Pro PHP XML and Web Services

Robert Richards

Apress®

Pro PHP XML and Web Services

ISBN-13: 978-1-59059-633-3

ISBN-10: 1-59059-633-1

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matt Wade
Technical Reviewers: Christian Stocker, Adam Trachtenberg
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Nancy Sixsmith
Indexer: Jan Wright
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

This book is dedicated to my wife and best friend, Julie.
Thank you for your patience, support, and encouragement
at the times I most needed it.

CHAPTER 5

PHP and XML

The latest version of PHP, PHP 5, introduces several new features and enhancements to the PHP language. PHP 5 introduced a new object model, exceptions, and new database support such as MySQLi and SQLite, and it makes major strides in the areas of XML and Web services. This chapter will introduce you to the new XML-based extensions, their founding library, and the basic functionality common to the PHP 5 XML extensions.

Introducing XML in PHP 5

Native XML support in PHP 4 was limited to certain basic technologies. The xml extension supported SAX, the domxml extension provided tree support as well as some XSLT support, and the xslt extension also provided XSLT support. With respect to Web services and data exchange, the wddx extension supported distributed data exchange, and xmlrpc supported XML-based remote procedure calls. Although this seems like a decent list of technologies, a fundamental problem was that each was its own distinct extension using its own underlying library. The extensions just did not work together, and to use all the extensions, you had to install all the necessary libraries.

The shortcomings of XML in PHP 4 caused much frustration for those using it. All this XML technology was available, but it would not work together. So, while PHP 5 was still in its early stages of development, a discussion began that would ultimately shape the future of XML in PHP 5. The developers decided to rework and rewrite the XML-based extensions to provide the greatest functionality and flexibility as possible.

libxml2 in PHP 5

The central library decided upon for the core of the XML extensions is libxml2, which you can find at `http://www.xmlsoft.org`. This library supports many of XML-related standards, including the XML, Namespaces, XML Schemas, Relax NG, XPath, and XInclude specifications—just to name a few. It was chosen for its vast XML support, which means additional technologies can be implemented in PHP, and it is one of the fastest parsers; also, it is actively maintained and widely used. Its sibling, the libxslt library, which is dependent upon libxml2 and also located at `http://www.xmlsoft.org`, handles XSL within PHP 5.

Both of these libraries, being actively maintained, continue to evolve by providing fixes to bugs and enhanced feature sets. To provide the best XML support possible, it is sometimes necessary to require newer versions of these two libraries in order to build PHP. Such is the case with PHP 5.1. The current minimum requirements for libxml2 and libxslt within PHP

are libxml2 2.5.10 and libxslt 1.0.18 under PHP 5.0.*x* and libxml2 2.6.11 and libxslt 1.0.18 under PHP 5.1.*x*. Although minimum requirements for these libraries have been established, it is always a good idea to keep your libraries current. The bugs fixed in the latest versions alone will enhance and ensure proper XML support within the PHP extensions. Some extensions also provide new or additional functionality available only with newer libxml2 libraries.

■ **Tip** Keeping your libxml2 and libxslt libraries current ensures you have the latest bug fixes, but it also means, for some PHP 5 extensions, you'll get additional functionality not found in earlier versions. Keep in mind that this does not mean your application will behave exactly as it did before an upgrade. Both libraries follow the XML specifications, so a fix in either library to conform to specifications may adversely affect any expected output. In cases such as this, it is advantageous to correct the problem in the application rather than rely on old library behavior that might not have been correct in the first place.

Core XML Extensions

Many XML extensions and packages exist for PHP, which will be mentioned later in this book. If you are a PHP Extension and Application Repository (PEAR) fan, you have not been forgotten. I have intentionally omitted discussing PEAR at this point because Chapter 13 is dedicated to PEAR and XML. I will limit the current scope of this chapter to an introduction of the extensions bundled with core PHP 5.

Tree-Based Parsers

Tree-based parsers allow you to construct or load existing XML documents so you can navigate or modify them. To do this, the entire XML document is created or loaded into memory as a tree. Given that the entire document must reside in memory, you need to consider your memory constraints when using these technologies. These parsers also tend to initially be slower for this same reason. Once in memory, however, these parsers offer the fastest access to data within a document compared to other types of parsers.

