessed collectively based on these criteria and returns an array of Zend_Http_Cookie objects by default. See below for examples.

```
<?php
// All cookies for the domain localhost.example
$cookies = $cookiejar->getMatchingCookies(
    'http://localhost.example/'
);
```

```
// All cookies for the domain localhost.example with a path or
// subpath of /some/path
$cookies = $cookiejar->getMatchingCookies(
    'http://localhost.example/some/path'
);

// All non-session cookies for the domain localhost.example
$cookies = $cookiejar->getMatchingCookies(
    'http://localhost.example/',
    false
);
?>
```

Alternatively, `getAllCookies` can be used to access all cookies contained in the cookie jar instance. When a cookie jar is only used to store cookies for a single domain, `getAllCookies` offers a more concise method than `getMatchingCookies` to retrieve all cookies for that domain. Like `getMatchingCookies`, `getAllCookies` also returns an array of `Zend_Http_Cookie` objects by default.

Redirection

The `maxdirects` configuration setting is an integer indicating the maximum number of redirections to perform before terminating. Upon termination, the client will simply return the last response it received. The `isRedirect` method of `Zend_Http_Response` returns `true` for responses with a 300-level status code.

Sections 10.3.2 and 10.3.3 of RFC 2616 indicate that when a redirection occurs, both the request method and parameters should be retained. In practice, most clients don't behave this way. Instead, parameters are cleared and the method reverts to GET. For consistency with other clients, `Zend_Http_Client` behaves this way by default. To force it to be compliant with the RFC, the `strictredirects` configuration setting can be set to `true`.

User Agents

The `useragent` configuration setting contains the user agent string to use and defaults to `'Zend_Http_Client'`.

HTTP Authentication

As of writing, `Zend_Http_Client` only supports Basic HTTP authentication. According to the Zend Framework Reference Guide, support for Digest is planned. To set HTTP authentication credentials, call the client's `setAuth` method and pass in the username and password.

Wrap-Up

For the reference guide section on Zend_Http_Client, see `http://framework.zend.com/manual/en/zend.http.html`.

For API documentation on all classes in the Zend_Http package, see `http://framework.zend.com/apidoc/core/classtrees_Zend_Http.html`.

Chapter 8

Rolling Your Own

First, it should go without saying that it's generally better to use and build upon an existing library rather than trying to roll your own from scratch. For one thing, you can get a number of features "for free" that way with no work required on your part. For another, developers outside of your team and projects work on those libraries, and in the words of Eric S. Raymond, "Given enough eyes, all bugs are shallow."

However, it certainly doesn't hurt to be familiar with this information even if you don't plan to build your own client. Doing so gives you more capability to troubleshoot issues with existing libraries and contribute patches back to their project teams.

Whatever your motivation, here we go.

Sending Requests

In addition to wrappers for specific protocols, the streams extension also offers socket transports for dealing with data at a lower level. One of these socket transports is for TCP, or Transmission Control Protocol, which is a core internet protocol used to ensure reliable delivery of an ordered sequence of bytes. The socket transport facilitates sending a raw data stream, in this case a manually constructed HTTP request, to a server.

```
<?php
```

```
$stream = stream_socket_client('tcp://localhost.example:80');
$request = "GET / HTTP/1.1\r\nHost: localhost.example\r\n\r\n";
fwrite($stream, $request);
echo stream_get_contents($stream);
fclose($stream);

/*
Example output:
HTTP/1.1 200 OK
Date: Wed, 21 Jan 2009 03:16:43 GMT
Server: Apache/2.2.9 (Ubuntu) PHP/5.2.6-2ubuntu4 with Suhosin-Patch
X-Powered-By: PHP/5.2.6-2ubuntu4
Vary: Accept-Encoding
Content-Length: 12
Connection: close
Content-Type: text/html

Hello world!
*/
?>
```

- The `stream_socket_client` function is used to establish a connection with the server, returning a connection handle resource assigned to `$stream`.

- `tcp://` specifies the transport to use.

- `localhost.example` is the hostname of the server.

- `:80` specifies the port on which to connect to the server, in this case `80` because it is the standard port for HTTP. The port to use depends on the configuration of the web server.

- `$request` contains the request to be sent to the server, where individual lines are separated with a CRLF sequence (see Chapter 2 "GET Requests") and the request ends with a double CRLF sequence (effectively a blank line) to indicate to the server that the end of the request has been reached. Note that the request *must* contain the ending sequence or the web server will simply hang waiting for the rest of the request.

- The `fwrite` function is used to transmit the request over the established connection represented by `$stream`.

- The stream_get_contents function is used to read all available data from the connection, in this case the response to the request.
- The fclose function is used to explicitly terminate the connection.

Depending on the nature and requirements of the project, not all facets of a request may be known at one time. In this situation, it is desirable to encapsulate request metadata in a data structure such as an associative array or an object. From this, a central unit of logic can be used to read that metadata and construct a request in the form of a string based on it.

Manually constructing requests within a string as shown in the example above also doesn't have ideal readability. If exact requests are known ahead of time and do not vary, an alternative approach is storing them in a data source of some type, then retrieving them at runtime and sending them over the connection as they are. Whether it is possible to take this approach depends on the level of variance in requests going between the web scraping application and the target application.

If the need arises to manually build query strings or URL-encoded POST request bodies, the http_build_query function allows this to be done using associative arrays.

Parsing Responses

Once you've received a response, the next step is obtaining the data you need from it. Taking the response from the last example, let's examine what this might look like.

```php
<?php
// Split the headers and body into separate variables
list($headers, $body) = explode("\r\n\r\n", $response, 2);

// Remove the status line from the headers
list($status, $headers) = explode("\r\n", $headers, 2);

// Parse the headers segment into individual headers
preg_match_all(
    "/(?P<name>[^:]+): (?P<value>[^\r]+)(?:$|\r\n[^ \t]*)/U",
    $headers,
    $headers,
    PREG_SET_ORDER
);
?>
```

Logic to separate individual headers must account for the ability of header values to span multiple lines as per RFC 2616 Section 2.2. As such, preg_match_all is used here to separate individual headers. See the later chapter on PCRE for more information on regular expressions. If a situation necessitates parsing data contained in URLs and query strings, check out the parse_url and parse_str functions. As with the request, it is generally desirable to parse response data into a data structure for ease of reference.

Transfer Encoding

Before parsing the body, the headers should be checked for a few things. If a Transfer-Encoding header is present and has a value of chunked, it means that the server is sending the response back in chunks rather than all at once. The advantage to this is that the server does not have to wait until the entire response is composed before starting to return it (in order to determine and include its length in the Content-Length header), which can increase overall server throughput.

When each chunk is sent, it is preceded by a hexadecimal number to indicate the size of the chunk followed by a CRLF sequence. The end of each chunk is also denoted by a CRLF sequence. The end of the body is denoted with a chunk size of 0, which is particularly important when using a persistent connection since the client must know where one response ends and the next begins.

The strstr function can be used to obtain characters in a string prior to a newline. To convert strings containing hexadecimal numbers to their decimal equivalents, see the hexdec function. An example of what these two might look like in action is included below. The example assumes that a request body has been written to a string.

```php
<?php
$unchunked = ’’;
do {
    if ($length = hexdec(strstr($body, "\r\n", true))) {
        $body = ltrim(strstr($body, "\r\n"));
        $unchunked .= substr($body, 0, $length);
        $body = substr($body, $length + 2);
    }
} while ($length > 0);
```

```
?>
```

See Section 3.6.1 and Appendix 19.4.6 of RFC 2616 for more information on chunked transfer encoding.

Content Encoding

If the zlib extension is loaded (which can be checked using the `extension_loaded` function or executing `php -m` from command line), the client can optionally include an `Accept-Encoding` header with a value of `gzip,deflate` in its request. If the server supports content compression, it will include a `Content-Encoding` header in its response with a value indicating which of the two compression schemes it used on the response body before sending it.

The purpose of this is to reduce the amount of data being sent to reduce bandwidth consumption and increase throughput (assuming that compression and decompression takes less time than data transfer, which is generally the case). Upon receiving the response, the client must decompress the response using the original scheme used by the server to compress it.

```
<?php
// If Content-Encoding is gzip...
$decoded = gzinflate(substr($body, 10));

// If Content-Encoding is deflate...
$decoded = gzuncompress($body);
?>
```

- Yes, the function names are correct. One would think that `gzinflate` would be used to decode a body encoded using the `deflate` encoding scheme. Apparently this is just an oddity in the naming scheme used by the zlib library.

- When the encoding scheme is `gzip`, a GZIP header is included in the response. `gzinflate` does not respond well to this. Hence, the header (contained in the first 10 bytes of the body) is stripped before the body is passed to `gzinflate`.

See RFC 2616 Section 3.5 for more information on content encoding. RFC 1951 covers specifics of the DEFLATE algorithm on which the `deflate` encoding scheme is based while RFC 1952 details the gzip file format on which the `gzip` encoding scheme is based.

Timing

Each time a web server receives a request, a separate line of execution in the form of a process or thread is created or reused to deal with that request. Left unchecked, this could potentially cause all resources on a server to be consumed by a large request load. As such, web servers generally restrict the number of requests they can handle concurrently. A request beyond this limit would be blocked until the server completed an existing request for which it had already allocated resources. Requests left unserved too long eventually time out.

Throttling is a term used to describe a client overloading a server with requests to the point where it consumes the available resource pool and thereby delays or prevents the processing of requests, potentially including requests the client itself sent last. Obviously, it's desirable to avoid this behavior for two reasons: 1) it can be construed as abuse and result in your IP being banned from accessing the server; 2) it prevents the client from being consistently functional.

Most web browsers will establish a maximum of four concurrent connections per domain name when loading a given resource and its dependencies. As such, this is a good starting point for testing the load of the server hosting the target application. When possible, measure the response times of individual requests and compare that to the number of concurrent requests being sent to determine how much of a load the server can withstand.

Depending on the application, real-time interaction with the target application may not be necessary. If interaction with the target application and the data it handles will be limited to the userbase of your web scraping application, it may be possible to retrieve data as necessary, cache it locally, store modifications locally, and push them to the target application in bulk during hours of non-peak usage. To discern what these hours are, observe response time with respect to the time of day in which requests are made to locate the time periods during which response times are consistently highest.

Chapter 9

Tidy Extension

At this point, you should have completed the retrieval phase of the web scraping process and have your raw response body ready and awaiting analysis. Congratulations on making it to the halfway mark.

While other response formats are certainly possible, chances are good that you are dealing with a markup-based language like HTML or XHTML. As such, this chapter and subsequent chapters will deal specifically with those formats. More often than not, HTML will be the language in use.

There are a number of available XML-focused PHP extensions that can deal with markup, which will be reviewed in chapters to follow. While these extensions do provide some support for HTML and are even somewhat forgiving about malformed markup, well-formed XHTML is their ideal input. This being the case, the first step in the analysis process is to perform any necessary cleanup on the response body in order to minimize the number of issues that can be encountered later during analysis.

Validation

The World Wide Web Consortium (W3C) provides a markup validation service in order to promote adherence to web standards. You can access this service by going to `http://validator.w3.org`. It accepts markup to be tested by URL, file upload, or direct input. This will give you an indication of what issues, if any, your document has with markup malformation.

Tidy

There are two ways to proceed in cleaning up markup malformations. One is manual, involves the use of basic string manipulation functions or regular expression functions, and can quickly become messy and rather unmanageable. The other is more automated and involves using the tidy extension to locate and correct markup issues. While the process is configurable, it obviously lacks the fine-grained control that comes with handling it manually.

