CHAPTER 20

Web Services

These days, it seems as if every few months we are told of some new technology that is destined to propel each and every one of us into our own personal utopia. You know, the place where all forms of labor are carried out by highly intelligent machines, where software writes itself, and where we're left to do nothing but lie on the beach and have grapes fed to us by androids? Most recently, the set of technologies collectively referred to as "Web Services" has been crowned as the keeper of this long-awaited promise. And although the verdict is still out as to whether Web Services will live up to the enormous hype that has surrounded them, some very interesting advancements are being made in this arena that have drastically changed the way that we think about both software and data within our newly networked world. This chapter discusses some of the more applicable implementations of Web Services technologies, and shows you how to use PHP to start incorporating them into your Web application development strategy *right now*.

To accomplish this goal without actually turning this chapter into a book unto itself, the discussion that follows isn't intended to offer an in-depth introduction to the general concept of Web Services. Devoting a section of this chapter to the matter simply would do the topic little justice, and in fact would likely do more harm than good. For a comprehensive introduction, please consult any of the many quality print and online resources that are devoted to the topic.

Nonetheless, even if you have no prior experience with or knowledge of Web Services, hopefully you'll find the discussion in this chapter to be quite easy to comprehend. The intention here is to demonstrate the utility of Web Services through numerous practical demonstrations, employing the use of two great PHP-driven third-party class libraries: Magpie and NuSOAP. The SOAP and SimpleXML extensions are also introduced, both of which are new to PHP 5. Specifically, the following topics are discussed:

- **Why Web Services?** For the uninitiated, this section very briefly touches upon the reasons for all of the work behind Web Services, and how they will change the landscape of application development.
- **Real Simple Syndication (RSS)**: The originators of the World Wide Web had little idea that their accomplishments in this area would lead to what is certainly one of the greatest technological leaps in the history of humankind. However, the extraordinary popularity of the medium caused the capabilities of the original mechanisms to be stretched in ways never intended by their creators. As a result, new methods for publishing information over the Web have emerged, and are starting to have as great an impact on the way we retrieve and review data as did their predecessors. One such technology is known as Real Simple Syndication, or RSS. This section introduces RSS, and demonstrates how you can incorporate RSS feeds into your development acumen using a great tool called Magpie.

- **SimpleXML**: New to PHP version 5, the SimpleXML extension offers a new and highly practical methodology for parsing XML. This section introduces this new feature, and offers several practical examples demonstrating its powerful and intuitive capabilities.
- **SOAP**: The SOAP protocol plays an enormously important role in the implementation of Web Services. This section discusses its advantages and, for readers running versions of PHP older than version 5, offers an in-depth look into one of the slickest PHP add-ons around: NuSOAP. In this section, you'll learn how to create PHP-based Web Services clients and servers, as well as integrate a PHP Web Service with a C# client. For those of you running PHP 5 or greater, this section also introduces PHP's SOAP extension, new to version 5.

■ **Note** Several of the examples found throughout this chapter reference the URL `http://www.example.com/`. When testing these examples, you'll need to change this URL to the appropriate location of the Web Service files on your server.

Why Web Services?

The term *computer science* is surely an oxymoron, because for those of us in the trenches, there is little doubt that our daily travails often sway more toward the path of artisan than of scientist. This is evident in the way that software has historically been designed. Although the typical developer generally adheres to a loosely defined set of practices and tools, much as an artist generally works with a particular medium and style, he tends to create software in the way he sees most fit. As such, it doesn't come as a surprise that although many programs resemble one another, they rarely follow the same set of rigorous principles that scientists might employ when carrying out similar experiments. Numerous deficiencies arise as a result of this refusal to follow generally accepted programming principles, with software being developed at a cost of maintainability, scalability, extensibility, and, perhaps most notably, interoperability.

This problem of interoperability has become even more pronounced over the past few years, given the incredible opportunities for cooperation that the Internet has opened up to businesses around the world. However, fully exploiting an online business partnership often, if not always, involves some level of system integration. Therein lies the problem: If the system designers never consider the possibility that they might one day need to tightly integrate their application with another, how will they ever really be able to exploit the Internet to its fullest advantage? Indeed, this has been a subject of considerable discussion almost from the onset of this new electronic age.

Web Services technology is today's most promising solution to the interoperability problem. Rather than offer up yet another interpretation of the definition of Web Services, here's an excellent interpretation provided in the W3C's "Web Services Architecture" document, currently a working draft (`http://www.w3.org/TR/ws-arch/`):

> *A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

Some of these terms may be alien to the newcomer; not to worry, because they're introduced later in the chapter. What is important to keep in mind is that Web Services open up endless possibilities to the enterprise, a sampling of which follows:

- **Software as a service**: Imagine building an e-commerce application that requires a means for converting currency among various exchange rates. However, rather than take it upon yourself to devise some means for automatically scraping the Federal Reserve Bank's Web page (`http://www.federalreserve.gov/releases/`) for the daily released rate, you instead plug in to its (hypothetical) Web Service for retrieving these values. The result is far more readable code, with much less chance for error from presentational changes on the Web page.

- **Significantly lessened Enterprise Application Integration (EAI) horrors**: Developers currently are forced to devote enormous amounts of time to hacking together one-off solutions to integrate disparate applications. Contrast this with connecting two Web Service–enabled applications, in which the process is highly standardized and reusable no matter the language.

- **Write once, reuse everywhere**: Because Web Services offer platform-agnostic interfaces to exposed application methods, they can be simultaneously used by applications running on disparate operating systems. For example, a Web Service running on an e-commerce server might be used to keep the CEO abreast of inventory numbers both via a Windows-based application and via a Perl script running on a Linux server that generates daily e-mails to the suppliers.

- **Ubiquitous access**: Because Web Services typically travel over the HTTP protocol, firewalls can be bypassed because port 80 (and 443 for HTTPS) traffic is almost always allowed. Although debate is currently underway as to whether this is really prudent, for the moment it is indeed an appealing solution to the often difficult affair of firewall penetration.

Such capabilities are tantalizing to the developer. Believe it or not, as is demonstrated throughout this chapter, you can actually begin taking advantage of Web Services right now.

Ultimately, only one metric will determine the success of Web Services: acceptance. Interestingly, several global companies have already made quite a stir by offering Web Services application programming interfaces (APIs) to their treasured data stores. Among the most interesting offers include those provided by the online superstore Amazon.com (`http://www.amazon.com/`), the famed Google search engine (`http://www.google.com/`), and Microsoft (`http://www.microsoft.com/`), stirring the imagination of the programming industry with their freely available standards-based Web Services. Since their respective releases, all three implementations have sparked the imaginations of programmers worldwide, who have gained valuable experience working with a well-designed Web Services architecture plugged into an enormous amount of data. Given such high-profile deployments, it isn't hard to imagine that other companies will soon follow.

Later in this chapter we'll explore the Google Web Services API. However, you're invited to take some time to learn more about all three APIs if you don't want to wait:

```
http://www.amazon.com/webservices/
http://www.google.com/apis/
http://msdn.microsoft.com/mappoint/
```

Real Simple Syndication

Given that the entire concept of Web Services largely sprung out of the notion that XML- and HTTP-driven applications would be harnessed to power the next generation of business-to-business applications, it's rather ironic that the first widespread implementation of the Web Services technologies happened on the end-user level. *Real Simple Syndication (RSS)* solves a number of problems that both Web developers and Web users have faced for years.

On the end-user level, all of us can relate to the considerable amount of time consumed by our daily surfing ritual. Most people have a stable of Web sites that they visit on a regular basis, and in some cases, several times daily. For each site, the process is almost identical: Visit the URL, weave around a sea of advertisements, navigate to the section of interest, and finally actually read the news story. Repeat this process numerous times, and the next thing you know, a fair amount of time has passed. Furthermore, given the highly tedious process, it's easy to neglect a particular information resource for days, potentially missing something of interest. In short, leave the process to a human, and something is bound to get screwed up.

Developers face an entirely different set of problems. Once upon a time, attracting users to your Web site involved spending enormous amounts of money on prime-time commercials and magazine layouts, and throwing lavish holiday galas. Then the novelty wore out (and the cash disappeared), and those in charge of the Web sites were forced to actually produce something substantial for their site visitors. Furthermore, they had to do so while working within the constraints of bandwidth limitations, the myriad of Web-enabled devices that sprung up, and an increasingly finicky (and time-pressed) user. Enter RSS.

RSS offers a formalized means for encapsulating a Web site's content within an XML-based structure, known as a *feed*. It's based on the premise that most site information shares a similar format, regardless of topic. For example, although sports, weather, and theater are all vastly dissimilar topics, the news items published under each would share a very similar structure, including a title, author, publication date, URL, and description. A typical RSS feed embodies all such attributes, and often much more, forcing an adherence to a presentation-agnostic format that can in turn be retrieved, parsed, and formatted in any means acceptable to the end user, without actually having to visit the syndicating Web site. With just the feed's URL, the user can store it, along with others if he likes, into a tool that is capable of retrieving and parsing the feed, allowing the user to do as he pleases with the information. Working in this fashion, you can use RSS feeds to do the following:

- Browse the rendered feeds using a standalone RSS aggregator application. Examples of popular aggregators include RSS Bandit (`http://www.rssbandit.org/`), Straw (`http://www.nongnu.org/straw/`), and SharpReader (`http://www.sharpreader.net/`). A screenshot of the SharpReader application is shown in Figure 20-1.
- Subscribe to any of the numerous Web-based RSS aggregators, and view the feeds via a Web browser. Examples of popular online aggregators include Feedster (`http://www.feedster.com/`), NewsIsFree (`http://www.newsisfree.com/`), and Bloglines (`http://www.bloglines.com/`).
- Retrieve and republish the syndicated feed as part of a third-party Web application or service. Moreover Technologies (`http://www.moreover.com/`) is an excellent example of such a service. Another popular use involves simply incorporating a rendered feed into your own Web site, taking the opportunity to provide additional third-party content to your readers. Later in this section, you'll learn how this is accomplished using the Magpie RSS class library.