Under PHP 4, domxml was the only native tree-based parser available. PHP 5 introduced the new parsers DOM and SimpleXML. If you are unfamiliar with these parsers, then you may be wondering why you need two. You will get an idea from the following descriptions; and after reading Chapters 6, 7, and 11, you will have the full picture.

SimpleXML Extension

Using the new functionality offered by PHP 5, SimpleXML provides an extremely simple and lightweight tool to manipulate XML documents. Compared to the DOM extension, SimpleXML has an easy-to-learn API because you can view the document as a tree of objects, where objects are synonymous with element nodes. Accessing a child element is as simple as using the child element's name as a property of an object. You can access attributes similarly to how you access an array. To a limited extent, SimpleXML also allows for content editing. You can find further information about SimpleXML in Chapter 7, which details this extension and offers examples on usage.

DOM Extension

The DOM extension is the PHP 5 replacement for domxml, which is now supported only under PHP 4. The DOM extension was created to address many of the shortcomings of domxml while also adhering to the W3C DOM specifications. Unlike SimpleXML, it has a large and complex API. This, however, is the price you pay for functionality. The DOM extension allows you to access all node types, allows you to create and modify complex documents, and gives you advanced navigation and functionality. An advantage to this extension, if you are coming from another language that incorporates a DOM-compliant parser, is that the API should already be familiar to you and easy to begin using under PHP. The next chapter covers this extension in detail.

Streaming Parsers

Unlike a tree-based parser, a streams-based parser does not load the entire document into memory, so memory usage and requirements remain at a minimum. Only small pieces of the document are available for processing at a time. PHP 5 offers both a push parser via the xml extension and a pull parser via the XMLReader extension. These parsers do not allow for document editing and offer little to no navigational capabilities, because they are forward-only streams. The minor exception to this is XMLReader.

xml Extension

The xml extension is the familiar SAX-based tool from PHP 4. Within PHP 5, a libxml compatibility layer has been added as the default library, eliminating the need for expat, although it may still be built using expat. SAX offers event-based parsing. Functions, known as *handlers*, are assigned to events, such as when the beginning or end of an element is encountered, and data is sent to the functions for processing. This is known as a *push parser* because you are not in control of the data sent to your functions. Upon the commencement of parsing, reading of the XML document begins. As events are triggered, your handler is executed with the data whether or not you are interested in the actual data. This continues until you halt the parser, a fatal error occurs, or it reaches the end of the document. Chapter 8 covers the xml extension API and offers examples.

XMLReader Extension

The XMLReader extension takes a different approach than the xml extension. It works as a forward-only cursor on the XML document, stopping at each node in the document. The user controls the progress through the document as well as decides whether any information should be retrieved from the current node pointed at by the cursor. It is for these reasons XMLReader is called a *pull parser*. The ease of use, because of a small API, gives it some advantages over the xml extension; in addition, XMLReader offers faster processing without an increase in memory usage, offers streaming validation using DTDs or RELAX NG, offers support for namespaces, offers support for `xml:base` and `xml:id`, and provides interoperability with the other PHP extensions. Chapter 9 gives you an in-depth look at XMLReader and its usage.

■ **Note** XMLReader is available for PHP 5.0 as a PHP Extension Community Library (PECL) extension. As of PHP 5.1, XMLReader is available as a core extension.

XSL Extension

XSL is an XML-based style sheet language and the language used to transform XML documents into other XML documents. Chapter 10 covers XSLT in more depth, but a quick example is when you take an XML document and create an XHTML document from select data within the original XML document.

Just as the XSLT support from the domxml extension has been removed in PHP 5, so has the xslt extension. Along with the new DOM extension, PHP 5 offers a new XSL extension to work alongside it. This time, the DOM and XSL extensions not integrated into a single extension, but XSL is its own entity (though still dependant upon DOM). A new feature, present in the XSL extension, is the ability to execute PHP and use the resulting data within the transformation.