The majority of this chapter will focus on using tidy to correct markup issues. For those issues that tidy cannot handle to your satisfaction, the approach mentioned earlier involving string and regular expression functions is your alternative. Regular expressions will be covered in more detail in a later chapter.

The tidy extension offers two API styles: procedural and object-oriented. Both offer mostly equivalent functionality (relevant differences will be covered later) and which to use is really a matter of preference. Though both API styles use objects of the `tidy` class, it is recommended that only one style be used as much as is feasible for the sake of consistency in syntax. Code examples in this chapter will use both styles.

Input

Before correcting markup issues in a set of markup data, the data has to be parsed into a `tidy` object. More often than not markup data will already be contained within a string when it is ready to be parsed, but may also be stored in an external file. See below for example of how to handle either of these cases.

```
<?php
// Procedural
$tidy = tidy_parse_string($string, $config);
$tidy = tidy_parse_file($filename, $config);

// Object-oriented
```

```
$tidy = new tidy;
$tidy->parseString($string, $config);
$tidy->parseFile($filename, $config);
?>
```

Configuration

Like the cURL extension, the tidy extension operates largely on the concept of configuration; hence, `$config` parameters are present in all calls in the above example. Unlike most other extensions, this parameter can actually be one of two things: an associative array of setting-value pairs or the path to an external configuration file.

The configuration file format is somewhat similar to individual style settings in a CSS stylesheet. An example is shown below. It's unlikely that a non-developer will need to access the configuration settings and not the PHP source code using tidy as well. As such, separation into an external configuration file is really only useful for the sake of not cluttering source code with settings. Additionally, because the configuration file is read from disk, it may pose performance concerns when in high use.

```
// single-line comment
/* multi-line comment */
indent: false /* setting: value */
wrap: 78
```

When using the object-oriented API, an alternative to using configuration files is subclassing the `tidy` class and overriding its `parseString` and `parseFile` methods to automatically include specific configuration setting values. This method allows for easy reuse of tidy configurations.

```
<?php
class mytidy extends tidy {
    private $_default = array(
        'indent' => false,
        'wrap' => 78
    );
    public function parseFile($filename, $config, $encoding,
        $use_include_path=false) {
        return parent::parseFile(
            $filename,
            array_merge($this->_default, $config),
            $encoding,
            $use_include_path
        );
    }

    public function parseString($filename, $config, $encoding,
        $use_include_path=false) {
        return parent::parseString(
            $filename,
            array_merge($this->_default, $config),
            $encoding,
            $use_include_path
        );
    }
}
?>
```

- `array_merge` is used to consolidate default parameter values in the `$_default` property into the specified `$config` array. Any parameters specified in the original `$config` array will take precedence over corresponding parameters specified in `$_default`.

- `parseFile` and `parseString` pass the modified `$config` parameter with all other provided parameters to their respective methods in the `tidy` class and return the resulting return value.

Options

Tidy includes a large number of configuration options, only a few of which are relevant in the context of this book.

Two options deal with output formats applicable for web scraping: `output-html` and `output-xhtml`. Both are specified as boolean values. These options are mutually exclusive, meaning that only one can be set to `true` at any given time. Generally `output-xhtml` is preferable, but may not always be feasible to use. It's important to compare tidy output to the original document to confirm that correction of document malformations hasn't resulted in data loss.

Document encoding is one area where issues may arise later depending on the configuration of tidy when it's used. For example, the XMLReader extension uses UTF-8 encoding internally, which may be problematic if your input document's encoding conflicts. `input-encoding` and `output-encoding` can be used to control the assumed encoding for each.

Other options are useful mainly for debugging purposes and should generally be turned off in production environments. This is a good reason for subclassing the `tidy` class to control default option values, so that two separate sets are easily accessible for different development environments.

Three of these options are `indent`, `indent-spaces`, and `indent-attributes`. The first of these, `indent`, is a boolean value indicating whether tidy should apply indentation to make the hierarchical relationships between elements more visually prominent. `indent-spaces` is an integer containing the number of whitespace characters used to denote a single level of indentation, defaulting to 2. Lastly, `indent-attributes` is a boolean value indicating whether each individual attribute within an element should begin on a new line.

Speaking of attributes, `sort-attributes` can be set to `alpha` in order to have element attributes be sorted alphabetically. It is set to `none` by default, which disables sorting.

If lines within a document tend to be long and difficult to read, the `wrap` option may be useful. It's an integer representing the number of characters per line that tidy should allow before forcing a new line. It is set to `68` by default and can be disabled entirely by being set to `0`.

Having no empty lines to separate blocks can also make markup difficult to read. `vertical-space` is a boolean value intended to help with this by adding empty lines for readability. It is disabled by default.

Debugging

As good a job as it does, tidy may not always be able to clean documents. When using tidy to repair a document, it's generally a good idea to check for what issues it encounters.

There are two types of issues to check for when using tidy for web scraping analysis: warnings and errors. Like their PHP counterparts, warnings are non-fatal and generally have some sort of automated response that tidy executes to handle them. Errors are not necessarily fatal, but do indicate that tidy may have no way to handle a particular issue.

All issues are stored in an error buffer regardless of their type. Accessing information in and about this buffer is one area in which the procedural and object-oriented APIs for the tidy extension differ.

```
<?php
// Procedural
$issues = tidy_get_error_buffer($tidy);

// Object-oriented
$issues = $tidy->errorBuffer;
?>
```

Note that `errorBuffer` is a property of the `$tidy` object, not a method. Also note the slight difference in naming conventions between the procedural function and the object property, versus the consistency held throughout most other areas of the APIs.

The error buffer contained within a string is in and of itself mostly useless. Below is a code sample derived from a user contributed comment on the PHP manual page for the `tidy_get_error_buffer` function. This parses individual components of each issue into arrays where they are more easily accessible.

```
<?php
preg_match_all(
    '/^(?:line (?P<line>\d+) column (?P<column>\d+) - )?' .
    '(?P<type>\S+): (?:\[(?:\d+\.?){4}]:)?(?P<message>.*)?$/m',
    $tidy->errorBuffer, // or tidy_get_error_buffer($tidy)
    $issues,
    PREG_SET_ORDER
);

print_r($issues);

/*
Example output:
```

```
Array
(
    [0] => Array
         (
             [0] => line 12 column 1 - Warning: <meta> element not
                empty or not closed
             [line] => 12
             [1] => 12
             [column] => 1
             [2] => 1
             [type] => Warning
             [3] => Warning
             [message] => <meta> element not empty or not closed
             [4] => <meta> element not empty or not closed
         )
)
*/
?>
```

The tidy extension also provides a way to get at the number of warnings and errors that are encountered without requiring that you manually parse the error buffer. Unfortunately and rather oddly, this is only supported in the procedural API. However, adding it to the object-oriented API by subclassing the `tidy` class is fairly simple. Examples of both are shown below.

```
<?php
// Procedural
$warnings = tidy_warning_count($tidy);
$errors = tidy_error_count($tidy);

// Object-oriented
class mytidy extends tidy {
    public function warningCount() {
        return tidy_warning_count($this);
    }
    public function errorCount() {
        return tidy_error_count($this);
    }
}
?>
```

Output

Obtaining the resulting output of tidy repairing a document is fairly simple.

```
<?php
// Procedural
$output = tidy_get_output($tidy);

// Object-oriented
$output = (string) $tidy;
?>
```

While the object-oriented API offers no public declaration of the magic method `__toString`, it can be cast to a string as well as output directly using the `echo` construct.

Wrap-Up

This concludes the chapter. At this point, you should have your obtained document in a format suitable for input to an XML extension. The following few chapters will be devoted to using specific extensions to searching and extracting data from repaired documents.

For the PHP manual section on the tidy extension, see `http://php.net/tidy`.

For documentation on the tidy library itself, see `http://tidy.sourceforge.net/#docs`.

For a tidy configuration setting reference, see `http://tidy.sourceforge.net/docs/quickref.html`.

Chapter 10

DOM Extension

Once the retrieved markup document has been cleaned up so that it validates, the next step is extracting useful data from it. For this, the ideal approach is to take advantage of the fact that the document is valid markup and apply an XML extension to it. PHP has several of these and the next few chapters will be devoted to covering them.

The namesake of this chapter is the Document Object Model (DOM) extension. This extension gets its name from a standardized language-independent API for navigating and manipulating valid and well-formed XML and HTML documents. The standard is maintained and recommended by the World Wide Web Consortium (W3C), an organization devoted to emerging internet-related standards. The chapter won't cover the DOM extension in its entirety, only parts that are relevant and essential to web scraping.

DOM XML

The DOM extension is only available in PHP 5. Its PHP 4-compatible predecessor, the DOM XML extension, has a somewhat different API but shares many of the same concepts. Examples in this chapter will be restricted to the DOM extension. The related section of the PHP manual can be consulted for equivalent specifics in the DOM XML extension at `http://php.net/manual/en/book.domxml.php`.

Types of Parsers

Before going much further, you should be aware that there are two types of XML parsers: **tree parsers** and **pull parsers**. Tree parsers load the entire document into memory and allow you to access any part of it at any time as well as manipulate it. Pull parsers read the document a piece at a time and limit you to working with the current piece being read.

The two types of parsers share a relationship similar to that between the `file_get_contents` and `fgets` functions: the former lets you work with the entire document at once and uses as much memory needed to store it, while the latter allows you to work with a piece of the document at a time and use less memory in the process.

When working with fairly large documents, lower memory usage is generally the preferable option. Attempting to load a huge document into memory all at once has the same effect on the local system as a throttling client does on a web server: in both cases, resources are consumed and system performance is debilitated until the system eventually locks up or crashes under the stress.

The DOM extension is a tree parser. In general, web scraping does not require the ability to access all parts of the document simultaneously. However, the type of data extraction involved in web scraping can be rather extensive to implement using a pull parser. The appropriateness of extension over the other depends on the size and complexity of the document.

Loading Documents

The `DOMDocument` class is where use of the DOM extension begins. The first thing to do is instantiate it and then feed it the validated markup data. Note that the DOM extension will emit warnings when a document is loaded if that document is not valid or well-formed. To avoid this, see the previous chapter on using the tidy extension. If tidy does not eliminate the issue, errors can be controlled as shown in the example below. Note that errors are buffered until manually cleared, so make a point of clearing them after each load operation if they are not needed to avoid wasting memory.

```
<?php
```

```
// Buffer DOM errors rather than emitting them as warnings
$oldSetting = libxml_use_internal_errors(true);

// Instantiate a container for the document
$doc = new DOMDocument;

// Load markup already contained within a string
$doc->loadHTML($htmlString);

// Load markup saved to an external file
$doc->loadHTMLFile($htmlFilePath);

// Get all errors if needed
$errors = libxml_get_errors();

// Get only the last error
$error = libxml_get_last_error();

// Clear any existing errors from previous operations
libxml_clear_errors();

// Revert error buffering to its previous setting
libxml_use_internal_errors($oldSetting);
?>
```

Tree Terminology

Once a document is loaded, the next natural step is to extract desired data from it. However, doing so requires a bit more knowledge about how the DOM is structured. Recall the earlier mention of tree parsers. If you have any computer science background, you will be glad to know that the term “tree” in the context of tree parsers does in fact refer to the data structure by the same name. If not, here is a brief rundown of related concepts.

A **tree** is a hierarchical structure (think family tree) composed of **nodes**, which exist in the DOM extension as the `DOMNode` class. Nodes are to trees what elements are to arrays: just items that exist within the data structure.