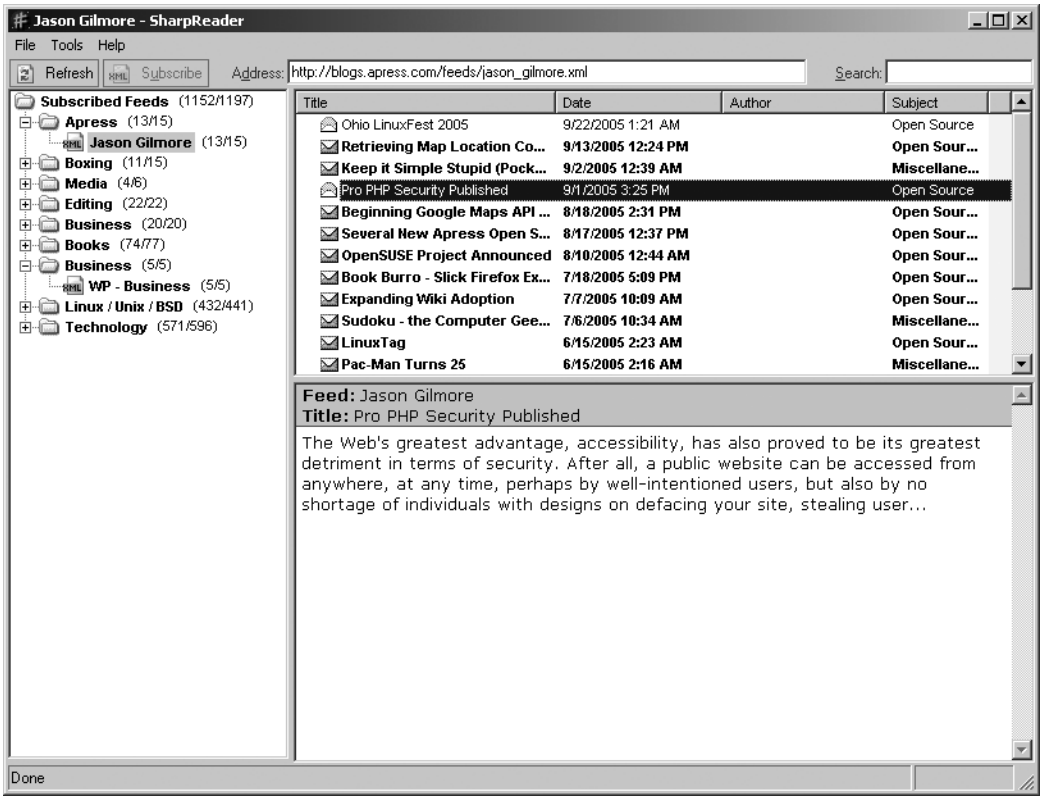

Figure 20-1. *The SharpReader interface, created by Luke Hutteman*

WHO'S PUBLISHING RSS FEEDS?

Believe it or not, RSS has actually officially been around since early 1999, and in previous incarnations since 1996. However, like many emerging technologies, it remained a niche tool of the "techie" community, at least until recently. The emergence and growing popularity of news aggregation sites and tools has prompted an explosion in terms of the creation and publication of RSS feeds around the Web. These days, you can find RSS feeds just about everywhere, including within these prominent organizations:

- **Yahoo! News**: `http://news.yahoo.com/rss/`
- **Christian Science Monitor**: `http://www.csmonitor.com/rss/`
- **CNET News.com**: `http://www.news.com/`
- **The BBC**: `http://www.bbc.co.uk/syndication/`
- **Wired.com**: `http://www.wired.com/news/rss/`

Given the adoption of RSS in such circles, it isn't really a surprise that we're hearing so much about this great technology these days.

RSS Syntax

If you're not familiar with the general syntax of an RSS feed, Listing 20-1 offers an example, which will be used as input for the scripts that follow. Although a discussion of RSS syntax specifics is beyond the scope of this book, you'll nonetheless find the structure and tags to be quite intuitive (after all, that's why they call it "Real Simple Syndication").

Listing 20-1. *A Sample RSS Feed (blog.xml)*

```
<?xml version="1.0" encoding="iso-8859-1"?>
   <rss version="2.0">
   <channel>
      <title>Jason Gilmore</title>
      <link>http://blogs.apress.com/</link>

   <item>
      <title>Ohio LinuxFest 2005</title>
      <link>http://blogs.apress.com/?p=639#more-639</link>
      <description>The annual Ohio LinuxFest 2005 conference is rapidly
      approaching, taking place at the Columbus Convention Center on October 1,
      2005...</description>
   </item>

   <item>
      <title>Retrieving Map Location Coordinates</title>
      <link>http://blogs.apress.com/?p=634#more-634</link>
      <description>In the first installment of a three-part series for
      Developer.com, you learned how to take advantage of Google&amp;apos;s amazing
      mapping API...</description>
   </item>

   <item>
      <title>Pro PHP Security Published</title>
      <link>http://blogs.apress.com/?p=626#more-626</link>
      <description>The Web&amp;apos;s greatest advantage, accessibility, has also
      proved to be its greatest detriment in terms of security...</description>
   </item>
   </channel>
   </rss>
```

Note that this example is somewhat stripped down, as there are numerous other elements found in an RSS 2.0 file such as the update period, language, and creator. However, for the purposes of the examples found in this chapter, it makes sense to remove those components that have little bearing on instruction. To view an example of a complete feed, see `http://blogs.apress.com/wp-rss.php`.

Now that you're a bit more familiar with the purpose and advantages of RSS, you'll next learn how to use PHP to incorporate RSS into your own development strategy. Although there are numerous RSS tools written for the PHP language, one in particular offers an amazingly effective solution for retrieving, parsing, and displaying feeds: MagpieRSS.

MagpieRSS

MagpieRSS (Magpie for short) is a powerful RSS parser written in PHP by Kellan Elliott-McCrea. It's freely available for download via `http://magpierss.sourceforge.net/` and is distributed under the GPL license. Magpie offers developers an amazingly practical and easy means for retrieving and rendering RSS feeds, as you'll soon see. In addition, Magpie offers users a number of cool features, including:

- **Simplicity**: Magpie gets the job done with a minimum of effort by the developer. For example, typing a few lines of code is all it takes to begin retrieving, parsing, and converting RSS feeds into an easily readable format.
- **Nonvalidating**: If the feed is well formed, Magpie will successfully parse it. This means that it supports all tag sets found within the various RSS versions, as well as your own custom tags.
- **Bandwidth-friendly**: By default, Magpie caches feed contents for 60 minutes, cutting down on use of unnecessary bandwidth. You're free to modify the default to fit caching preferences on a per-feed basis (which is demonstrated later). If retrieval is requested after the cache has expired, Magpie will retrieve the feed only if it has been changed (by checking the Last-modified and ETag headers provided by the Web server). In addition, Magpie recognizes HTTP's GZIP content-negotiation ability when supported.

Installing Magpie

Like most PHP classes, installing Magpie is as simple as placing the relevant files within a directory that can later be referenced from a PHP script. The instructions for doing so follow:

1. Download Magpie from `http://magpierss.sourceforge.net/`.
2. Extract the package contents to a location convenient for inclusion from a PHP script. For instance, consider placing third-party classes within an aptly named directory located within the `PHP_INSTALL_DIR/includes/` directory. Note that you can forego the hassle of typing out the complete path to the Magpie directory by adding its location to the `include_path` directive found in the `php.ini` file.
3. Include the Magpie class (`rss_fetch.inc`) within your script:

```
require('magpie/rssfetch.php');
```

That's it! You're ready to begin using Magpie.