Data Exchange and Web Services

Using XML for exchanging data and integrating systems has become a hot topic of conversation. PHP 5 includes three native extensions in this area: wddx, xmlrpc, and SOAP. While both the wddx and xmlrpc extensions have been around since the PHP 4 days, the new native SOAP extension was created exclusively for PHP 5.

wddx Extension

Web Distributed Data Exchange (WDDX) offers the ability to serialize data and their native types into platform-neutral XML. This XML can then be transmitted to another system that can unserialize the data into its own native data types. No specific transport agent is defined for this technology, because you can use any Internet protocol. WDDX is strictly for serializing and unserializing data. Unlike the XML-RPC or SOAP technologies, WDDX doesn't attempt to define methods for calling remote functions. The wddx extension, which will be covered in Chapter 15, is the tool for utilizing the WDDX technology.

xmlrpc Extension

As you read in Chapter 1, XML-RPC was one of the early Web services. It is similar to WDDX in that data and their types are serialized and unserialized into/from XML, but it goes beyond this. XML-RPC defines HTTP as its transport agent and includes the mechanism for calling remote functions, which are also transported via an XML document. The xmlrpc extension, which will be covered in Chapter 15, is the extension supporting XML-RPC in PHP 5.

SOAP Extension

Native SOAP support in PHP 5 was a major advancement for the XML-based technologies. Prior to its inception, the alternatives were implementations written in PHP, such as PEAR::SOAP and NuSOAP. Although those are viable alternatives, the biggest advantage to native support written in C is the great improvement in speed as well as the extension being considered the standard SOAP implementation for PHP. You can find detailed information about the SOAP extension and its usage in Chapter 18.

libxml Extension

The libxml extension in PHP 5 is not your typical extension. It does not offer any type of specific XML technology. This extension serves as the center of common functionality shared across all XML-based extensions and uses libxml2 as its backend. This includes functionality exposed to PHP developers as well as those developing extensions using libxml2 as their library. Within PHP 5.0.*x*, the only user functionality you could control was stream contexts. You may ask why this is important. Later in the "Introducing PHP Streams" section, I will explain the relationship of PHP streams and XML. After PHP 5.0 was originally rolled out, one of the biggest issues developers brought up about using the extensions concerned the way error handling was implemented in XML. PHP 5.1 introduced new error handling that could be controlled and accessed through the libxml extension. I'll also discuss error handling for both PHP 5.0 and 5.1 later in the "Performing Error Handling" section.

Configuring libxml Support

By default the libxml extension is enabled. Using Windows, libxml2 is built into PHP. You do not need to worry about the `libxml2.dll` file as you did under PHP 4. Disabling this extension causes all extensions based on libxml to be disabled as well. You disable this and the other extensions simply by adding the following directive to your `configure` directive:

```
--disable-libxml
```

Because you are reading this book on PHP 5 and XML, I highly doubt this is something you would want to do. But it may be possible you still want the extensions coming from PHP 4 so you can continue to use expat. You can do this using the following:

```
--with-libexpat-dir= /path_to_libexpat
```

This directive takes priority over the `configure` directive for libxml, and if used, the extensions xml, wddx, and xmlrpc will be built using expat support rather than libxml2 support.

■ **Note** Unless you are encountering problems using the libxml2 library with the xml, wddx, and/or xmlrpc extensions, using libxml2 is highly recommended. Not only does it offer a performance boost, but it also has a greater number of active developers who can provide support in the event of any problems with extensions.

The libxml extension is enabled by default, but if it is disabled (because running some packaged version has changed the code shipped from the `http://www.php.net` site), you can enable it with this:

```
--enable-libxml
```

You can specify the location of the libxml2 libraries through a configuration directive. If you cannot determine the location by running `configure`, or if you would like to specify a different location such as testing a different version of libxml2, you can set the path using the following:

```
--with-libxml-dir=/path_to_libxml_config
```

This directive looks for the file `/path_to_libxml_config/bin/xml2-config`.

In many cases, you will not have to worry about changing or including any directives for libxml. The default `configure` included with PHP 5 works right out of the box for most systems, but this will depend upon your operating system. You can find installation help in the PHP manual as well as many places on the Internet. Now that you have your system up and running with libxml support, it's time to look at what libxml extension and libxml2 support means with respect to using any of the XML-based extensions.