Each individual node can have zero or more **child nodes** that are collectively represented by a `childNodes` property in the `DOMNode` class. `childNodes` is an instance of the class `DOMNodeList`, which is exactly what it sounds like. Other related properties include `firstChild` and `lastChild`. **Leaf nodes** are nodes that have no children, which can be checked using the `hasChildNodes` method of `DOMNode`.

All nodes in a tree have a single **parent node**, with one exception: the **root node** from which all other nodes in the tree stem. If two nodes share the same parent, they are appropriately referred to as **sibling nodes**. This relationship is shown in the `previousSibling` and `nextSibling` properties in `DOMNode`.

Lastly, child nodes of a node, child nodes of those child nodes, and so on are collectively known as **descendant nodes**. Likewise, the parent node of a node, that parent node's parent node, and so on are collectively known as **ancestor nodes**.

An example may help to showcase this terminology.

```
<html>
<body>
<ul id="thelist">
  <li>Foo</li>
  <li>Bar</li>
</ul>
</body>
</html>
```

- `html` is the root node.
- `body` is the first (and only) child of `html`.
- `ul` is the first (and only) child of `body`.
- The `li` nodes containing `Foo` and `Bar` are the first and last child nodes of `ul` respectively.
- The `li` node containing `Bar` node is the next sibling of the `li` node containing `Foo`.
- The `li` node containing `Foo` is likewise the previous sibling of the `li` node containing `Bar`.
- The `ul` and `li` nodes are descendants of the `body` node.

Elements and Attributes

At this point, the DOM transcends the tree analogy. There are multiple types of nodes, or to phrase that within the context of the DOM extension, DOMNode has multiple subclasses. The main two you'll be dealing with are DOMElement for **elements** and DOMAttr for **attributes**. Here are how these concepts apply to the example in the last section.

- ul is the name of an element.
- id is the name of an attribute of the ul element.
- thelist is the value of the id attribute.
- Foo and Bar are the values of the li elements.

Locating Nodes

Two methods of the DOMDocument class allow you to reduce the number of nodes you have to traverse to find the data you want fairly quickly.

getElementById attempts to locate a single element that meets two criteria: 1) it is a descendant of the document's root element; 2) it has a given id attribute value. If such an element is found, it is returned as a DOMElement instance; if not, null is returned.

getElementsByTagName attempts to locate all elements that meet two criteria: 1) it is a descendant of the document's root element; 2) it has a given element name (such as ul). This method always returns a DOMNodeList of any found elements. The DOMNodeList class has a length property that will be equal to 0 if no elements are found. It is also iterable, so it can be used as the subject of a foreach loop.

The DOMElement class also has a getElementsByTagName method, which functions the same way with the exception that located elements will be descendants of that element instead of the document's root element.

```
<?php
// One way get the list items in the last example
$listItems = $doc->getElementsByTagName('li');
```

```
// A slightly more specific way (better if there are multiple lists)
if ($list = $doc->getElementById('thelist')) {
    $listItems = $list->getElementsByTagName('li');
}

// Yet another way if the list doesn't have an id
$lists = $doc->getElementsByTagName('ul');
if ($lists->length) {
    $list = $lists->item(0);
    $listItems = $list->getElementsByTagName('li');
}

// Outputs "thelist" (without quotes)
echo $list->getAttribute('id');

// Outputs "Foo" on one line, then "Bar" on another
foreach ($listItems as $listItem) {
    echo $listItem->nodeValue, PHP_EOL;
}

// Outputs text content inside <ul id="thelist"> and </ul>
echo $list->nodeValue;
?>
```

XPath and DOMXPath

Somewhat similar to the way that regular expressions allow instances of character patterns to be found within strings, XPath allows instances of node patterns to be found within XML-compatible documents. Both technologies accomplish their purpose by providing a syntax comprised of **meta-characters** to express these patterns in a concise and succinct way. With the DOM extension, support for version 1.0 of the XPath standard is implemented as the `DOMXPath` class.

The `DOMXPath` constructor has a single required parameter: an existing `DOMDocument` instance on which queries will be performed. `DOMXPath` has two other relevant methods: `evaluate` and `query`. Both accept a string containing an XPath expression with which to query the document as their first parameter.

Optionally, a `DOMNode` instance associated with the document may be passed in as the second parameter (`$contextNode`) for either method. When specified, that node will become the **context node** and query results will be restricted to that node and its descendants. Otherwise, the root element of the document is assumed to be the context node.

The difference between `evaluate` and `query` is that the latter will always return an instance of `DOMNodeList` whereas the former will attempt to return an appropriately typed result if only one result is found.

Absolute Addressing

The process of using an XPath expression to obtain a set of nodes to which that expression applies is referred to as **addressing**. The remainder of the chapter will cover various aspects of addressing and related expression syntax.

XPath expressions share several similarities with UNIX filesystem paths, both of which are used to traverse conceptual tree structures. See the example below for specific instances of this. The previous HTML example used to illustrate various concepts of markup languages is reused here to showcase XPath addressing.

```
<?php
// Load a markup document
$doc = new DOMDocument;
$doc->loadHTML('
    <html>
    <body>
    <ul id="thelist">
      <li>Foo</li>
      <li>Bar</li>
    </ul>
    </body>
    </html>
');

// Configure an object to query the document
$xpath = new DOMXPath($doc);

// Returns a DOMNodeList with only the html node
$list = $xpath->query('/html');

// Returns a DOMNodeList with only the body node
$list = $xpath->query('/html/body');
```

```
// Also returns a DOMNodeList with only the body node
$list = $xpath->query('//body');
?>
```

- In the first two examples, note that the root element (`html`) is referenced in the expression even though it is assumed to be the context node (since no other node is specified as the second parameter in either `query` call).
- A **single forward slash** `/` indicates a parent-child relationship. `/html/body` addresses all `body` nodes that are children the document's root `html` element (which in this case only amounts to a single result).
- A **double forward slash** `//` indicates an ancestor-descendant relationship. `//body` addresses all `body` nodes that are descendants of the context node (which again only amounts to a single result).

The single and double forward slash operators can be used multiple times and in combination with each other as shown below.

```
<?php
// Returns all ul nodes that are descendants of the body node
$list = $xpath->query('//body//ul');

// Returns all li nodes that are children of the ul nodes
$list = $xpath->query('//body//ul/li');
?>
```

Namespaces
If you attempt to address nodes by their element name and receive no results when it appears you should, it's possible that the document is namespacing nodes. The easiest way to get around this is to replace the element name with a condition.
For example, if you are using the expression *ul, an equivalent expression that disregards the namespace would be* `∗[name()=“ul”]` where ∗ is a wildcard for all nodes and the `name` function compares the node name against a given value.

Relative Addressing

The above examples use what is called **absolute addressing**, similar in concept to absolute filesystem paths. The next example covers use **relative addressing**.

```
<?php
// Returns all ul nodes that have li child nodes
$list = $xpath->query('//body//ul/li/..');
?>
```

Where `//body//ul/li` selects `li` nodes, `//body//ul/li/..` instead selects `ul` nodes by using relative addressing (specifically `..`) to indicate that the parent node of the addressed node (`li`) should be returned rather than the addressed node itself. Compare this with the same sequence used to refer to the parent directory of the current directory on a UNIX filesystem.

Addressing Attributes

Examples thus far have dealt with element nodes in particular. XPath also supports the selection of attribute nodes.

```
// Returns the id attribute nodes of all ul nodes
$list = $xpath->query('//ul/@id');

// Returns all ul nodes that have id attribute nodes
$list = $xpath->query('//ul/@id/..');

// Returns all id attribute nodes
$list = $xpath->query('//@id');
?>
```

- `//ul/@li` addresses all `id` attribute nodes associated with `ul` element nodes that are descendants of the context node.
- Note how the use of relative addressing in the second example applies to attributes nodes just as it would apply to child element nodes.

- `//@id` addresses all `id` attribute nodes that are descendants of the context node regardless of the element with which they are associated.

Unions

When addressing nodes, it is possible to perform a single query that uses multiple expressions. This is referred to as addressing the **union** of those expressions.

```
<?php
// Returns the li child nodes of all ul and ol nodes
$list = $xpath->query('//ol/li|//ul/li');

// Returns the th and td nodes of all tr nodes
$list = $xpath->query('//tr/th|//tr/td');
?>
```

- The first example returns all list items (`li`) of both unordered (`ul`) and ordered (`ol`)) lists.
- The second example returns all header (`th`) and data (`td`) cells of table rows (`tr`).

Conditions

In addition to their surrounding hierarchical structure, nodes can also be addressed based on **conditions**.

```
<?php
// Returns all ul nodes with an id attribute node
$list = $xpath->query('//ul[@id]');

// Returns all li child nodes of the ul node with an id of "thelist"
$list = $xpath->query('//ul[@id = "thelist"]/li');

// Returns the first ul node that is a descendant of the context node
$list = $xpath->query('//ul[1]');
```

```
// Returns the first li child node of each ul node
$list = $xpath->query('//ul/li[1]');

// Returns all ul nodes containing an li node with the value "foobar"
$list = $xpath->query('//ul[li = "foobar"]');
?>
```

- Square brackets are used to delimit a conditional expression.
- Element and attribute nodes are denoted the same way within a condition as they are outside of one. That is, elements are simply referred to by element name and attribute names are prefixed with `@`.
- The = operator is used for equality comparisons. The converse, the `!=` operator, checks for inequality. Other fairly standard comparison operators are also supported, including <, <=, >, and >=.
- A condition comprised only of a single number is actually short for `position() = #` where `#` is the number used. `position` is a function that returns the position of each individual node within the current context.

Resources

Only a fraction of what XPath offers has been covered here, mainly basic concepts and areas that are most likely to be applicable when using XPath to extract data from retrieved markup documents. Other functions and operators and more advanced concepts are detailed further in the resources cited at the end of the chapter. Review of those resources is highly recommended for more extensive and complex data extraction applications.

- DOM documentation in the PHP manual: `http://php.net/dom`
- An excellent overview of XML and XPath: `http://schlitt.info/opensource/blog/0704_xpath.html`
- More information on XML: `http://en.wikibooks.org/wiki/XML:_Managing_Data_Exchange`

- DOM Level 3 Core standard: `http://www.w3.org/TR/DOM-Level-3-Core`
- DOM Level 3 XPath standard: `http://www.w3.org/TR/DOM-Level-3-XPath`

Chapter 11

SimpleXML Extension

Like the DOM extension, the SimpleXML extension provides a tree parser. However, the DOM extension keeps an API fairly consistent with that of the standard on which it is based. The API offered by SimpleXML is one that most PHP developers will find to be less massive, more intuitive and easier to use compared to that of the DOM extension.

SimpleXML makes use of the flexibility and versatility of PHP arrays by using them to access both individual elements from a collection with the same element name as well as element attributes. It also adds iterability to markup documents, making them simple to traverse.

However, these advantages do not come without cost. Like other tree parsers, SimpleXML must parse the entire document before it can be used. As such, it tends to be slower when parsing larger documents than a pull parser like XMLReader (which will be covered later). For a script dealing with a smaller document, however, SimpleXML works well in a pinch.

Loading a Document

Where `DOMDocument` is the central class in the DOM extension, `SimpleXMLElement` is its counterpart in the SimpleXML extension. Loading a markup document is as simple as specifying it via the class constructor.

```
<?php
// Loads markup already contained within a string
$sxe = new SimpleXMLElement($markupString);

// Loads markup contained within an external file
$sxe = new SimpleXMLElement($filePath, null, true);
?>
```

The constructor's second parameter (specified as `null` in the second example shown above) allows for further configuration of the instance. For documents with less than valid markup, using the value `LIBXML_NOERROR | LIBXML_NOWARNING` for this parameter may be useful as it suppresses error and warning reports.