How Magpie Parses a Feed

Magpie parses a feed by placing it into an object consisting of four fields: `channel`, `image`, `items`, and `textinput`. In turn, `channel` is an array of associative arrays, while the remaining three are associative arrays. The following script retrieves the `blog.xml` feed, outputting it using the `print_r()` statement:

```
<?php
   require("magpie/rss_fetch.inc");
   $url = "http://localhost/book/20/blog.xml";
   $rss = fetch_rss($url);
   print_r($rss);
?>
```

This returns the following output (containing only one item, for readability):

```
MagpieRSS Object (
   [parser] => Resource id #9
   [current_item] => Array ( )
   [items] => Array (

      [0] => Array (
      [title] => Ohio LinuxFest 2005
      [link] => http://blogs.apress.com/?p=639#more-639</
      [description] => The annual Ohio LinuxFest 2005 conference is rapidly
                       approaching, taking place at the Columbus Convention
                       Center on October 1, 2005...
      [summary] => The annual Ohio LinuxFest 2005 conference is rapidly
                   approaching, taking place at the Columbus Convention Center on
                   October 1, 2005...
      )

      [1] => Array (
      [title] => Retrieving Map Location Coordinates
      [link] => http://blogs.apress.com/?p=634#more-634
      [description] => In the first installment of a three-part series for
                       Developer.com, you learned how to take advantage of
                       Google's amazing mapping API...
      [summary] => In the first installment of a three-part series for
                   Developer.com, you learned how to take advantage of Google's
                   amazing mapping API...
      )

      [2] => Array (
      [title] => Pro PHP Security Published
      [link] => http://blogs.apress.com/?p=626#more-626
      [description] => The Web's greatest advantage, accessibility, has also
                       proved to be its greatest detriment in terms of
                       security...
      [summary] => The Web's greatest advantage, accessibility, has also proved
                   to be its greatest detriment in terms of security... )
      )
```

```
[channel] => Array (
   [title] => Jason Gilmore
   [link] => http://blogs.apress.com/
   [tagline] =>
)

[textinput] => Array ( )
[image] => Array ( )
[feed_type] => RSS
[feed_version] => 2.0
[encoding] => ISO-8859-1
[_source_encoding] =>
[ERROR] =>
[WARNING] =>
[_CONTENT_CONSTRUCTS] => Array (
   [0] => content [1] => summary [2] => info [3] => title
   [4] => tagline [5] => copyright )
[_KNOWN_ENCODINGS] => Array (
   [0] => UTF-8
   [1] => US-ASCII
   [2] => ISO-8859-1 )
   [stack] => Array ( )
   [inchannel] => [initem] => [incontent] => [intextinput] =>
   [inimage] => [current_field] => [current_namespace] =>
   [last_modified] => Mon, 26 Sep 2005 19:43:48 GMT
   [etag] => "50e4-413-fa6a7a9f"
)
```

Note the presence of the four object attributes in each element of the `items` array. While the `summary` and `description` attributes may seem redundant, this information is replicated because Magpie supports both RSS and an alternative syndication format known as Atom (`http://www.intertwingly.net/wiki/pie/FrontPage`), which uses the attribute `Summary` instead of `Description`. When retrieving RSS values using the Magpie methods, which are introduced soon, such redundancy will be neither apparent nor relevant. Following `items` is the `channel` array, which contains information pertinent to the feed in general, including the feed `title`, `domain`, and other attributes not shown in the example feed. Finally, information pertinent to the feed's technical aspects is offered, including the encoding type, date of last modification, and RSS version. Of course, for most users, only the information found in the `items` and `channel` arrays is of interest, so don't worry too much about the attributes that aren't particularly familiar.

The following examples demonstrate how the data is peeled from this object and presented in various fashions.

Retrieving an RSS Feed

Based on your knowledge of Magpie's parsing behavior, rendering the feed components should be trivial. Listing 20-2 demonstrates how easy it is to render a retrieved feed within a standard browser.

Listing 20-2. *Rendering an RSS Feed with Magpie*

```
<?php
require("magpie/rss_fetch.inc");

// RSS feed location?
$url = "http://localhost/book/20/blog.xml";
// Retrieve the feed
$rss = fetch_rss($url);

// Format the feed for the browser
$feedTitle = $rss->channel['title'];
echo "Latest News from <strong>$feedTitle</strong>";
foreach ($rss->items as $item) {
   $link = $item['link'];
   $title = $item['title'];
   // Not all items necessarily have a description, so test for one.
   $description = isset($item['description']) ? $item['description'] : "";
   echo "<p><a href=\"$link\">$title</a><br />$description</p>";
}

?>
```

Note that Magpie does all of the hard work of parsing the RSS document, placing the data into easily referenced arrays. Figure 20-2 shows the fruits of this script.

Figure 20-2. *Rendering an RSS feed within the browser*

As you can see from Figure 20-2, each feed item is formatted with the title linking to the complete entry. So, for example, following the Ohio LinuxFest 2005 link will take the user to `http://ablog.apress.com/?p=639#more-639`.

Aggregating Feeds

Of course, chances are you're going to want to aggregate multiple feeds and devise some means for viewing them simultaneously. To do so, you can simply modify Listing 20-2, passing in an array of feeds. A bit of CSS may also be added to shrink the space required for output. Listing 20-3 shows the rendered version.

Listing 20-3. *Aggregating Multiple Feeds with Magpie*

```
<style><!--
p { font: 11px arial,sans-serif; margin-top: 2px;}
//-->
</style>

<?php
require("magpie/rss_fetch.inc");

// Compile array of feeds
$feeds = array(
"http://localhost/book/20/blog.xml",
"http://news.com.com/2547-1_3-0-5.xml",
"http://slashdot.org/slashdot.rdf");

// Iterate through each feed
foreach ($feeds as $feed) {

   // Retrieve the feed
   $rss = fetch_rss($feed);

   // Format the feed for the browser
   $feedTitle = $rss->channel['title'];
   echo "<p><strong>$feedTitle</strong><br />";

   foreach ($rss->items as $item) {
      $link = $item['link'];
      $title = $item['title'];
      $description = isset($item['description']) ? $item['description'].
                     "<br />" : "";
      echo "<a href=\"$link\">$title</a><br />$description";
   }
   echo "</p>";

}

?>
```

Figure 20-3 depicts the output based on these three feeds.

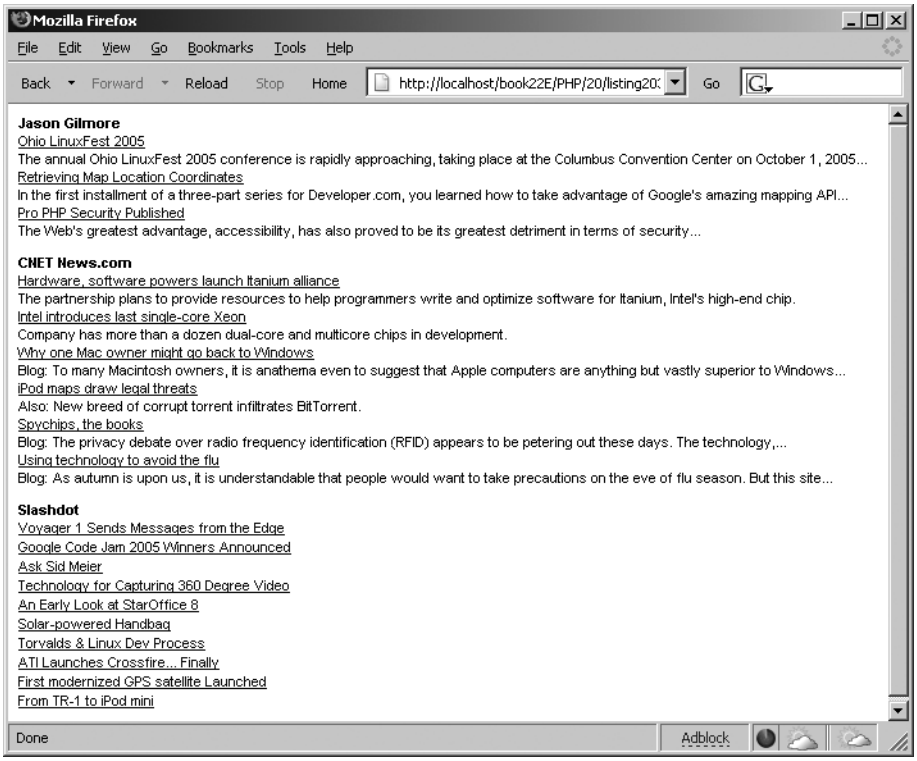

Figure 20-3. *Aggregating feeds*

Although the use of a static array for containing feeds certainly works, it might be more practical to maintain them within a database table, or at the very least a text file. It really all depends upon the number of feeds you'll be using, and how often you intend on managing the feeds themselves.

Limiting the Number of Displayed Headlines

Some Web site developers are so keen on RSS that they wind up dumping quite a bit of information into their published feeds. However, you might be interested in viewing only the most recent items, and ignoring the rest. Because Magpie relies heavily on standard PHP language features such as arrays and objects for managing RSS data, limiting the number of headlines is trivial, because you can call upon one of PHP's default array functions for the task. The function `array_slice()` should do the job quite nicely. For example, suppose you want to limit total headlines displayed for a given feed to three. You can use `array_slice()` to truncate it prior to iteration, like so:

```
$rss->items = array_slice($rss->items, 0, 3);
```

Revising the previous script to include this call results in output similar to that shown in Figure 20-4.