Introducing Encoding

Internationalization is something encountered frequently when dealing with XML and when working on the Internet in general. Those new to XML often run into problems when dealing with documents not based on the ANSI encoding or the UTF-8 encoding. Basic knowledge of Unicode is highly suggested, because you will need to understand what it means for a string to be encoded and why it is important to know what encoding is used.

Parsers are required to support UTF-8 and UTF-16 at a minimum. The libxml2 library supports a few additional encodings natively, and when built with iconv support (`http://www.gnu.org/software/libiconv/`), it can support all encodings supported by iconv. The iconv library provides functionality for conversions between different encodings. You may already be familiar with this through the PHP iconv extension. Table 5-1 lists the base encodings supported by the libxml2 library. This is not an exhaustive list of available encoding names because many encodings are aliases to many of these character sets.

Table 5-1. *Base Default Encodings Supported in libxml2*

Character Set	Encoding
UTF-8	`UTF-8`
UTF-16	`UTF-16`
UTF-16 Big Endian	`UTF-16BE`
UTF-16 Little Endian	`UTF-16LE`
ISO-8859-1	`ISO-8859-1`
ASCII	`ASCII`
US_ASCII	`US_ASCII`
HTML	`HTML`

The last character set listed, HTML, is a special encoding within libxml2. It is used for output only and includes predefined HTML entities. For regular XML use, you should ignore this encoding; Chapter 10 will demonstrate its use.

Encoding Detection

As you have seen in earlier chapters, you specify the document encoding in the XML declaration. For documents without a specified encoding, libxml2 attempts to detect the encoding

based on the first few characters of the document or a byte order mark (BOM). A BOM is a sequence of bytes at the beginning of a data stream and can indicate the encoding form used. Table 5-2 lists the byte sequences and their corresponding encodings.

Table 5-2. *Byte Order Mark and Encodings*

Byte	Encoding
FE FF	UTF-16BE
FF FE	UTF-16LE
EF BB BF	UTF-8

Documents without a specified encoding or BOM in the data stream can also have their encoding detected based on the first few characters of the XML or test declaration. The encoding will be able to be detected only if a declaration exists. Table 5-3 lists the sequence of characters by their hexadecimal values and the corresponding encodings.

Table 5-3. *No BOM and Corresponding Encodings*

Character	Encoding
00 00 00 3C	ISO-10646-UCS-4
3C 00 00 00	ISO-10646-UCS-4
00 00 3C 00	ISO-10646-UCS-4
00 3C 00 00	ISO-10646-UCS-4
3C 3F 78 6D	UTF-8
4C 6F A7 94	EBCDIC
3C 00 3F 00	UTF-16LE
00 3C 00 3F	UTF-16BE

It may be evident now why XML declarations are recommended. Specifying an encoding not only eliminates the need for a parser to attempt to autodetect the encoding of the document, but it also makes it evident to someone looking at the document. In the event the encoding is not present in the declaration and is unable to be detected, libxml2 will use UTF-8 for the encoding, which is also the encoding it stores documents as internally. For instance, Listing 5-1 uses French characters and ISO-8859-1 encoding, although not explicitly specified.

Listing 5-1. *XML Document with French with No Encoding Defined*

```
<doc>
   <élément>contenu d'élément</élément>
</doc>
```

In this example, I didn't add any BOMs to the data stream, and no XML declaration exists. The parser cannot determine encoding, so it uses UTF-8 as a fallback. This presents a problem. The document is not proper UTF-8 encoding and thus fails when the parser attempts to load it. Trying to actually load this document results in the following libxml2 error:

```
Input is not proper UTF-8, indicate encoding !
```

Now that you know this fails, you can try using an XML declaration, as demonstrated in Listing 5-2, but still not specify encoding. This will at least give libxml2 a chance to try to auto-detect the encoding used.