Finally, the third parameter used in the second constructor call is set to `true` to indicate that `$filePath` is in fact a path to a file containing markup rather than the markup itself.

Accessing Elements

Use of SimpleXML looks a bit like XPath expressions, except that the same effect is accomplished programmatically through the API rather than via formatted expressions stored in strings. Unlike XPath expressions, SimpleXML automatically assumes that element access is relative to the root node.

```
<?php
$markupString = '
    <html>
    <body>
    <ul id="thelist">
      <li>Foo</li>
      <li>Bar</li>
    </ul>
    </body>
    </html>
';

// Outputs "Foo" -- note that the "html" element isn't referenced
$sxe = new SimpleXMLElement($markupString);
echo $sxe->body->ul->li[0];

// Also works, assumes the first "li" element if several are present
```

```
echo $sxe->body->ul->li;

// Also outputs "Foo" -- note that "html" is referenced this time
$doc = new DOMDocument();
$doc->loadHTML($markupString);
$xpath = new DOMXPath($doc);
echo $xpath->evaluate('/html/body/ul/li[0]');

// Outputs "ul"
echo $sxe->body->ul->getName(), PHP_EOL;

// Outputs "Foo" then "Bar"
foreach ($sxe->body->ul->li as $li) {
    echo $li, PHP_EOL;
}

// Does the same thing
foreach ($sxe->body->ul->children() as $li) {
    echo $li, PHP_EOL;
}
?>
```

When referencing a child element of the current element (via the `body` property of `$sxe` above, for example), note that the child element being accessed is also an instance of `SimpleXMLElement`. This means it's not only possible to chain element access, but also to reference the last element in such a chain using a variable. See the first `foreach` loop shown above for an example of both of these.

Accessing Attributes

Where element access makes use of enumerated arrays for accessing multiple elements with the same name on the same hierarchical level, attribute access makes use of associative arrays. The example below uses the same `$markupString` sample data as in the previous example.

```
<?php
// Outputs "thelist"
echo $sxe->body->ul['id'];

// Outputs "id=thelist"
```

```
foreach ($sxe->body->ul->attributes() as $name => $value) {
    echo $name, '=', $value, PHP_EOL;
}

// Another way to output "thelist"
$attrs = $sxe->body->ul->attributes();
echo $attrs->id;
?>
```

What the `attributes()` method actually returns is a `SimpleXMLElement` instance that provides access to attribute names and values in the same way that `SimpleXMLElement` normally makes child elements and their values available. As such, the returned instance can be used as the subject of a `foreach` loop to iterate over the properties that it exposes.

```
<!-- Actual markup -->
<ul id="thelist"></ul>

<!-- How attributes() exposes it as a SimpleXMLElement instance -->
<ul>
  <id>thelist</id>
</ul>
```

A Debugging Bug

Bug #44973, which affects the SimpleXML extension, is reported to be present in PHP 5.2.6 and may be present in other versions as well. The bug is exposed when an attempt is made to output a node accessed via a SimpleXMLElement instance using `echo`, `print_r()`, `var_dump()`, and other similar functions. If the node has both attributes and a text value, the attributes will not be shown in the output of these functions. This does not mean that attribute values are not accessible in this situation; they simply can't be output in this fashion unless referenced directly. More information on this is available at `http://bugs.php.net/bug.php?id=44973`.

Comparing Nodes

To compare an element or attribute with a string value, you must first explicitly cast it to a string. Otherwise, it will be treated as an object, which may causes type issues including emitting errors.

```
<?php
if ((string) $sxe->body->ul['id'] == 'thelist') {
    echo htmlentities((string) $sxe->body->ul['id']);
}
?>
```

DOM Interoperability

With both the DOM and SimpleXML extensions being tree parsers, this allows for a certain degree of interoperability between the two. This can be handy if you prefer one over the other when refactoring legacy code or if you have use cases for both within a single application.

```
<?php
// Converts a SimpleXMLElement to a DOMElement
$domElement = dom_import_simplexml($simpleXmlElement);

// Converts a DOMNode to a SimpleXMLElement
$simpleXmlElement = simplexml_import_dom($domNode);
?>
```

XPath

Like the DOM extension, SimpleXML also supports XPath. Rather than using a separate class for it as the DOM does, the `SimpleXMLElement` class itself includes a method for it. Where the `query()` and `evaluate()` methods of the `DOMXPath` class return a `DOMNodeList` instance, the `xpath` method of `SimpleXMLElement` instead returns an enumerated array of `SimpleXMLElement` instances that match the given XPath expression.

```
<?php
// Returns all list items from the previous example
$elements = $sxe->xpath('//ul[@id="thelist"]/li');

// Outputs "Foo" then "Bar"
foreach ($elements as $li) {
    echo $li, PHP_EOL;
}
?>
```

Wrap-Up

This concludes the chapter. Hopefully it's given you a good idea of how the SimpleXML extension functions and the ease of use it provides. One thing to note is that this chapter hasn't really shown very many practical examples of its usage. This is best learned simply by experimentation with real world data. Eventually, you will get a feel for when SimpleXML can be used and when DOM should be the preferred solution.

For SimpleXML documentation in the PHP manual, see `http://php.net/manual/en/book.simplexml.php`.

Chapter 12

XMLReader Extension

The previous two chapters have covered two available XML extensions that implement tree parsers. This chapter will focus on the XMLReader extension, which implements a pull parser.

As mentioned in the chapter on the DOM extension, pull parsers differ from tree parsers in that they read documents in a piecewise fashion rather than loading them into memory all at once. A consequence of this is that pull parsers generally only traverse documents once in one direction and leave you to collect whatever data is relevant to you along the way.

Before getting started, a noteworthy point is that XMLReader's underlying library, libxml, uses UTF-8 encoding internally. As such, encoding issues will be mitigated if any document you imported (particularly one that's been cleaned using the tidy extension) is encoded appropriately to avoid issues with conflicting encodings.

XML Parser
The XML Parser extension, as it is referred to in the PHP manual, is a predecessor of XMLReader and an alternative for PHP 4 environments. Its API is oriented to a more event-driven style of programming as opposed to the iterative orientation of the XMLReader extension. For more information on the XML Parser extension, see `http://php.net/manual/en/book.xml.php`.

Loading a Document

The focal class of the XMLReader extension is aptly named `XMLReader`. It doesn't declare a constructor, but rather offers two methods for introducing XML data into it.

```
<?php
// Loads a document contained within a string
$doc = XMLReader::xml($xmlString);

// Loads a document from an external file
$doc = XMLReader::open($filePath);
?>
```

Both of these methods have two additional parameters.

The second parameter is a string specifying the encoding scheme for the input document. It is optional and defaults to `'UTF-8'` if unspecified or specified as `null`. Valid values for this parameter aren't included in the PHP manual, but can be found in the reference for the underlying libxml2 library at `http://www.xmlsoft.org/encoding.html#Default`.

The third parameter is an integer value that can be set in bitmask fashion using constants from the libxml extension. This is the preferred method to configure the parser over using the deprecated `setParserProperty()` method. The specific constants that can be used to form the bitmask (using the bitwise OR operator `|`) are listed below. Descriptions for them can be found at `http://php.net/manual/en/libxml.constants.php`.

- `LIBXML_COMPACT`
- `LIBXML_DTDATTR`
- `LIBXML_DTDLOAD`
- `LIBXML_DTDVALID`
- `LIBXML_NOBLANKS`
- `LIBXML_NOCDATA`
- `LIBXML_NOENT`

- LIBXML_NOERROR
- LIBXML_NONET
- LIBXML_NOWARNING
- LIBXML_NSCLEAN
- LIBXML_XINCLUDE

As an example, a call that configured the parser to suppress errors and warnings might look like this.

```
<?php
$doc = XMLReader::xml(
    $xmlString,
    null,
    LIBXML_NOERROR | LIBXML_NOWARNING
);
?>
```

Iteration

The XMLReader instance acts as both an iterator over the entire document as well as a data object for the current node pointed to by that iterator. It contains a set of read-only properties (described at http://php.net/manual/en/class.xmlreader.php#xmlreader.props) that represent those of the current node and are updated as the iterator position changes.

```
<?php
while ($doc->read()) {
    // ...
}
?>
```

The read() method attempts to move the iterator position to the next node and returns a boolean value indicating whether or not it was successful. That is, it returns false once it reaches the end of the document. As such, it's perfect for use in a while loop as shown above.

Nodes

As in other extensions, each node has a type that is stored in the reader's nodeType property. The types in which you are generally interested are still elements (XMLReader::ELEMENT) and attributes (XMLReader::ATTRIBUTE), possibly also text (XMLReader::TEXT) and CDATA (XMLReader::CDATA) elements as well. Additionally, the XMLReader extension has a node type for ending elements (i.e. closing tags), XMLReader::END_ELEMENT. Its importance will become more obvious in the next section.

The example below shows how to check the node type against an appropriate constant. For a list of these constants, see http://php.net/manual/en/class.xmlreader.php#xmlreader.constants.

```
<?php
while ($doc->read()) {
    if ($doc->nodeType == XMLReader::ELEMENT) {
        var_dump($doc->localName);
        var_dump($doc->value);
        var_dump($doc->hasValue);
    }
}
?>
```

Also like other extensions, nodes have names. There are two properties for this, name and localName. The former represents the fully qualified name, including the namespace specification, while the latter represents the node name by itself and is the one you will generally want to use.

Elements and Attributes

Attribute nodes can have values. When the iterator points to an attribute node, the value property will be populated with the node's value and the hasValue property can be used to check for its presence.

Element nodes can have attributes. When the iterator points to an element node, the `hasAttributes` property indicates the presence of attributes and the `getAttribute()` method can be used to obtain an attribute value in the form of a string.

The example below uses both of these together to parse data from an HTML table.

```
<?php
$inTable = false;
$tableData = array();

while ($doc->read()) {
    switch ($doc->nodeType) {
        case XMLREADER::ELEMENT:
            if ($doc->localName == ’table’
                && $doc->getAttribute(’id’) == ’thetable’) {
                $inTable = true;
            } elseif ($doc->localName == ’tr’ && $inTable) {
                $row = count($tableData);
                $tableData[$row] = array();
            } elseif ($doc->localName == ’td’ && $inTable) {
                $tableData[$row][] = $doc->readString();
            }
            break;
        case XMLREADER::END_ELEMENT:
            if ($doc->localName == ’table’ && $inTable) {
                $inTable = false;
            }
            break;
    }
}
?>
```

This showcases the main difference between pull parsers and tree parsers: the former have no concept of hierarchical context, only of the node to which the iterator is currently pointing. As such, you must create your own indicators of context where they are needed.

In this example, the node type is checked as nodes are read and any node that isn’t either an opening or closing element is ignored. If an opening element is encountered, its name (`$doc->localName`) is evaluated to confirm that it’s a table and its `id` attribute value (`$doc->getAttribute(’id’)`) is also examined to confirm that it has a value of `’thetable’`. If so, a flag variable `$inTable` is set to `true`. This is used to indicate to subsequent if branch cases that the iterator points to a node that is within the desired table.

The next if branch is entered when table row elements within the table are encountered. A combination of checking the node name and the previously set $inTable flag facilitates this. When the branch is entered, a new element in the $tableData array is initialized to an empty array. This array will later store data from cells in that row. The key associated with the row in $tableData is stored in the $row variable.

Finally, the last if branch executes when table cell elements are encountered. Like the row branch, this branch checks the node name and the $inTable flag. If the check passes, it then stores the current node's value in the array associated with the current table row.