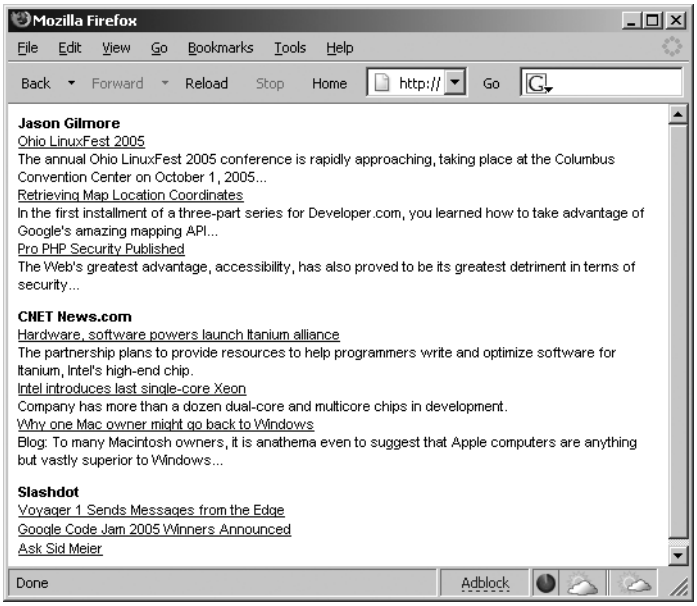

Figure 20-4. *Limiting the number of headlines for each feed*

Caching Feeds

One final topic to discuss regarding Magpie is its caching feature. By default, Magpie caches feeds for 60 minutes, on the premise that the typical feed will likely not be updated more than once per hour. Therefore, even if you constantly attempt to retrieve the same feeds, say once every 5 minutes, any updates will not appear until the feed cache is at least 60 minutes old. However, some feeds are published more than once an hour, or the feed might be used to publish somewhat more pressing information. (RSS feeds don't necessarily have to be used for browsing news headlines; you could use them to publish information about system health, logs, or any other data that could be adapted to its structure. It's also possible to extend RSS as of version 2.0, but this matter is beyond the scope of this book.) In such cases, you may want to consider modifying the default behavior.

To completely disable caching, disable the constant `MAGPIE_CACHE_ON`, like so:

```
define('MAGPIE_CACHE_ON', 0);
```

To change the default cache time (measured in seconds), you can modify the constant `MAGPIE_CACHE_AGE`, like so:

```
define('MAGPIE_CACHE_AGE',1800);
```

Finally, you can opt to display an error instead of a cached feed in the case that the fetch fails, by enabling the constant `MAGPIE_CACHE_FRESH_ONLY`:

```
define('MAGPIE_CACHE_FRESH_ONLY', 1)
```

You can also change the default cache location (by default, the same location as the executing script), by modifying the MAGPIE_CACHE_DIR constant:

```
define('MAGPIE_CACHE_DIR', '/tmp/magpiecache/');
```

SimpleXML

Everyone agrees that XML signifies an enormous leap forward in data management and application interoperability. Yet how come it's so darned hard to parse? Although powerful parsing solutions are readily available, DOM, SAX, and XSLT to name a few, each presents a learning curve that is just steep enough to cause considerable gnashing of the teeth among those users interested in taking advantage of XML's practicalities without an impractical time investment. Leave it to an enterprising PHP developer (namely, Sterling Hughes) to devise a graceful solution. SimpleXML offers users a very practical and intuitive methodology for processing XML structures, and is enabled by default as of PHP 5. Parsing even complex structures becomes a trivial task, accomplished by loading the document into an object and then accessing the nodes using field references, as you would in typical object-oriented fashion.

The XML document displayed in Listing 20-4 is used to illustrate the examples offered in this section.

Listing 20-4. *A Simple XML Document*

```
<?xml version="1.0" standalone="yes"?>
<library>
   <book>
      <title>Pride and Prejudice</title>
      <author gender="female">Jane Austen</author>
      <description>Jane Austen's most popular work.</description>
   </book>
   <book>
      <title>The Conformist</title>
      <author gender="male">Alberto Moravia</author>
      <description>Alberto Moravia's classic psychological novel.</description>
   </book>
   <book>
      <title>The Sun Also Rises</title>
      <author gender="male">Ernest Hemingway</author>
      <description>The masterpiece that launched Hemingway's
      career.</description>
   </book>
</library>
```

SimpleXML Functions

A number of SimpleXML functions are available for loading and parsing the XML document. Those functions are introduced in this section, along with several accompanying examples.

■ **Note** To take advantage of SimpleXML, you need to disable the PHP directive `zend.ze1_compatibility_mode`.

simplexml_load_file()

`object simplexml_load_file (string *filename*)`

This function loads an XML file specified by `filename` into an object. If a problem is encountered loading the file, `FALSE` is returned. Consider an example:

```
<?php
   $xml = simplexml_load_file("books.xml");
   var_dump($xml);
?>
```

This code returns:

```
object(simplexml_element)#1 (1) {
["book"]=> array(3) {
   [0]=> object(simplexml_element)#2 (3) {
      ["title"]=> string(19) "Pride and Prejudice"
      ["author"]=> string(11) "Jane Austen"
      ["description"]=> string(32) "Jane Austen's most popular work."
   }
   [1]=> object(simplexml_element)#3 (3) {
      ["title"]=> string(14) "The Conformist"
      ["author"]=> string(15) "Alberto Moravia"
      ["description"]=> string(46) "Alberto Moravia's classic
                                    psychological novel."
   }
   [2]=> object(simplexml_element)#4 (3) {
      ["title"]=> string(18) "The Sun Also Rises"
      ["author"]=> string(16) "Ernest Hemingway"
      ["description"]=> string(56) "The masterpiece that launched
                                    Hemingway's career."
   }
   }
}
```

Note that dumping the XML will not cause the attributes to show. To view attributes, you need to use the `attributes()` method, introduced later in this section.

simplexml_load_string()

```
object simplexml_load_string (string data)
```

If the XML document is stored in a variable, you can use the `simplexml_load_string()` function to read it into the object. This function is identical in purpose to `simplexml_load_file()`, except that the lone input parameter is expected in the form of a string rather than a file name.

simplexml_import_dom()

```
object simplexml_import_dom (domNode node)
```

The Document Object Model (DOM) is a W3C specification that offers a standardized API for creating an XML document, and subsequently navigating, adding, modifying, and deleting its elements. PHP provides an extension capable of managing XML documents using this standard, titled the DOM XML extension. You can use this function to convert a node of a DOM document into a SimpleXML node, subsequently exploiting use of the SimpleXML functions to manipulate that node.

SimpleXML Methods

Once an XML document has been loaded into an object, several methods are at your disposal. Presently, four methods are available, each of which is introduced in this section.

attributes()

```
object simplexml_element->attributes()
```

XML attributes provide additional information about an XML element. In the sample XML document in Listing 20-4, only the `author` node possesses an attribute, namely `gender`, used to offer information about the author's gender. You can use the `attributes()` method to retrieve these attributes. For example, suppose you want to retrieve the gender of each author:

```
<?php
   $xml = simplexml_load_file("books.xml");
   foreach($xml->book as $book) {
      echo $book->author." is ".$book->author->attributes().".<br />";
   }
?>
```

This example returns:

```
Jane Austen is female.
Alberto Moravia is male.
Ernest Hemingway is male.
```

You can also directly reference a particular book author's gender. For example, suppose you want to determine the gender of the author of the second book in the XML document:

```
echo $xml->book[2]->author->attributes();
```

This example returns:

```
male
```

Often a node possesses more than one attribute. For example, suppose the `author` node looks like this:

```
<author gender="female" age="20">Jane Austen</author>
```

It's easy to output the attributes with a `for` loop:

```
foreach($xml->book[0]->author->attributes() AS $a => $b) {
   echo "$a = $b <br />";
}
```

This example returns:

```
gender = female
age = 20
```

asXML()

```
string simplexml_element->asXML()
```

This method returns a well-formed XML 1.0 string based on the SimpleXML object. An example follows:

```
<?php
   $xml = simplexml_load_file("books.xml");
   echo htmlspecialchars($xml->asXML());
?>
```

This example returns the original XML document, except that the newline characters have been removed, and the characters have been converted to their corresponding HTML entities.

children()

```
object simplexml_element->children()
```

Often, you might be interested in only a particular node's children. Using the `children()` method, retrieving them becomes a trivial affair. Suppose for example that the `books.xml` document was modified so that each book included a cast of characters. The Hemingway book might look like the following:

```
   <book>
      <title>The Sun Also Rises</title>
      <author gender="male">Ernest Hemingway</author>
      <description>The masterpiece that launched Hemingway's
      career.</description>
      <cast>
         <character>Jake Barnes</character>
         <character>Lady Brett Ashley</character>
         <character>Robert Cohn</character>
         <character>Mike Campbell</character>
      </cast>
   </book>
```

Using the `children()` method, you can easily retrieve the characters:

```
<?php
   $xml = simplexml_load_file("books.xml");
   foreach($xml->book[2]->cast->children() AS $character) {
      echo "$character<br />";
   }
?>
```

This example returns:

```
Jake Barnes
Lady Brett Ashley
Robert Cohn
Mike Campbell
```

xpath()

```
array simplexml_element->xpath (string path)
```

XPath is a W3C standard that offers an intuitive, path-based syntax for identifying XML nodes. For example, referring to the `books.xml` document, you could retrieve all `author` nodes using the expression `/library/book/author`. XPath also offers a set of functions for selectively retrieving nodes based on value.

Suppose you want to retrieve all authors found in the `books.xml` document:

```
<?php
   $xml = simplexml_load_file("books.xml");
   $authors = $xml->xpath("/library/book/author");
   foreach($authors AS $author) {
      echo "$author<br />";
   }
?>
```

This example returns:

```
Jane Austen
Alberto Moravia
Ernest Hemingway
```

You can also use XPath functions to selectively retrieve a node and its children based on a particular value. For example, suppose you want to retrieve all `book` titles where the author is named "Ernest Hemingway":

```
<?php
   $xml = simplexml_load_file("books.xml");
   $book = $xml->xpath("/library/book[author='Ernest Hemingway']");
   echo $book[0]->title;
?>
```

This example returns:

```
The Sun Also Rises
```

SOAP

The postal service is amazingly effective at transferring a package from party A to party B, but its only concern is ensuring the safe and timely transmission. The postal service is oblivious to the nature of the transaction, provided that it is in accordance with the postal service's terms of service. As a result, a letter written in English might be sent to a fisherman in China, and that letter will indeed arrive without issue, but the recipient would probably not understand a word of it. The same holds true if the fisherman were to send a letter to you written in his native language; chances are you wouldn't even know where to begin.