Here's where the XMLREADER::END_ELEMENT node type comes into play. Once the end of the table is reached, no further data should be read into the array. So, if the ending element has the name 'table' and the $inTable flag currently indicates that the iterator points to a node within the desired table, the flag is then set to false. Since no other tables should theoretically have the same id attribute, no if branches will execute in subsequent while loop iterations.

If this table was the only one of interest in the document, it would be prudent to replace the $inTable = false; statement with a break 2; statement. This would terminate the while loop used to read nodes from the document as soon as the end of the table was encountered, preventing any further unnecessary read operations.

readString() Availability

As its entry in the PHP manual notes, the readString() method used in the above example is only present when the XMLReader extension is compiled against certain versions of the underlying libxml library.

If this method is unavailable in your environment, an alternative in the example would be to have opening and closing table cell checks that toggle their own flag ($inCell for example) and switch cases for the TEXT and CDATA node types that check this flag and, when it is set to true, add the contents of the value property from the XMLReader instance to the $tableData array.

DOM Interoperation

One nice feature of the XMLReader extension is the expand() method, which returns an object of the equivalent DOM extension class for the node to which the iterator currently points. Element nodes, for example, cause this method to return a DOMElement instance.

The example below illustrates a modification to the previous example and provides another alternative to the readString() method for getting at the contents of table cells by taking advantage of this DOM interoperability.

```
<?php
if ($doc->localName == ’td’ && $inTable) {
    $node = $doc->expand();
    $tableData[$row][] = $node->nodeValue;
}
?>
```

Closing Documents

Once all necessary read operations have been conducted on a document, the close() method of the XMLReader instance should be called to cleanly terminate file access or otherwise indicate that access to the original data source is no longer needed.

Wrap-Up

Hopefully this chapter has given you an idea of how pull parsers work in general and when it’s better to use one like XMLReader over a tree parser. For more information on XMLReader, check out its section in the PHP manual at http://www.php.net/manual/en/book.xmlreader.php.

Chapter 13

CSS Selector Libraries

This chapter will review several libraries that are built on top of the XML extensions described in previous chapters. These libraries provide interfaces that uses CSS selector expressions to query markup documents rather than a programmatic API or XPath expressions. Don't be concerned if you aren't familiar with CSS selectors, as part of this chapter showcases basic expressions alongside their XPath equivalents.

> **CSS Versions**
> There are multiple versions of the CSS standard and supported selectors vary with each version. This chapter will cover a subset of those available in CSS3. Versions of the CSS standard supported by particular libraries are noted where available. A list of differences between the two common versions, CSS2 and CSS3, can be found at `http://www.w3.org/TR/css3-selectors/#changesFromCSS2`.

Reason to Use Them

Before getting into the "how" of using CSS selector libraries, it's probably best to get the "why" (and "why not") out of the way first. It goes without saying that these libraries add a layer of complexity to applications that use them, introducing another potential point of failure. They implement expression parsers in order to take CSS selector expressions as their input and translate them into equivalent userland operations, which can have an impact on performance.

Those points aside, they do offer a syntax that is considered by some to be more intuitive. Most developers these days know at least some CSS and probably have a little experience with a JavaScript library that uses CSS selectors for node selection on the client side, such as jQuery. Thus, knowledge of that particular area is transferable to the server side when libraries based around it are used.

To sum this up, if you're particularly concerned about performance or simplicity of the application with regard to the number of components on which it is dependent, it's probably best to stick with something like XPath. Otherwise, CSS selector libraries are worthy of consideration for making use of a common existing skillset.

Even if your decision is to stick with XPath, keep reading. This chapter will also show some XPath equivalents for each set of explained CSS selectors, which may help to further your understanding of XPath. Note that these comparions are not necessarily comprehensive and there may be multiple ways to express the same thing in any given case.

> **jQuery Examples**
> The documentation for the jQuery library itself actually has excellent visual client-side examples of selectors. If you find you aren't certain that you understand any of the descriptions of CSS selectors that follow, the jQuery demos and source code will likely prove to be helpful supplements. You can find them at `http://api.jquery.com/category/selectors/`.

Basics

Let's look at a few basic selectors and their results when applied to a markup example.

```
<html>
<body>
<div id="nav">
    <ul class="horizontal">
        <li><a href="/home">Home</a></li>
        <li><a href="/about-us">About Us</a></li>
        <li><a href="/contact-us">Contact Us</a></li>
    </ul>
    <img src="/img/ad1.jpg" alt="Advertisement #1">
    <img src="/img/ad2.jpg" alt="Advertisement #2">
</div>
</body>
</html>
```

- `#nav` would select the `div` element because it has an `id` attribute value of `nav`.
- `li` would select all `li` elements by their node name.
- `.horizontal` would select the `ul` element because it has a class of `horizontal`. (Note that elements can have multiple classes.)
- `*` would select all elements in the document.
- `li, a` would select all `li` and `a` elements in the document by combining the two selectors `li` and `a` into a comma-delimited list.

Here are the XPath equivalents along side their respective CSS counterparts. Aside from the `.class` selector, the XPath expressions are not significantly longer or more complex.

Selector	**CSS**	**XPath**
id	`#nav`	`//*[@id=“nav”]`
element	`li`	`//li`

class	`.horizontal`	`//*[@class=“horizontal”`
		`or starts-with(@class, “horizontal ”)`
		`or contains(@class, “ horizontal ”)`
		`or ends-with(@class, “ horizontal”)]`

wildcard	`*`	`//*`
multiple	`li, a`	`//li|//a`

Hierarchical Selectors

Now let's move on to hierarchical selectors, which use document structure as a way to select elements, using the previous markup example.

- body ul would select all ul elements that are descendants of body elements.
- ul > li would select all li elements that are children of ul elements.
- ul + img would select all img elements that immediately follow a ul sibling element (i.e. are their next sibling).
- ul ˜ img would select all img elements that follow a ul sibling element (i.e. come after them and have the same parent node).

The third and fourth items in this list probably look fairly similar. The difference is in the word "immediately." By this example, the third item in the list would only select the Advertisment #1 image because it comes immediately after its ul sibling element in the document. The fourth item in the list, on the other hand, would select both images because both follow their ul sibling element.

Selector	CSS	XPath
ancestor descendant	body ul	*body*ul
parent > child	ul > li	//ul/li
prev + next	ul + img	//ul/following-sibling::img[1]
prev ˜ siblings	ul ˜ img	//ul/following-sibling::img

Basic Filters

The selectors reviewed up to this point in the chapter have always changed the type of nodes being selected. Conversely, when a filter is appended to an expression, it merely restricts the returned set of nodes to a subset of those matching the original expression.

Note that available filters vary per library. Support for filters in jQuery is fairly comprehensive and as such it is used as the primary reference for sections related to filters in this chapter.

- `li:first` selects only the first `li` node found in the document.
- `li:last` likewise selects the last `li` node found in the document.
- `li:even` selects all evenly positioned nodes in the document beginning from `0`.
- `li:odd` likewise selects all oddly positioned nodes in the document, also beginning from `0`.
- `li:eq(0)` selects the `li` node with a position of `0` within the set of `li` nodes (i.e. the first one) in the document.
- `li:gt(0)` selects all `li` nodes with a position greater than `0` within the set of `li` nodes (i.e. all but the first one) in the document.
- `li:lt(1)` selects all `li` nodes with a position less than 1 within the set of `li` nodes (i.e. the first one) in the document.
- `:header` matches all header nodes. (i.e. h1, h2, etc.)
- `:not(:first)` negates the `:first` selector and thus selects all `li` nodes except the first one in the document.

Selector	**CSS**	**XPath**
first node	`li:first`	`//li[1]`
last node	`li:last`	`//li[last()]`
even nodes	`li:even`	`//li[position() mod 2 = 0]`
odd nodes	`li:odd`	`//li[position() mod 2 = 1]`
specific node	`li:eq(0)`	`//li[1]`
all nodes after	`li:gt(0)`	`//li[position() > 1]`
all nodes before	`li:lt(1)`	`//li[position() < 2]`
header nodes	`:header`	`//h1|//h2|//h3|//h4|//h5|//h6`
all nodes not matching an expression	`:not(:first)`	`//*[not(position() = 1)]`

Note when reading this table that CSS selectors begin set indices at `0` whereas XPath begins them at `1`.

Content Filters

Where basic filters are based mainly on the type of node or its position in the result set, content filters are based on node value or surrounding hierarchical structure.

- `a:contains(“About Us”);` selects all `a` nodes where the node value contains the substring “`About Us`”.
- `img:empty` selects all `img` nodes that contain no child nodes (including text nodes).
- `li:has(a:contains(“About Us”))` selects all `li` nodes that contain an `a` node with the substring “`About Us`” in its node value.
- `li:parent` selects all `li` nodes that contain child nodes (including text nodes).

Selector	CSS	XPath
nodes containing text	`a:contains(“About Us”)`	`//a[contains(text(), “About Us”)]`
nodes without children	`img:empty`	`//img[not(node())]`
nodes containing a selector match	`li:has(a:contains (“About Us”))`	`//li//a[contains( text(), “About Us”)]`
nodes with children	`li:parent`	`//li[node()]`

Attribute Filters

Filters up to this point have been specific to element nodes, but they also exist for attribute nodes. Attribute filters are surrounded by square brackets in both CSS and XPath, but differ in that CSS uses mostly operators for conditions while XPath uses mostly functions. Unlike other filters described in this chapter, support for attribute filters is fairly universal between different libraries.

- `[href]` matches all nodes that have an attribute node with the name `href`.
- `[href="/home"]` matches all nodes with an attribute node named `href` that has a value of "`/home`".
- `[href!="/home"]` matches all nodes with an attribute node named `href` that do not have a value of "`/home`".
- `[hrefˆ="/"]` matches all nodes with an attribute node named `href` and have a value that starts with "`/`".
- `[href$="-us"]` matches all nodes with an attribute node named `href` and have a value that ends with "`-us`".
- `[href*="-us"]` matches all nodes with an attribute node named `href` and have a value that contains "`-us`" anywhere within the value.
- `[src*="ad"][altˆ="Advertisement"]` matches all nodes that have both an attribute node named `src` with a value containing "`ad`" and an attribute node named `alt` with a value starting with "`Advertisement`".

Selector	**CSS**	**XPath**
has attribute	`[href]`	`//*[@href]`
has attribute value	`[href="/home"]`	`//*[@href="/home"]`
has different attribute value	`[href!="/home"]`	`//*[@href!="/home"]`
has attribute value starting with substring	`[hrefˆ="/"]`	`//*[starts-with(@href, "/")]`
has attribute value ending with substring	`[href$="-us"]`	`//*[ends-width(@href, "-us")]`
has attribute value containing substring	`[href*="-us"]`	`//*[contains(@href, "-us")]`
multiple attribute filters	`[src*="ad"][altˆ= "Advertisement"]`	`//*[contains(@src, "ad") and starts-with(@alt, "Advertisement")]`

Child Filters

Child filters are fairly similar to the basic filters reviewed earlier, except applied to child nodes.

- `ul:nth-child(2)` selects the second child element within each `ul` node. The parameter passed to the filter can also be `even` or `odd` (which are figured relative to child position within the parent element) or can use expressions involving a variable `n` (such as `3n` for every third child).

- `li:first-child` selects all `li` nodes that are the first child of their parent node.

- `li:last-child` likewise selects all `li` nodes that are the last child of their parent node.

- `li:only-child` selects all `li` nodes that are the only child of their parent node.

Selector	CSS	XPath
nth child nodes	`ul:nth-child(2)`	`//ul/*[position() = 2]`
first child nodes	`li:first-child`	`//*/*[name() = “li” and position() = 1]`
last child nodes	`li:last-child`	`//*/*[name() = “li” and position() = last()]`
only child nodes	`li:only-child`	`//*/*[name() = “li” and count() = 1]`

Form Filters

Form filters are just a more convenient shorthand for other expressions.

- `:input` matches all input, textarea, select, and button elements.

- `:text`, `:hidden`, `:password`, `:radio`, `:checkbox`, `:submit`, `:image`, `:reset`, and `:file` all matches input elements with their respective types.

- `:button` matches all button elements and input elements of type button.

- `:enabled` matches all form elements that are not disabled, `:disabled` matches those that are.
- `:checked` matches all radio and checkbox elements that are checked.
- `:selected` matches all option elements that are selected.

Selector	CSS	CSS Alt	XPath
all form elements	:input	input, textarea, select, button	//input|//textarea|//select|//button
form elements of specific types	:text	input [type=“text”]	//input[type=“text”]
button elements	:button	button, input[type=“button”]	//button|//input [type=“button”]
enabled elements	:enabled	:not([disabled=“disabled”])	//*[contains(“input textarea select button”, name()) and (not(@disabled) or @disabled!=“disabled”])
disabled elements	:disabled	[disabled=“disabled”]	//*[contains(“input textarea select button”, name()) and @disabled=“disabled”]
checked elements	:checked	:input[checked=“checked”]	//input[contains (“checkbox radio”, @type) and @checked=“checked”]
selected elements	:selected	option[selected=“selected”]	//option[@selected=“selected”]

Libraries

At this point, CSS selectors have been covered to the extent that all or a subset of those supported by a given library are explained. This section will review some library implementations that are available, where to find them, what feature set they support, and some advantages and disadvantages of using them.

PHP Simple HTML DOM Parser

The major distinguishing trait of this library is its requirements: PHP 5 and the PCRE extension (which is pretty standard in most PHP distributions). It has no external dependencies on or associations with other libraries or extensions, not even the standard XML extensions in PHP.

The implication of this is that all parsing is handled in PHP itself, which makes it likely that performance will not be as good as libraries that build on a PHP extension. However, in environments where XML extensions (in particular the DOM extension) may not be available (which is rare), this library may be a good option. It offers basic retrieval support using PHP's filesystem functions (which require the configuration setting `allow_url_fopen` to be enabled to access remote documents).

The documentation for this library is fairly good and can be found at `http://simplehtmldom.sourceforge.net/manual.htm`. Its main web site, which includes a link to download the library, is available at `http://simplehtmldom.sourceforge.net`. It is licensed under the MIT License.

Zend_Dom_Query

One of the components of Zend Framework, this library was originally created to provide a means for integration testing of applications based on the framework. However, it can function independently and apart from the framework and provides the functionality needed in the analysis phase of web scraping. At the time of this writing, Zend Framework 1.10.1 requires PHP 5.2.4 or higher.

Zend_Dom_Query makes extensive use of the DOM extension. It supports XPath through use of the DOM extension's DOMXPath class and handles CSS expressions by transforming them into equivalent XPath expressions. Note that only CSS 2 is supported, which excludes non-attribute filters.

It's also worth noting that Zend_Dom_Query offers no retrieval functionality. All methods for introducing documents into it require that those documents be in string form beforehand. If you are already using Zend Framework, a readily available option for retrieval is Zend_Http_Client, which is also discussed in this book.

Documentation for Zend_Dom_Query can be found at `http://framework.zend.com/manual/en/zend.dom.query.html`. At this time, there is no officially supported method of downloading only the Zend_Dom package. The entire framework can be downloaded from `http://framework.zend.com/download/current/` and the directory for the Zend_Dom package can be extracted from it. An unofficial method of downloading individual packages can be found at `http://epic.codeutopia.net/pack/`. Zend Framework components are licensed under the New BSD License.

phpQuery

phpQuery is heavily influenced by jQuery and maintains similarity to it insofar as its runtime environment being the server (as opposed to the client) will allow. It requires PHP 5.2 and the DOM extension as well as the Zend_Http_Client and Zend_Json components from Zend Framework, which are bundled but can be substituted with the same components from a local Zend Framework installation.

CSS support is limited to a subset of CSS3. Most jQuery features are supported including plugin support, with porting of multiple jQuery plugins being planned. Other components include a CLI utility that makes functionality from the phpQuery library available from command line and a server component for integrating with jQuery via calls made from it on the client side. Retrieval support is included in the form of integration with Zend_Http_Client.

Documentation and download links are available from `http://code.google.com/p/phpquery/`. It is licensed under the MIT License.

DOMQuery

This library is actually a project of my own. While still in alpha at the time of this writing, it is fairly functional and includes a full unit test suite. Like some of the other libraries mentioned in this chapter, it requires PHP 5 and makes heavy use of the DOM extension.

Unlike the others, however, it does not implement a CSS selector parser in order to offer related functionality. Instead, it does so programmatically through its API. For example, rather than passing the name of an element (say `div`) to a central query method, an `element()` method accepts the name of an element for which to query. Though this makes it a bit less concise than other libraries, it also makes it more expressive and only requires a basic knowledge of DOM concepts in order to operate it.

It can be downloaded at `http://github.com/elazar/domquery/tree/master`. The central class `DOMQuery` is documented using phpDoc-compatible API docblocks and the unit test suite offers use cases for each of the available methods.

Wrap-Up

This concludes the current chapter. You should come away from reading it with knowledge of the pros and cons of using CSS selector-based libraries and situations where their use is appropriate, specific CSS selectors and possible equivalents for them in XPath, and particular library implementations to consider.

Chapter 14

PCRE Extension

There are some instances where markup documents may be so hideously malformed that they're simply not usable by an XML extension. Other times, you may want to have a way to check the data you've extracted to ensure that it's what you expect. Changes to the structure of markup documents may be significant, to the point where your CSS or XPath queries return no results. They may also be small and subtle, such that while you do get query results, they contain less or different data than intended.

While either of these tasks could be done with basic string handling functions and comparison operators, in most cases the implementation would prove to be messy and unreliable. **Regular expressions** provide a syntax consisting of **meta-characters** whereby patterns within strings are expressed flexibly and concisely. This chapter will deal with regular expressions as they relate to the Perl-Compatible Regular Expression (PCRE) PHP extension in particular.

A common bad practice is to use only regular expressions to extract data from markup documents. While this may work for simple scripts that are only intended to be used once or very few times in a short time period, it is more difficult to maintain and less reliable in the long term. Regular expressions simply were not designed for this purpose, whereas other markup-specific extensions discussed in previous chapters are more suited for the task. It is a matter of using the best tool for the job, and to that end, this practice should be avoided.

POSIX Extended Regular Expressions
Many PHP developers will cut their teeth on regular expressions using the POSIX regular expression extension, also called the ereg extension. The functions from this extension are being deprecated in PHP 5.3 in favor of those in the PCRE extension, which are faster and provide a more powerful feature set. Aside from differences in syntax for some special character ranges, most ereg expressions require only the addition of expression delimiters to work with preg functions.

Pattern Basics

Let’s start with something simple: detection of a substring anywhere within a string.

```
<?php
// Substring detection with a basic string function
$present = (strpos($string, ’foo’) !== false);

// Same with a PCRE function
$present = (preg_match(’/foo/’, $string) == 1);
?>
```

Notice that the pattern in the `preg_match()` call is fairly similar to the string used in the `strpos()` call. In the former, `/` is used on either side of the pattern to indicate its beginning and end. The first character in the pattern string is considered to be the **pattern delimiter** and can be any character you specify. When choosing what you want to use for this character (`/` is the most common choice), bear in mind that you will have to escape it (covered in the Escaping section later) if you use it within the pattern. This will make more sense a little later in the chapter.

A difference between the two functions used in this example is that `strpos()` returns the location of the substring within the string beginning at `0` or `false` if the substring is not contained within the string. This requires the use of the === operator to tell the difference between the substring being matched at the beginning of the string or not at all. By contrast, `preg_match()` returns the number of matches it found. This will be either `0` or `1` since `preg_match()` stops searching once it finds a match.

Anchors

You may want to check for the presence of a pattern at the beginning or end of a string rather than simply checking to see if the pattern is contained anywhere within the string. The meta-characters for this are collectively referred to as **anchors**.

```
<?php
// Beginning of the string with a basic string function
$start = (strpos($string, ’foo’) === 0);

// Same with a PCRE function
$start = (preg_match(’/^foo/’, $string) == 1);

// End of the string with basic string functions
$end = (substr($string, - strlen(’foo’)) == ’foo’);

// Same with a PCRE function
$end = (preg_match(’/foo$/’, $string) == 1);

// Means the same as an exact match with the string
$equal = (preg_match(’/^foo$/’, $string) == 1);
?>
```

- ˆ (also called the circumflex character) is used at the beginning of an expression within a pattern to indicate that matching should start at the beginning of a string.
- Likewise, `$` is used to indicate that matching of an expression within a pattern should stop at the end of a string.
- When used together, ˆ and `$` can indicate that the entirety of `$string` matches the pattern exactly.

Start of String or Line
It’s important to note that the behavior of these two operators can vary. By default, they match the beginning and end of `$string`. If the multi-line modifier is used, they match the beginning and end of each line in `$string` instead. This will be covered in more detail later in the Modifiers section of this chapter.

Alternation

It's possible to check for multiple expressions simultaneously in a single pattern, also called **alternation**, using the pipe meta-character `|`.

```
<?php
// Matches 'foo' or 'bar' or 'baz' anywhere
$matches = (preg_match('/foo|bar|baz/', $string) == 1);
?>
```

Note that the `^` and `$` are not implicitly applied to all expressions in an alternation; they must be used explicitly for each expression.

```
<?php
// $result == 1
$result = preg_match('/^foo|bar/', 'abar');

// $result == 0
$result = preg_match('/^foo|^bar/', 'abar');
?>
```

The first example returns `1` because `'abar'` contains `'bar'`, since `^` is not applied to that expression in the pattern. The second example does apply `^` to `'bar'` and, since `'abar'` begins with neither `'foo'` nor `'bar'`, it returns `0`.

Repetition and Quantifiers

Part of a pattern may or may not be present, or may be repeated a number of times. This is referred to as **repetition** and involves using meta-characters collectively referred to as **quantifiers**.

```
<?php
// Matches 'a' 0 times or 1 time if present
$matches = (preg_match('/a?/', $string) == 1);

// Matches 'a' 0 or more times, however many that may be
$matches = (preg_match('/a*/', $string) == 1);
```

```
// Matches ’a’ 1 or more times, however many that may be
$matches = (preg_match(’/a+/’, $string) == 1);

// Matches ’a’ 0 times or 1 time if present, same as ?
$matches = (preg_match(’/a{0,1}/’, $string) == 1);

// Matches ’a’ 0 or more times, same as *
$matches = (preg_match(’/a{0,}/’, $string) == 1);

// Matches ’a’ 1 or more times, same as +
$matches = (preg_match(’/a{1,}/’, $string) == 1);

// Matches ’a’ exactly 2 times
$matches = (preg_match(’/a{2}/’, $string) == 1);

?>
```

Note that any use of curly brackets that is not of the form `{X}`, `{X,}`, or `{X,Y}` will be treated as a literal string within the pattern.