This isn't unlike what might occur if two applications attempt to talk to each other across a network. Although they could employ messaging protocols like HTTP and SMTP in much the same way that we make use of the postal service, it's quite unlikely one will be able to say anything of discernible interest to the other. However, if the parties agree to send data using the same messaging language, and both are capable of understanding messages sent to them, then the dilemma is resolved. Granted, both parties might go about their own way of interpreting that language (more about that in a bit), but nonetheless the commonality is all that's needed to ensure comprehension. Web Services often employ the use of something called SOAP as that common language. Here's the formalized definition of SOAP, as stated within the SOAP 1.2 specification (`http://www.w3.org/TR/SOAP12-part1/`):

> *SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.*

Keep in mind that SOAP is only responsible for defining the construct used for the exchange of messages; it does not define the protocol used to transport that message, nor does it describe the features or purpose of the Web Service used to send or receive that message. This means that you could conceivably use SOAP over any protocol, and in fact could route a SOAP message over numerous protocols during the course of transmission. A sample SOAP message is offered in Listing 20-5 (formatted for readability).

Listing 20-5. *A Sample SOAP Message*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
   <SOAP-ENV:Envelope SOAP
               ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:si="http://soapinterop.org/xsd">
      <SOAP-ENV:Body>
         <getRandQuoteResponse>
            <return xsi:type="xsd:string">
            "My main objective is to be professional but to kill him.",
                Mike Tyson (2002)
            </return>
        </getRandQuoteResponse>
     </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

If you're new to SOAP, it would certainly behoove you to take some time to become familiar with the protocol. A simple Web search will turn up a considerable amount of information pertinent to this pillar of Web Services. Regardless, you should be able to follow along with the ensuing discussion quite easily, because the first SOAP-related project introduced, NuSOAP, does a fantastic job of taking care of most of the dirty work pertinent to the assembly, parsing, submission, and retrieval of SOAP messages. Following the NuSOAP discussion, PHP 5's new SOAP extension is introduced, showing you how you can create both SOAP clients and servers using native language functionality.

NuSOAP

NuSOAP is a powerful group of PHP classes that makes the process of consuming and creating SOAP messages trivial. Written by Dietrich Ayala, NuSOAP works seamlessly with many of the most popular SOAP server implementations, and is released under the LGPL. NuSOAP offers a bevy of impressive features, including:

- **Simplicity**: NuSOAP's object-oriented approach hides many of the details pertinent to the SOAP message assembling, parsing, submission, and reception, allowing the user to concentrate on the application itself.

- **WSDL generation and importing**: NuSOAP will generate a WSDL document corresponding to a published Web Service and can import a WSDL reference for use within a NuSOAP client.
- **A proxy class**: NuSOAP can generate a proxy class that allows for the remote methods to be called as if they were local.
- **HTTP proxying**: For varying reasons (security and auditing are two), some clients are forced to delegate a request to an HTTP proxy, which in turn performs the request on the client's behalf. That said, any SOAP request would need to pass through this proxy rather than directly query the service server. NuSOAP offers basic support for specifying this proxy server.
- **SSL**: NuSOAP supports secure communication via SSL if the CURL extension is made available via PHP.

All of these features are discussed in further detail throughout this section. For starters, however, you need to install NuSOAP. This simple process is introduced next.

■ **Note** NuSOAP was originally known as SOAPx4, and in fact is a rewrite of the original project. The name was changed in accordance with an agreement by the project author (Dietrich Ayala) and the company NuSphere, which had at one point sponsored development.

Installing NuSOAP

Installing NuSOAP is really a trivial affair, done in three steps:

1. Download the latest stable distribution from `http://dietrich.ganx4.com/nusoap/`.
2. Extract the package contents to a location convenient for inclusion from a PHP script. Consider placing third-party classes within an aptly named directory located within the `PHP_INSTALL_DIR/includes/` directory—this is for convenience reasons only, and isn't a requirement.
3. Include the NuSOAP class (`nusoap.php`) within your script:

```
require('nusoap/nusoap.php');
```

That's it! You're ready to begin using NuSOAP.

■ **Caution** At the time of writing, there was a naming conflict between the NuSOAP class and that found in PHP 5's native SOAP extension (introduced later in this chapter). While the intention of introducing NuSOAP is to offer those readers not yet running PHP 5 the opportunity to take advantage of SOAP-driven Web services, if for some reason you prefer to use NuSOAP over the SOAP extension, you'll need to disable the native extension.

Consuming a Web Service

Rather than go through the motions of creating a useless "Hello World" type of example, it seems more practical to create a client that actually consumes a live, real-world Web Service. As mentioned earlier in the chapter, several large organizations have already started offering public Web Services, including Google, Yahoo!, and Microsoft. The particularly compelling Google Web Service provides a solution for searching the Web via its databases without having to actually visit the Web site. For example, you could use the Web Service in conjunction with any SOAP-capable language (PHP, C#, Perl, or Python, to name a few) to build a custom interface for searching the site, be it the Web, desktop, or command line. However, numerous other interesting features are available to developers, such as the ability to take advantage of Google's amazing spell-checker (which appears at the top of any search results page if the engine thinks that you potentially misspelled a search term).

The next several examples take advantage of Google's Web Service, demonstrating both NuSOAP's capabilities and a number of interesting features offered by this Web Service. Before you can execute these examples, however, you need to go to the Google Web Service site (`http://www.google.com/apis/`) and obtain a license key by registering for a free account. It only takes a moment to do, so go ahead and take care of that now.

You also need to download the developer's kit (available via the aforementioned URL), because the WSDL file is bundled into it. As you've done for previous third-party packages in this chapter, place the unzipped package in a location where the WSDL file is easily accessible by a PHP script, or just copy the WSDL file into the same directory as the script, because that's the only file you'll need from this package.

Once you have completed these two steps, proceed to the next section.

■ **Caution** At present, Google's Web Service is limited to 1,000 queries per day. So while it's great for experimentation or personal use, don't plan to integrate it into your corporate Web site anytime soon. Also, be sure to read through the API terms of service if you plan to use the Web Service in any way: `http://www.google.com/apis/api_terms.html`.

Listing 20-6 offers the first example, which uses Google's spell-checker method, `doSpellingSuggestion()`, to offer suggestions for the misspelled word "fireplace."

Listing 20-6. *Consuming Google's Web Service*

```
<?php

   require("nusoap/nusoap.php");

   // Insert your Google API key
   $key = 'INSERT YOUR KEY HERE';

   // Point to a WSDL file
   $wsdl = "googleapi/GoogleSearch.wsdl";
```

```
    // Create a new soapclient object
    $client = new soapclient($wsdl, 'wsdl');

    // Suppose user enters keyword via Web form (would be via $_POST)
    $keyword = "freplace";

    // Which parameters should be passed to the doSpellingSuggestion() method?
    $input = array('phrase' => $keyword, 'key' => $key);

    // Call the doSpellingSuggestion() method
    $suggestion = $client->call('doSpellingSuggestion', $input);

    // Prompt user to consider searching using suggested term
    echo "Supplied search term not found. Perhaps you meant
          <a href=''>$suggestion</a>?";

?>
```

Executing this example produces the following output:

```
Supplied search term not found. Perhaps you meant <a href=''>fireplace</a>?
```

Of course, by itself this example isn't particularly useful. However, it would be trivial to execute Google's `doSpellingSuggestion()` method should an attempt to search your internal Web site produce zero results. The empty link found in the output could be completed to take the user back to your search engine, this time automatically inputting the suggested keyword.

You may be wondering how the array keys and method name were determined. After all, you can't just make up these names. You can determine this in either of two ways: review the WSDL file, which breaks down each method and its corresponding parameters, or append `?wsdl` to the end of the service URL for NuSOAP-created services.

Creating a Method Proxy

You can also access the Web Service's methods directly, as if the service were a local class library. This is done by creating a proxy via the `getProxy()` method. Listing 20-6 has been revised to do exactly this. Listing 20-7 offers the revised script.

Listing 20-7. *Using NuSOAP's Proxy Class*

```
<?php

    require("nusoap/nusoap.php");

    // Insert your Google API key
    $key = 'INSERT YOUR KEY HERE';
```

```
    // Point to the WSDL file
    $wsdl = "googleapi/GoogleSearch.wsdl";

    // Create a new soapclient object
    $client = new soapclient($wsdl, 'wsdl');

    // Create a proxy so you can call the Google methods directly
    $proxy = $client->getProxy();

    // Suppose user enters keyword via Web form (would be via $_POST)
    $keyword = "freplace";

    // Pass keyword to doSpellingSuggestion() method.
    $suggestion = $proxy->doSpellingSuggestion($key, $keyword);

    // Prompt user to consider searching using suggested term
    echo "Supplied search term not found. Perhaps you
          meant <a href=''>$suggestion</a>?";

?>
```

Executing this example produces the same output as that found from Listing 20-6. The difference is that making the remote method calls in this fashion is much more convenient.