Subpatterns

You’ll notice in the examples from the previous section that only a single character was used. This is because the concept of **subpatterns** hadn’t been introduced yet. To understand these, it’s best to look an example that doesn’t use them in order to understand the effect they have on how the pattern matches.

```
<?php
// Matches ’a’ follow by one or more instances of ’b’
$matches = (preg_match(’/ab+/’, $string) == 1);
?>
```

Without subpatterns there would be no way to match, for example, one or more instances of the string `'ab`’. Subpatterns solve this pattern by allowing individuals parts of a pattern to be grouped using parentheses.

```
<?php
// Matches ’ab’ one or more times
$matches = (preg_match(’/(ab)+/’, $string) == 1);
```

```
// Matches ’foo’ or ’foobar’
$matches = (preg_match(’/foo(bar)?/’, $string) == 1);

// Matches ’ab’ or ’ac’
$matches = (preg_match(’/a(b|c)/’, $string) == 1);

// Matches ’ab’, ’ac’, ’abb’, ’abc’, ’acb’, ’acc’, etc.
$matches = (preg_match(’/a(b|c)+/’, $string) == 1);
?>
```

Matching

Subpatterns do a bit more than let you define parts of a pattern to which alternation or repetition apply. When a match is found, it’s possible to obtain not only the substring from the original string that matched the entire pattern, but also substrings that were matched by subpatterns.

```
<?php
if (preg_match(’/foo(bar)?(baz)?/’, $string, $match) == 1) {
    print_r($match);
}
?>
```

The third parameter to `preg_match()`, `$match`, will be set to an array of match data if a match is found. That array will contain at least one element: the entire substring that matched the pattern. Any elements that follow will be subpattern matches with an index matching that subpattern’s position within the pattern. That is, the first subpattern will have the index `1`, the second subpattern will have the index `2`, and so on.

If a pattern is conditional (i.e. uses `?`) and not present, it will either have an empty element value in the array or no array element at all.

```
<?php
if (preg_match(’/foo(bar)?/’, ’foo’, $match) == 1) {
    // $match == array(’foo’);
}
```

```
if (preg_match('/foo(bar)?(baz)?/', 'foobaz', $match) == 1) {
    // $match == array('foo', '', 'baz');
}
?>
```

- In the first example, the `(bar)?` subpattern ends the entire pattern and is not matched. Thus, it has no entry in `$match`.
- In the second example, the `(bar)?` subpattern does not end the entire pattern and is not matched. Thus, it has an empty entry in `$match`.

Subpatterns can also contain other subpatterns.

```
<?php
if (preg_match('/foo(ba(r|z))?/', 'foobar', $match) == 1) {
    // $match == array('foobar', 'bar', 'r');
}
?>
```

Aside from passing `$match` to `print_r()` or a similar function, an easy way to tell what a subpattern's position will be in `$match` is to count the number of opening parentheses in the pattern from left to right until you reach the desired subpattern.

Using the syntax shown above, any subpattern will be **captured** (i.e. have its own element in the array of matches). Captured subpatterns are limited to 99 and total subpatterns, captured or no, is limited to 200. While this realistically shouldn't become an issue, it's best to denote subpatterns that do not require capture using `(?:` instead of `(` to begin them.

Additionally, since PHP 4.3.3, subpatterns may be assigned meaningful names to be used as their indices in the array of matches when they are captured. To assign a name to a subpattern, begin it with the syntax `(?P<name>` instead of `(` where `name` is the name you want to assign to that subpattern. This has the advantage of making code more expressive and easier to maintain as a result.

Escaping

There may be instances where you want to include literal characters in patterns that are usually interpreted as meta-characters. This can be accomplished via a \ meta-character.

```
<?php
// Matches literal [
$matches = (preg_match('/\[/', $string) == 1);

// Matches literal \
$matches = (preg_match('/\\/', $string) == 1);

// Matches expression delimiter /
$matches = (preg_match('/\//', $string) == 1);

// Matches any of the standard escape sequences \r, \n, or \t
$matches = (preg_match('/\r|\n|\t/', $string) == 1);
?>
```

Note that it is necessary to double-escape ” in the second example because the string ’
’ is interpreted to be a single backslash by PHP whether or not it is used in a regular expression. In other cases, no escaping of ” is needed for the escape sequence to be interpreted properly.

> **Double Escaping**
> For more information on the reasoning behind the double-escape example in this section, see http://php.net/manual/en/language.types.string.php#language.types.string and the Backslash section of http://php.net/manual/en/regexp.reference.php.

Escape Sequences

There are three ways to match a single character that could be one of several characters.

The first way involves using the . meta-character, which will match any single character except a line feed ("\n") without use of modifiers (which will be covered later). This can be used with repetition just like any other character.

The second way requires using special escape sequences that represent a range of characters. Aside from the escape sequences mentioned in the previous section's examples, here are some that are commonly used.

- `\d`: a digit, 0 through 9.
- `\h`: a horizontal whitespace character, such as a space or a tab.
- `\v`: a vertical whitespace character, such as a carriage return or line feed.
- `\s`: any whitespace character, the equivalent of all characters represented by `\h` and `\v`.
- `\w`: any letter or digit or an underscore.

Each of these escape sequences has a complement.

- `\D`: a non-digit character.
- `\H`: a non-horizontal whitespace character.
- `\V`: a non-vertical whitespace character.
- `\S`: a non-whitespace character.
- `\W`: a character that is not a letter, digit, or underscore.

The third and final way involves using **character ranges**, which are characters within square brackets (`[` and `]`). A character range represents a single character, but like normal single characters they can have repetition applied to them.

```
<?php
// Matches the same as \d
$matches = (preg_match('/[0-9]/', $string) == 1);

// Matches the same as \w
$matches = (preg_match('/[a-zA-Z0-9_]/', $string) == 1);
?>
```

Ranges are respective to ASCII (American Standard Code for Information Interchange). In other words, the ASCII value for the beginning character must precede the ASCII value for the ending character. Otherwise, the warning "Warning: preg_match(): Compilation failed: range out of order in character class at offset *n*" is emitted, where *n* is character offset within the regular expression.

Within square brackets, single characters and special ranges are simply listed side by side with no delimiter, as shown in the second example above. Additionally, the escape sequences mentioned earlier such as `\w` can be used both inside and outside square brackets.

ASCII Ranges
For an excellent ASCII lookup table, see `http://www.asciitable.com`.

There are two other noteworthy points about character ranges, as illustrated in the examples below.

```
<?php
// Using a literal ] in a character range is done like so
$matches = (preg_match('/[\]]/', $string) == 1);

// Matches any character that is not 'a'
$matches = (preg_match('/[^a]/', $string) == 1);

// Using a literal ^ in a character range is done like so
$matches = (preg_match('/[\^]/', $string) == 1);
$matches = (preg_match('/[a^]/', $string) == 1);
?>
```

- To use a literal `]` character in a character range, escape it in the same manner in which other meta-characters are escaped.
- To negate a character range, use ˆ as the first character in that character range. (Yes, this can be confusing since ˆ is also used to denote the beginning of a line or entire string when it is not used inside a character range.) Note that negation applies to all characters in the range. In other words, a negated character range means "any character that is not any of these characters."

- To use a literal ^ character in a character range, either escape it in the same manner in which other meta-characters are escaped or do not use it as the first or only character in the range.

> **ctype Extension**
> Some simple patterns have equivalent functions available in the ctype library. These generally perform better and should be used over PCRE when appropriate. See `http://php.net/ctype` for more information on the ctype extension and the functions it offers.

Modifiers

The reason for having pattern delimiters to denote the start and end of a pattern is that the pattern precedes **modifiers** that affect the matching behavior of meta-characters. Here are a few modifiers that may prove useful in web scraping applications.

- `i`: Any letters in the pattern will match both uppercase and lowercase regardless of the case of the letter used in the pattern.

- `m`: ^ and `$` will match the beginning and ends of lines within the string (delimited by line feed characters) rather than the beginning and end of the entire string.

- `s` (lowercase): The `.` meta-character will match line feeds, which it does not by default.

- `S` (uppercase): Additional time will be spent to analyze the pattern in order to speed up subsequent matches with that pattern. Useful for patterns used multiple times.

- `U`: By default, the quantifiers `*` and `+` behave in a manner referred to as "greedy." That is, they match as many characters as possible rather than as few as possible. This modifier forces the latter behavior.

- u: Forces pattern strings to be treated as UTF-8 encoded strings.

The example below matches because the i modifier is used, which means that the pattern matches 'a' and 'A'.

```
<?php
$matches = (preg_match('/a/i', 'A') == 1);
?>
```

Wrap-Up

This chapter has covered the most essential aspects of regular expressions that apply to validation of scraped data. There are more advanced aspects of regular expressions that may be useful in other areas. Further review of the PCRE section of the PHP manual is encouraged.

For pattern modifiers, see http://php.net/manual/en/reference.pcre.pattern.modifiers.php.

For pattern syntax, see http://php.net/manual/en/reference.pcre.pattern.syntax.php.

For an excellent book on regular expressions, see “Mastering Regular Expressions,” ISBN 0596528124.

Chapter 15

Tips and Tricks

Chapters preceding this one are intended to lay out pertinent information about the skills required to build a web scraping application. This chapter focuses more on a collection of practical applications and methodologies for that material. It is meant to tie all previously covered material together and round out the book as a whole. Some topics covered here will be specific to a particular class of web scraping applications, some will be generally applicable, and most will assume that the application is intended for long-term use. Hopefully they will all illustrate what matters to consider when designing your application.

Batch Jobs

Web scraping applications intended for long-term use generally function in one of two ways: real-time or batch.

A web scraping application implemented using the real-time approach will receive a request and send a request out to the target application being scraped in order to fulfill the original request. There are two advantages to this. First, any data pulled from the target application will be current. Second, any data pushed to the target application will be reflected on that site in nearly the same amount of time it would take if the data had been pushed directly to the target application. This approach has the disadvantage of increasing the response time of the web scraping application, since the client essentially has to wait for two requests to complete for every one request that would normally be made to the target application.

The batch approach is based on synchronization. For read operations, data is updated on a regular interval. For write operations, changes are stored locally and then pushed out in batches (hence the name) to the target application, also on a regular interval. The pros and cons to this approach are the complement of those from the real-time approach: updates will not be real-time, but the web scraping application's response time will not be increased. It is of course possible to use a batch approach with a relatively low interval in order to approximate real-time while gaining the benefits of the batch approach.

The selection of an approach depends on the requirements of the web scraping application. In general, if real-time updates on either the web scraping application or target application are not required, the batch approach is preferred to maintain a high level of performance.

Availability

Regardless of whether a web scraping application takes a real-time or batch approach, it should treat the remote service as as potential point of failure and account for cases where it does not return a response. Once a tested web scraping application goes into production, common causes for this are either service downtime or modification. Symptoms of these include connection timeouts and responses with a status code above the 2xx range.

An advantage of the batch approach in this situation is that the web scraping application's front-facing interface can remain unaffected. Cached data can be used or updates can be stored locally and synchronization can be initiated once the service becomes available again or the web scraping application has been fixed to account for changes in the remote service.

Parallel Processing

Two of the HTTP client libraries previously covered, cURL and pecl_http, support running requests in parallel using a single connection. While the same feature cannot be replicated exactly using other libraries, it is possible to run multiple requests on separate connections using processes that are executed in parallel.

Even if you are using a library supporting connection pooling, this technique is useful for situations when multiple hosts are being scraped since each host will require a separate connection anyway. By contrast, doing so in a single process means it is possible for requests sent earlier to a host with a lower response rate to block those sent later to another more responsive host.

See Appendix B for a more detailed example this.