Publishing a Web Service

Of course, you might want to not only consume Web Services, but also publish them. After all, how better to offer your vast compilation of boxing quotes to the world? In this section, you'll learn how to use NuSOAP to create a Web Service that does just this.

For starters, you need to create a PostgreSQL table that hosts the quotes. Although a real-world implementation would involve multiple tables, this example is purposely kept simple, with everything encapsulated in a single table named `quotation`:

```
CREATE TABLE quotation (
   id SERIAL,
   boxer VARCHAR(30) NOT NULL,
   quote TEXT NOT NULL,
   year DATE NOT NULL,
   PRIMARY KEY(id)
);
```

Assume that this table has been packed with profound statements from the world's greatest fighters. Next, you need to create the Web Service. The commented script is offered in Listing 20-8.

Listing 20-8. *The Boxing Quote Web Service (boxing.php)*

```
<?php
   require('nusoap/nusoap.php');

   // Function: getRandQuote()
   // Inputs: None
   // Outputs: A string containing information about a quote,
   //  its attribution, and date.
   function getRandQuote() {
     // Connect to the PGSQL server
     $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");

      // Create and execute the query
      $query = "SELECT boxer, quote, date_part('year', year) FROM quotation
                     ORDER BY RAND() LIMIT 1";
      $result = pg_query($query);
      $row = pg_fetch_array($result);

      // Retrieve, assemble, and return the quote data
      $boxer = $row["boxer"];
      $quote = $row["quote"];
      $year = $row["year"];
      return "\"$quote\", $boxer ($year)";
   }

   // Instantiate a new soap server object
   $server = new soap_server;

   // Register the getRandQuote() method
   $server->register("getRandQuote");

   // Automatically execute any incoming request
   $server->service($HTTP_RAW_POST_DATA);
?>
```

All that's left is to create a client capable of consuming our service. This client is offered in Listing 20-9.

Listing 20-9. *A Boxing Web Service Client*

```
<?php
   require_once('nusoap/nusoap.php');
   $serviceURL = "http://localhost/book/20/boxingserver.php";
   $soapclient = new soapclient($serviceURL);
   $quote = $soapclient->call('getRandQuote');
   echo "<p>Your random boxing quotation of the moment:<br />$quote</p>";
?>
```

Contacting the Web Service using this client results in a random quote being retrieved from the `quotation` database table. Sample output follows:

```
"It's easy to do anything in victory. It's in defeat that a man reveals himself.",
Floyd Patterson (1935)
```

Returning an Array

You'll often want to retrieve various items of information from a Web Service, such as a profile of a given fighter in the boxing quote example. One of the easiest ways to do so is by returning an array back to the client. This is accomplished using PHP's default functionality, returning the array just like any other variable. This is demonstrated in Listing 20-10.

Listing 20-10. *Returning an Array to the Client*

```
<?php
   require_once('nusoap/nusoap.php');
   // Create a new server
   $server = new soap_server;

   // Register the retrieveBio() function
   $server->register("retrieveBio");

   // Define the retrieveBio() function
   function retrieveBio() {
      // Assume that this information was retrieved from a database
      $boxer["name"] = "Muhammed Ali";
      $boxer["age"] = 61;
      $boxer["bio"] = "Ali held the World heavyweight title three times
                       throughout his career.";
      return $boxer;
   }

   $HTTP_RAW_POST_DATA = isset($HTTP_RAW_POST_DATA) ?
   $HTTP_RAW_POST_DATA : '';

   $server->service($HTTP_RAW_POST_DATA);
?>
```

The client can contact the `retrieveBio()` function, and parse the array information using the `list()` statement, like so:

```
<?php
   require_once('nusoap/nusoap.php');
```

```
    // Always create a parameter array
    $params = array();

    // Create a new SOAP client
    $client = new soapclient("http://localhost/book/20/boxing.php");

    // Execute the remote method retrieveBio()
    $boxer = $client->call('retrieveBio', $params);

    // Parse the returned associative array
    $name = $boxer["name"];
    $age = $boxer["age"];
    $bio = $boxer["bio"];

    // Output the information
    echo "<strong>$name</strong> ($age years)<br />$bio";
?>
```

Executing the client results in the following output:

```
<strong>Muhammed Ali</strong> (61 years)<br />
Ali held the World heavyweight title three times throughout his career.
```

Generating a WSDL Document

You'll need to generate a Web Services Definition Language (WSDL) document in order to offer clients the opportunity to call methods via a proxy as was demonstrated in Listing 20-7. Doing so via NuSOAP is surprisingly easy, accomplished with few modifications to the servers demonstrated thus far. Two additional methods must be called to initiate WSDL configuration and specify the WSDL namespace: `configureWSDL()` and `schemaTargetNamespace()`, respectively. In addition, because PHP is a loosely typed language, both the input and returned values must be defined using XML Schema, which hints at the datatype requirements. Listing 20-11 is a modified version of Listing 20-7, offering WSDL generation support.

Listing 20-11. *Generating WSDL*

```
<?php
    require('nusoap/nusoap.php');

    $server = new soap_server();

    // Initiate WSDL configuration
    $server->configureWSDL('boxing', 'urn:boxing');

    // Designate the WSDL namespace
    $server->wsdl->schemaTargetNamespace = 'urn:boxing';
```

```
   // Register the getRandQuote() function.
   $server->register("getRandQuote",
               array('format' => 'xsd:string'),
               array('return' => 'xsd:string'),
               'urn:boxing',
               'urn:boxing#getRandQuote');

   function getRandQuote() {
      $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
      $query = "SELECT boxer, quote, year FROM quotation
                ORDER BY RAND() LIMIT 1";
      $result = pg_query($query);
      $row = pg_fetch_array($result);
      $boxer = $row["boxer"];
      $quote = $row["quote"];
      $year = $row["year"];
      return "\"$quote\", $boxer ($year)";
   }

$HTTP_RAW_POST_DATA = isset($HTTP_RAW_POST_DATA) ? $HTTP_RAW_POST_DATA : '';
$server->service($HTTP_RAW_POST_DATA);
?>
```

Fault Handling

NuSOAP offers a class for handling errors that may occur during execution. This class, named `soap_fault`, has four attributes:

- `faultactor`: This optional attribute indicates which service caused the error.
- `faultcode`: This required attribute indicates the type of error. There are four possible values: `Client`, `MustUnderstand`, `Server`, and `VersionMismatch`. A `Client` error is returned when an error message is found within the message returned by the client. A `MustUnderstand` error occurs when a mandatory header has been found that is not understood. A `Server` error occurs when a processing error has occurred on the server. Finally, a `VersionMismatch` error occurs when incompatible namespaces have been used.
- `faultdetail`: This optional attribute contains additional information about the error.
- `faultstring`: This required attribute contains an error description.

These attributes are initialized via the class constructor, like so:

```
<?php
   if ($bid < 10)
      return new soap_fault("Client", "",
                            "Dollar value must be greater than 10!", "");
   else
      return "Bid accepted";
?>
```

The soap_fault class also offers one method, serialize(). This function returns a complete SOAP message consisting of the fault information. Consider an example:

```
$fault = new soap_fault("Client", "",
                        "Dollar value must be greater than 10!", "");
$fault->serialize();
```

This returns the following:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:si="http://soapinterop.org/xsd">
<SOAP-ENV:Body>
   <SOAP-ENV:Fault>
      <faultcode>Client</faultcode>
      <faultactor></faultactor>
      <faultstring>Dollar value must be greater than 10!</faultstring>
      <detail><soapVal xsi:type="xsd:string"></soapVal></detail>
   </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Designating an HTTP Proxy

If the client requires use of an HTTP proxy server, it can be set using the setHTTPProxy() method. This method takes two arguments, the proxy address and its port:

```
$client = new soapclient("http://www.example.com/boxing/server.php", 80);
$client->setHTTPproxy("proxy.examplecompany.com", 8080);
```

All subsequent communication with the SOAP server initiated by this client will be routed through the designated proxy.

Debugging Tools

NuSOAP offers a debugging feature, which can be enabled on both the client and server sides. The method for enabling on each is identical, done by setting the debug_flag property to TRUE. For example:

```
$client = new soapclient($endpoint);
$client->debug_flag = true;
```

When debugging via the client, you can begin accessing debugging information via the property `debug_str`, like so:

```
echo $client->debug_str;
```

Because the returned string is quite lengthy, it will be difficult to read if you output it to the browser. You can improve its readability by replacing all newline characters with the `br` tag via the `nl2br()` function:

```
echo nl2br($soapclient->debug_str);
```

When debugging via the server, the debugging information will automatically be appended to any response. Interestingly, you can also enable server debugging from the client side by appending `?debug=1` to the Web Service endpoint URL. This causes the server to automatically append the debugging information to the response, as if debugging were enabled on the server side.

Two additional debugging attributes are available to the client:

- `request`: Retrieves the request header and accompanying SOAP message. It's called like so:

```
echo $soapclient->request;
```

- `response`: Retrieves the request response header and its accompanying SOAP message. It's called like so:

```
echo '<xmp>'.$soapclient->response.'</xmp>';
```

Secure Connections

Security should always be a subject of considerable concern when developing Internet-based applications. One of the de facto security safeguards in widespread use today is the Secure Sockets Layer (SSL) protocol, used to encrypt traffic sent over the Internet. NuSOAP supports SSL connections if the cURL extension is configured for PHP. Due to this extension's popularity, it's been bundled with PHP 5, and is enabled by configuring PHP with the `--with-curl` option.