Crawlers

Some web scraping applications are intended to serve as crawlers to index content from web sites. Like all other web scraping applications, the work they perform can be divided into two categories: retrieval and analysis. The parallel processing approach is applicable here because each category of work serve to populate the work queue of the other.

The retrieval process is given one or more initial documents to retrieve. Each time a document is retrieved, it becomes a job for the analysis process, which scrapes the markup searching for links (a elements) to other documents, which may be restricted by one or more relevancy factors. Once analysis of a document is complete, addresses to any currently unretrieved documents are then fed back to the retrieval process.

This situation of mutual supply will hypothetically be sustained until no documents are found that are unindexed or considered to be relevant. At that point, the process can be restarted with the retrieval process using appropriate request headers to check for document updates and feeding documents to the analysis process where updates are found.

Forms

Some web scraping applications must push data to the target application. This is generally accomplished using HTTP POST requests that simulate the submission of HTML forms. Before such requests can be sent, however, there are a few events that generally have to transpire. First, if the web scraping application is intended to be presented to a user, a form that is at least somewhat similar must be presented to that user. Next, data submitted via that form by the user should be validated to ensure that it is likely to be accepted by the target application.

The applicability of this technique will vary by project depending on requirements and how forms are structured. It involves scraping the markup of the form in the target application and using the scraped data to generate something like a metadata file or PHP source code file that can be dropped directly into the web scraping application project. This can be useful to expedite development efforts for target applications that have multiple forms or complex forms for which POST requests must be simulated.

For the purposes of formulating a POST request, you will want to query for elements with the names `input`, `select`, `textarea`, or possibly `button` that have a `name` attribute. Beyond that, here are a few element-specific considerations to take into account.

- `input` elements with a `type` attribute value of `checkbox` or `radio` that are not checked when the form on the web scraping application side is submitted should not have their `value` attribute values included when the POST request is eventually made to the target application. A common practice that negates this is positioning another element with a `type` attribute value of `hidden` with the same name as the checkbox element before that element in the document so that the value of the hidden element is assumed if the checkbox is not checked.

- `select` elements may be capable of having multiple values depending on whether or not the `multiple` attribute is set. How this is expressed in the POST request can depend on the platform on which the target application is running. The best way to determine this is to submit the form on the target application via a client that can show you the underlying POST request being made.

- `input` elements that have a `maxlength` attribute are restricted to values of that length or less. Likewise, `select` elements are restricted to values in the `value` attributes of their contained `option` child elements. Both should be considered when validating user-submitted data.

Web Services

Web scraping applications are often built because the target application offers no web service or data formatted for automated consumption. However, some of these sites do eventually come to offer a web service after the web scraping application has already been built. As such, it's important to keep this potential eventuality in mind during the design phase.

The introduction of a web service will not negate previously described concerns regarding retrieval. Latency can still prove to be an issue for an application that attempts to access either the target application or a corresponding web service in real-time. Ideally, complexity will be reduced and performance increased in the analysis process when switching from a web scraping application to an API.

However, both areas of code will likely need to be replaced if an API offering does materialize. As such, it's important to design an API that approximates a hypothetical web service offering as closely as possible and centralizes logic that will need to be replaced in that event. By doing so, existing local application logic that uses the existing API will require little or no change.

Legalities aside (see Appendix A for more on those), there are reasons to consider maintaining an existing web scraping application over a new web service offering. These can include web service data offerings being limited or incomplete by comparison or uptime of the web service being below an acceptable tolerance. In the former case, web scraping logic can be replaced with web service calls where the two data offerings overlap for increased data reliability. In the latter case, the web scraping logic can conduct web service calls when the service is available and use cached data or store data updates locally until the service becomes available again.

Testing

As important as unit testing is to quality assurance for an application, it's all the more important to a web scraping application because it's reliant on its target to remain unchanged. Queries of markup documents must be checked to assert that they produce predictable results and data extracted from those markup documents must be validated to ensure that it is consistent with expectations. In the case of real-time applications, HTTP responses must also be checked to ensure that the target application is accessible and has not changed drastically such that resources are no longer available at their original locations.

During development, it's advisable to download local copies of documents to be scraped and include them as part of the unit test suite as it's developed. Additionally, the test suite should include two working modes: local and remote. The former case would perform tests on the aforementioned local document copies while the latter would download the documents from the target site in real-time. In the event that any areas of the web scraping application stop functioning as expected, contrasting the results of these two working modes can be very helpful in determining the cause of the issue.

PHPUnit is among the most popular of PHP unit testing frameworks available. See `http://phpunit.de` for more information. Among its many features is the option to output test results to a file in XML format. This feature and others similar to it in both PHPUnit and other unit testing frameworks is very useful in producing results that can be ported to another data medium and made accessible to the web scraping application itself. This facilitates the ability to temporarily restrict or otherwise disable functionality in that application should tests relevant to said functionality fail.

The bottom line is this: debugging a web application is like trying to kill a moving housefly with a pea shooter. It's important to make locating the cause of an issue as easy as possible to minimize the turn-around time required to update the web scraping application to accommodate for it. Test failures should alert developers and lock down any sensitive application areas to prevent erroneous transmission, corruption, or deletion of data.

That's All Folks

Thank you for reading this book. Hopefully you've enjoyed it and learned new things along the way or at least been given food for thought. Ideally, reading this will have far-reaching effects on how you build web scraping applications in the future and it will be one of the books you refer back to time and time again. Beyond this book, what remains to be learned is learned by doing. So, go forth and scrape!

Appendix A

Legality of Web Scraping

The legality of web scraping is a rather complicated question, mainly due to copyright and intellectual property laws. Unfortunately, there is no easy and completely cut-and-dry answer, particularly because these laws can vary between countries. There are, however, a few common points for examination when reviewing a prospective web scraping target.

First, web sites often have documents known as Terms of Service (TOS), Terms or Conditions of Use, or User Agreements (hereafter simply known as TOS documents for the sake of reference). These are generally located in an out-of-the-way location like a link in the site footer or in a Legal Documents or Help section. These types of documents are more common on larger and more well-known web sites. Below are segments of several such documents from web sites that explicitly prohibit web scraping of their content.

- "You specifically agree not to access (or attempt to access) any of the Services through any automated means (including use of scripts or web crawlers)..." – Google Terms of Service, section 5.3 as of 2/14/10
- "You will not collect users' content or information, or otherwise access Facebook, using automated means (such as harvesting bots, robots, spiders, or scrapers) without our permission." – Facebook Statement of Rights and Responsibilities, Safety section as of 2/14/10

- "Amazon grants you a limited license to access and make personal use of this site ... This license does not include ... any use of data mining, robots, or similar data gathering and extraction tools." – Amazon Conditions of Use, LICENSE AND SITE ACCESS section as of 2/14/10
- "You agree that you will not use any robot, spider, scraper or other automated means to access the Sites for any purpose without our express written permission." – eBay User Agreement, Access and Interference section as of 2/14/10
- "... you agree not to: ... access, monitor or copy any content or information of this Website using any robot, spider, scraper or other automated means or any manual process for any purpose without our express written permission; ..." – Expedia, Inc. Web Site Terms, Conditions, and Notices, PROHIBITED ACTIVITIES section as of 2/14/10
- "The foregoing licenses do not include any rights to: ... use any robot, spider, data miner, scraper or other automated means to access the Barnes & Noble.com Site or its systems, the Content or any portion or derivative thereof for any purpose; ..." – Barnes & Noble Terms of Use, Section I LICENSES AND RESTRICTIONS as of 2/14/10

Determining whether or not the web site in question has a TOS document will be the first step. If you find one, look for clauses using language similar to that of the above examples. Also, look for any broad "blanket" clauses of prohibited activities under which web scraping may fall.

If you find a TOS document and it does not expressly forbid web scraping, the next step is to contact representatives who have authority to speak on behalf of the organization that owns the web site. Some organizations may allow web scraping assuming that you secure permission with appropriate authorities beforehand. When obtaining this permission, it is best to obtain a document in writing and on official letterhead that clearly indicates that it originated from the organization in question. This has the greatest chance of mitigating any legal issues that may arise.

If intellectual property-related allegations are brought against an individual as a result of usage of an automated agent or information acquired by one, assuming the individual did not violate any TOS agreement imposed by its owner or related computer use laws, a court decision will likely boil down to whether or not the usage of said information is interpreted as "fair use" with respect to copyright laws in the geographical area in which the alleged offense took place.

Please note that these statements are very general and are not intended to replace the consultation of an attorney. If TOS agreements or lack thereof and communications with the web site owner prove inconclusive, it is highly advisable to seek legal council prior to any attempts being made to launch an automated agent on a web site. This is another reason why web scraping is a less-than-ideal approach to solving the problem of data acquisition and why it should be considered only in the absence of alternatives.

Some sites actually use license agreements to grant open or mildly restricted usage rights for their content. Common licenses to this end include the GNU Free Documentation license and the Creative Commons licenses. In instances where the particular data source being used to acquire data is not relevant, sources that use licenses like these should be preferred over those that do not, as legalities are significantly less likely to become an issue.

The second point of inspection is the legitimacy of the web site as the originating source of the data to be harvested. Even large companies with substantial legal resources, such as Google, have run into issues when their automated agents acquired content from sites illegally syndicating other sites. In some cases, sites will attribute their sources, but in many cases they will not.

For textual content, entering direct quotations that are likely to be unique from the site into major search engines is one method that can help to determine if the site in question originated the data. It may also provide some indication as to whether or not syndicating that data is legal.

For non-textual data, make educated guesses as to keywords that correspond to the subject and try using a search engine specific to that particular data format. Searches like this are not intended to be extensive or definitive indications, but merely a quick way of ruling out an obvious syndication of an original data source.

Appendix B

Multiprocessing

When a PHP script is executed using the CLI SAPI (i.e. from a command line), that instance of execution exists as a process in the local operating system. The Process Control (pcntl) extension makes it possible for PHP scripts to perform what is called a process **fork**. This entails the original PHP process (then called the **parent process**) creating a copy of itself (appropriately called the **child process**) that includes everything from the data associated with the process to the current point of execution following the fork instruction.

Once the fork completes, both processes exist and execute independently of each other. The parent process can fork itself multiple times in succession and retains a limited awareness of its child processes. In particular, it is notified when any given child process terminates.

Because each child process is a copy of its parent process, the number of child processes that can be forked is limited by the hardware resources of the local system. CPU will merely limit the speed at which all child processes can complete. Memory, however, can become a bottleneck if the local system's RAM is exceeded and it has to resort to using swap space for storage.

This restriction creates the desire to fork as many processes as possible to complete a job in parallel without hitting any resource limits. This is generally achieved by using a predetermined cap on the number of processes to fork. Once any given child processes completes, however, it may be desirable to create a new child process in its place to complete another unit of work. An implementation of this is presented in the example below.

```php
<?php
// Maximum number of child processes to fork
$children_max = 10;

// Counter for current number of child processes
$children_count = 0;

// Temporary variable for the status of a terminated child process
$child_status = null;

// Process identifier, initialized so the parent branch executes
$pid = 1;

while (true) {
    if ($pid == -1) {
        trigger_error(’Unable to fork process’, E_USER_ERROR);
    } elseif ($pid) {
        // Parent process

        // Check for more work

        // Break out of the loop if the job is finished

        // Fork another child
        $children_count++;
        $pid = pcntl_fork();

        // Wait to fork more children if the max is reached
        if ($children_count == $children_max) {
            pcntl_wait($child_status);
            $children_count--;
        }
    } else {
        // Child process

        // Perform a unit of work

        exit;
    }
}
?>
```

Process Control Resources
For more on the Process Control extension, see `http://php.net/manual/en/book.pcntl.php`.