Secure connections are initiated as is done via the Web browser, by prefacing the domain address with `https` rather than `http`.

PHP 5's SOAP Extension

In response to the community clamor for Web Services–enabled applications, and the popularity of third-party SOAP extensions, a native SOAP extension was incorporated into PHP 5. This section introduces this new object-oriented extension, offering several examples demonstrating how easy it is to create SOAP clients and servers. Along the way, you'll learn more about many of the functions and methods available through this extension. Before you can follow along with the accompanying examples, you need to take care of a few prerequisites, which are discussed next.

Prerequisites

PHP's SOAP extension requires the GNOME XML library. You can download the latest stable `libxml2` package from `http://www.xmlsoft.org/`. Binaries are also available for the Windows platform. Version 2.5.4 or greater is required. You'll also need to configure PHP with the `--enable-soap` extension.

Creating a SOAP Client

Creating a SOAP client with the new native SOAP extension is easier than you think. Although several client-specific methods are provided with the SOAP extension, only `SoapClient()` is required to create a complete WSDL-enabled client object. Once created, it's just a matter of calling the SOAP server's exposed functions. The `SoapClient()` method and several others are introduced next, guiding you through the process of creating a functional SOAP client as the section progresses. In the later section, "SOAP Client and Server Interaction," you'll find a complete working example of interaction between a client and server created using this extension.

SoapClient()

```
object SoapClient->SoapClient (mixed wsdl [, array options])
```

The `SoapClient()` constructor instantiates a new instance of the `SoapClient` class. The `wsdl` parameter determines whether the class will be invoked in WSDL or non-WSDL mode. If the former, then the parameter will point to the WSDL file; otherwise, it will be set to null. The discretionary `options` parameter is an array that accepts the following parameters:

- `actor`: This parameter specifies the name, in URI format, of the role that a SOAP node must play in order to process the header.
- `compression`: This parameter specifies whether data compression is enabled. Presently, `gzip` and `x-gzip` are supported. According to the TODO document, support is planned for HTTP compression.
- `exceptions`: Enabling this parameter turns on the exception-handling mechanism. It is enabled by default.
- `login`: If HTTP authentication is used to access the SOAP server, this parameter specifies the username.
- `password`: If HTTP authentication is used to access the SOAP server, this parameter specifies the password.
- `proxy_host`: This parameter specifies the name of the proxy host when connecting through a proxy server.
- `proxy_login`: This parameter specifies the proxy server username if one is required.
- `proxy_password`: This parameter specifies the proxy server password if one is required.
- `proxy_port`: This parameter specifies the proxy server port when connecting through a proxy server.

- `soap_version`: This parameter specifies whether SOAP version 1.1 or 1.2 should be used. This defaults to version 1.1.
- `trace`: If you would like to examine SOAP request and response envelopes, you'll need to enable this by setting it to 1.

Establishing a connection to a Web Service is trivial. The following example creates a `SoapClient` object that references the XMethods.net Weather Web Service, first introduced in the NuSOAP discussion earlier in this chapter:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $client = new SoapClient($ws);
?>
```

However, just referencing the Web Service really doesn't do you much good. You'll want to learn more about the methods exposed by this Web Service. Of course, you can open up the WSDL document in the browser or a WSDL viewer. However, you can also retrieve the methods programmatically using the `__getFunctions()` method, introduced next.

__getFunctions()

```
array SoapClient->__getFunctions()
```

The `__getFunctions()` method returns an array consisting of all methods exposed by the service referenced by the `SoapClient` object. The following example establishes a connection to the XMethods.net Weather Web Service and retrieves a list of available methods:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $client = new SoapClient($ws);
   var_dump($client->__getFunctions());
?>
```

This example returns:

```
array(1) {
      [0]=> string(30) "float getTemp(string $zipcode)"
   };
```

A single exposed method has been returned, `getTemp()`, which accepts a ZIP code as its lone parameter. The following example uses this method:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $zipcode = "20171";
   $client = new SoapClient($ws);
   echo "It's ".$client->getTemp($zipcode)." degrees at zipcode $zipcode.";
?>
```

This example returns:

```
It's 74 degrees at zipcode 20171.
```

__getLastRequest()

```
string SoapClient->__getLastRequest()
```

When you're debugging, it's useful to view the SOAP request in its entirety, headers and all. You can do so by turning on tracing when creating the `SoapClient` object, and invoking the `__getLastRequest()` method after a SOAP request has been executed. This is best explained with an example:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $zipcode = "20171";
   $client = new SoapClient($ws,array('trace' => 1));
   $temperature = $client->getTemp($zipcode);
   echo htmlspecialchars($client->__getLastRequest());
?>
```

This example returns (formatted for readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1="urn:xmethods-Temperature"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body><ns1:getTemp>
      <zipcode xsi:type="xsd:string">20171</zipcode>
      </ns1:getTemp>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

__getLastResponse()

```
object SoapClient->__getLastResponse()
```

The `__getLastRequest()` method is useful for reviewing the SOAP request in its entirety, envelope and all. When debugging, it's equally useful to review the response, accomplished using the `__getLastResponse()` method. As is the case with `__getLastRequest()`, tracing must be turned on. Consider an example:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $zipcode = "20171";
   $client = new SoapClient($ws,array('trace' => 1));
   $temperature = $client->getTemp($zipcode);
   echo htmlspecialchars($client->__getLastResponse());
?>
```

This example returns (formatted for readability):

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">76.0</return>
      </ns1:getTempResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Creating a SOAP Server

Creating a SOAP server with the new native SOAP extension is easier than you think. Although several server-specific methods are provided with the SOAP extension, only three methods are required to create a complete WSDL-enabled server. This section introduces these and other methods, guiding you through the process of creating a functional SOAP server as the section progresses. The next section, "SOAP Client and Server Interaction," offers a complete working example of the interaction between a WSDL-enabled client and server created using this extension. To illustrate this, the examples in the upcoming section refer to Listing 20-12, which offers a sample WSDL file. Directly following the listing, a few important SOAP configuration directives are introduced that you need to keep in mind when building SOAP services using this extension.

Listing 20-12. *A Sample WSDL File (boxing.wsdl)*

```
<?xml version="1.0" ?>
  <definitions name="boxing"
               targetNamespace="http://www.example.com/boxing"
    xmlns:tns="http://www.example.com/boxing"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
  <message name="getQuoteRequest">
    <part name="boxer" type="xsd:string" />
   </message>

   <message name="getQuoteResponse">
     <part name="return" type="xsd:string" />
   </message>

   <portType name="QuotePortType">
     <operation name="getQuote">
       <input message="tns:getQuoteRequest" />
       <output message="tns:getQuoteResponse" />
     </operation>
   </portType>

   <binding name="QuoteBinding" type="tns:QuotePortType">
     <soap:binding
           style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
     <operation name="getQuote">
       <soap:operation soapAction="" />
         <input>
           <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </input>
         <output>
           <soap:body use="encoded"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </output>
     </operation>
   </binding>

  <service name="boxing">
    <documentation>Returns quote from famous pugilists</documentation>
    <port name="QuotePort" binding="tns:QuoteBinding">
       <soap:address
        location="http://localhost/book/20/boxing/boxingserver.php" />
    </port>
  </service>
</definitions>
```

Important Configuration Directives

There are three important configuration directives that you need to keep in mind when building SOAP services using the native SOAP extension. These directives are introduced in this section.

soap.wsdl_cache_enabled

Scope: PHP_INI_ALL; Default value: 1

This directive determines whether the WSDL caching feature is enabled.

soap.wsdl_cache_dir

Scope: PHP_INI_ALL; Default value: /tmp

This directive determines the location where WSDL documents are cached.

soap.wsdl_cache_ttl

Scope: PHP_INI_ALL; Default value: 86400

This directive determines the time, in seconds, that a WSDL document is cached.

SoapServer()

```
object SoapServer->SoapServer (mixed wsdl [, array options])
```

The SoapServer() constructor instantiates a new instance of the SoapServer class in WSDL or non-WSDL mode. If you require WSDL mode, you need to assign the wsdl parameter the WSDL file's location, or else set it to NULL. The discretionary options parameter is an array used to set one or both of the following options:

- actor: Identifies the SOAP server as an actor, defining its URI.
- soap_version: Determines the supported SOAP version, and must be set with the syntax SOAP_*x*_*y*, where *x* is an integer specifying the major version number, and *y* is an integer specifying the corresponding minor version number. For example, SOAP version 1.2 would be assigned as SOAP_1_2.

The following example creates a SoapServer object referencing the boxing.wsdl file:

```
$soapserver = new SoapServer("boxing.wsdl");
```

Of course, if the WSDL file resides on another server, you can reference it using a valid URI. For example:

```
$soapserver = new SoapServer("http://www.example.com/boxing.wsdl");
```

However, creating a SoapServer object is only one task of several required to create a basic SOAP server. Next, you need to export at least one function, a task accomplished using the addFunction() method, introduced next.

■ **Note** If you're interested in exposing all methods in a class through the SOAP server, use the method setClass(), introduced later in this section.

---

addFunction()

```
void SoapServer->addFunction (mixed functions)
```

You can make a function available to clients by exporting it using the `addFunction()` method. In the WSDL file, there is only one function to implement, `getQuote()`. It takes `$boxer` as a lone parameter, and returns a string. Let's create this function and expose it to connecting clients:

```
<?php
   function getQuote($boxer) {
      if ($boxer == "Tyson") {
         $quote = "My main objective is to be professional
                   but to kill him. (2002)";
      } elseif ($boxer == "Ali") {
         $quote = "I am the greatest. (1962)";
      } elseif ($boxer == "Foreman") {
         $quote = "Generally when there's a lot of smoke,
                   there's just a whole lot more smoke. (1995)";
      } else {
         $quote = "Sorry, $boxer was not found.";
      }
      return $quote;
   }

   $soapserver = new SoapServer("boxing.wsdl");

   $soapserver->addFunction("getQuote");
?>
```

When two or more functions are defined in the WSDL file, you can choose which ones are to be exported by passing them in as an array, like so:

```
$soapserver->addFunction(array("getQuote","someOtherFunction");
```

Alternatively, if you would like to export all functions defined in the scope of the SOAP server, you can pass in the constant, `SOAP_FUNCTIONS_ALL`, like so:

```
$soapserver->addFunction(array(SOAP_FUNCTIONS_ALL);
```

It's important to understand that exporting the functions is not all that you need to do to produce a valid SOAP server. You also need to properly process incoming SOAP requests, a task handled for you via the method `handle()`. This method is introduced next.

handle()

```
void SoapServer->handle ([string soap_request])
```

Incoming SOAP requests are received by way of either the input parameter `soap_request` or the PHP global `$HTTP_RAW_POST_DATA`. Either way, the method `handle()` will automatically direct the request to the SOAP server for you. It's the last method executed in the server code. You call it like this:

```
$soapserver->handle();
```

setClass()

```
void SoapServer->setClass (string class_name [, mixed args])
```

Although the `addfunction()` method works fine for adding functions, what if you want to add class methods? This task is accomplished with the `setClass()` method, with the `class_name` parameter specifying the name of the class, and the optional `args` parameter specifying any arguments that will be passed to a class constructor. Let's create a class for the boxing quote service, and export its methods using `setClass()`:

```
<?php
  class boxingQuotes {
    function getQuote($boxer) {
      if ($boxer == "Tyson") {
        $quote = "My main objective is to be professional
                  but to kill him. (2002)";
        } elseif ($boxer == "Ali") {
          $quote = "I am the greatest. (1962)";
        } elseif ($boxer == "Foreman") {
          $quote = "Generally when there's a lot of smoke,
                    there's just a whole lot more smoke. (1995)";
        } else {
          $quote = "Sorry, $boxer was not found.";
        }
        return $quote;
      }
    }

    $soapserver = new SoapServer("boxing.wsdl");

    $soapserver->setClass("boxingQuotes");
    $soapserver->handle();
?>
```

The decision to use `setClass()` instead of `addFunction()` is irrelevant to any requesting clients.

setPersistence()

```
void SoapServer->setPersistence (int mode)
```

One really cool feature of the SOAP extension is the ability to persist objects across a session. This is accomplished with the `setPersistence()` method. This method only works in conjunction with `setClass()`. Two modes are accepted:

- SOAP_PERSISTENCE_REQUEST: This mode specifies that PHP's session-handling feature should be used to persist the object.
- SOAP_PERSISTENCE_SESSION: This mode specifies that the object is destroyed at the end of the request.

SOAP Client and Server Interaction

Now that you're familiar with the basic premises of using this extension to create both SOAP clients and servers, this section presents an example that simultaneously demonstrates both concepts. This SOAP service retrieves a famous quote from a particular boxer, and that boxer's last name is requested using the exposed getQuote() method. It's based on the boxing.wsdl file shown in Listing 20-12. Let's start with the server.

Boxing Server

The boxing server is simple but practical. Extending this to connect to a database server would be a trivial affair. Let's consider the code:

```
<?php
  class boxingQuotes {
    function getQuote($boxer) {
      if ($boxer == "Tyson") {
        $quote = "My main objective is to be professional
                  but to kill him. (2002)";
        } elseif ($boxer == "Ali") {
          $quote = "I am the greatest. (1962)";
        } elseif ($boxer == "Foreman") {
          $quote = "Generally when there's a lot of smoke,
                    there's just a whole lot more smoke. (1995)";
        } else {
          $quote = "Sorry, $boxer was not found.";
        }
        return $quote;
      }
    }

    $soapserver = new SoapServer("boxing.wsdl");

    $soapserver->setClass("boxingQuotes");
    $soapserver->handle();
?>
```

The client, introduced next, will consume this service.

Boxing Client

The boxing client consists of just two lines, the first instantiating the WSDL-enabled SoapClient() class, and the second executing the exposed method getQuote(), passing in the parameter "Ali":

```
<?php
   $client = new SoapClient("boxing.wsdl");
   echo $client->getQuote("Ali");
?>
```

Executing the client produces the following output:

```
I am the greatest. (1962)
```

Using a C# Client with a PHP Web Service

Although Linux is in widespread use as a server platform, it's apparent that the Microsoft Windows operating system will continue to dominate the desktop for some time to come. That said, quite a bit of interest has been generated regarding using Web Services as the tool of choice to enable Windows-based desktop applications to seamlessly integrate with Linux-based server applications. This section offers a brief yet effective example that demonstrates just how easy it is to do this. Specifically, we'll create a simple console-based C# application that talks to the PHP-based boxing Web Service built using the NuSOAP extension (refer to Listing 20-8). Although it's simplistic, this example should provide you with enough information to get the ball rolling on more complex applications.

In this final example, a C# application and our PHP Web Service will be coerced into playing nice with each other. This example is particularly compelling because it demonstrates just how easy it is to integrate a Windows desktop application and an open-source server. Because not everybody has a copy of Visual Studio .NET at their disposal, this example uses the freely downloadable .NET Framework SDK, which contains all the tools you need to successfully carry out this experiment. If you're running Visual Studio .NET, the general process is the same, although considerably more streamlined.

For demonstration purposes, we'll use the PHP-based boxing Web Service discussed throughout this chapter. The finished C# client simply invokes the `getRandQuote()` function, outputting a random quotation to a console window. Example output is provided in Figure 20-5.

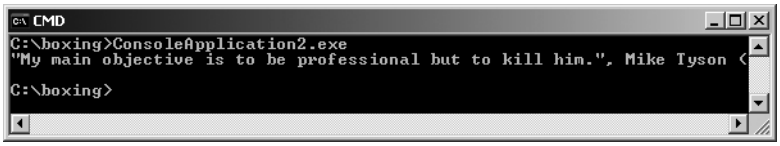

Figure 20-5. *Retrieving a random quote via a C# client*

If you don't already have it installed, you need to download and install the .NET Framework SDK to follow along with the example. Because the URL is quite long, execute a search on the Microsoft site (`http://search.microsoft.com/`) for the package. In addition, you need to download the .NET Framework Redistributable Package, which is also readily available from the Microsoft Web site. If you're unfortunate enough to be using a dial-up connection, consider ordering both on CD, because the SDK weighs in at over 100MB, while the redistributable package tops out at over 24MB.

Once the packages are installed, it's time to begin. For starters, you need to generate a C# proxy for the Web Service. You can do this by using the Web Services Description Language tool (`wsdl.exe`), included within the SDK. Reference the WSDL-enabled boxing server script shown in Listing 20-8:

```
wsdl /l:CS /protocol:SOAP http://localhost/book/20/boxing.php?wsdl
```

The result is a file named `boxing.cs`. Feel free to open it up and examine the file's contents; just be sure not to change anything. Next, you'll compile this proxy as a DLL library. This is necessary because the DLL will be referenced by the C# application so that the Web Service's methods can be called. You compile a DLL like you would any other C# program, using the C# compiler tool (`csc.exe`):

```
csc /t:library /r:System.Web.Services.dll /r:System.Xml.dll boxing.cs
```

The `/r` flags tell the compiler to reference these libraries during the compilation process. The result is a file named `boxing.dll`. In turn, you'll reference this DLL when you compile the C# SOAP client, discussed next.

■ **Note** Generating and compiling the proxy via the command line is indeed a tedious process. Bear in mind that the process is automated within Visual Studio .NET, greatly reducing development overhead.

Finally, create the C# application. Although you could conceivably create a full-blown GUI application using a text editor, to stay on track, we'll forego doing so here. Instead, create a simple console application, as shown in Listing 20-13.

Listing 20-13. *The C# SOAP Client*

```
using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;

namespace ConsoleApplication
{
   class boxing
   {
      [STAThread]
      static void Main(string[] args)
      {
         BoxingService bx = new BoxingService();
         Console.WriteLine(bx.getRandQuote());
      }
   }
}
```

Compile this client, like so:

```
csc boxing.cs /r:boxing.dll
```

What results is a file named `boxing.exe`. This is the executable C# client. Finally, test your program by executing it, like so:

```
C:\vs\proj\book\20\boxing.exe
```

Pending no unforeseen issues, you should see output similar to that shown in Figure 20-5.

Summary

The promise of Web Services and other XML-based technologies has generated an incredible amount of work in this area, with progress regarding specifications, and the announcement of new products and projects happening all of the time. No doubt such efforts will continue, given the incredible potential that this concentration of technologies has to offer.

In the next chapter, you'll turn your attention to the security-minded strategies that developers should always keep at the forefront of their development processes.