Listing 5-2. *XML Document with French and XML Declaration but No Encoding*

```
<?xml version="1.0"?>
<doc>
   <élément>contenu d'élément</élément>
</doc>
```

This isn't surprising—the parser encounters the same error. The parser detected the XML declaration but detected it as UTF-8. So, the parser used the same encoding regardless of whether you specified the XML declaration. If you saved the document in Listing 5-2 as a file in UTF-16 format, the autodetection would have at least noticed this and tried loading it using UTF-16 as the encoding.

For the last try to get this document to load properly, set the `encoding` attribute on the XML declaration, as illustrated in Listing 5-3.

Listing 5-3. *XML Document with French and Encoding Specified*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<doc>
   <élément>contenu d'élément</élément>
</doc>
```

This time it finally loads without an error. The encoding you needed in this case was `ISO-8859-1`, which allows the use of the French characters within the document. If you now instructed the parser to dump the document to the standard console, you might not expect to see what it outputs:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<doc>
   <´l´ment>contenu d'´l´ment</´l´ment>
</doc>
```

You need to remember that your console may not be able to display all characters correctly. This output is from a console that doesn't support the ISO-8859-1 character set. The output is actually correct; it just doesn't look correct. The document was sent to a file, rather than to the standard output, so the contents of the file should be identical to the document in Listing 5-3. This leads to the next topic of discussion, internal storage of an XML document within libxml2.

Internal Encoding

Regardless of the encoding specified for a document, the encoding is stored internally within libxml2 in UTF-8 format. You may be wondering why you need to care about how internal data is encoded. This is actually important to understand when using any of the XML-based

extensions within PHP 5. The information contained within this section may save you countless hours of beating your head against the wall.

Once a document is loaded into the parser, you should completely ignore that an encoding may have been specified for the document. The document is stored and processed using UTF-8 encoding. Virtually all interaction with a parser or data from the parser must be performed using UTF-8 encoded data. Note that in a few instances this does not hold true, and as you read the later chapters covering the specific extensions, you will learn about the specific cases.

■ **Caution** Documents are internally stored using UTF-8 encoding. Interaction with XML data in these cases must be performed using UTF-8 data. You may need to perform encoding conversions using an extension such as iconv or mbstring in order to avoid a corruption of data.

The iconv and mbstring extensions in PHP are your friends. When dealing with data that is not UTF-8 compliant, you need to perform conversions. These extensions allow you to convert data to and from UTF-8 based on virtually any encoding you need to use. Say you need to add a new element with the content `contenu d'élément` to a document. Although you haven't gotten there yet, this example will use the DOM extension. Listing 5-4 illustrates how to use iconv and mbstring in order to perform encoding conversions. Because I have not covered the DOM extension yet, I have omitted the bulk of the code needed for processing.

Listing 5-4. *Encoding and Decoding Using iconv and mbstring*

```
<?php
$isostring = "contenu d'élément";

/* Conversions from ISO-8859-1 to UTF-8 */
$utf8string = iconv("ISO-8859-1", "UTF-8", $isostring);
$uft8string2 = mb_convert_encoding($isostring, "UTF-8", "ISO-8859-1");

/* Additional DOM code here */
$newelement = new DOMElement('newelement', $ utf8string);
$newelement2 = new DOMElement('newelement2', $ utf8string2);
/* Additional DOM code here */

/* Retrieve the content from newelement set above */
$value = $newelement->nodeValue;

/* Conversions from UTF-8 to ISO-8859-1 */
$isostring1 = iconv("UTF-8", "ISO-8859-1", $value);
$isostring2 = mb_convert_encoding($value, "ISO-8859-1", "UTF-8");
?>
```

The original data you began with, `contenu d'élément`, is stored in the variable `$isostring`. This data is in `ISO-8859-1` encoding, but in order to interact with the DOM extension, which is

based on libxml2, you need to convert `$isostring` to UTF-8. The code in Listing 5-4 illustrates how to perform this conversion using both iconv and mbstring (but you need to use only one). Be aware of the ordering of arguments for the functions. From the PHP manual, the prototypes for these functions are as follows:

```
string mb_convert_encoding (string str, string to_encoding [, mixed from_encoding])
string iconv (string in_charset, string out_charset, string str)
```

After performing the conversions, the strings using UTF-8 encoding, `$utf8string` and `$utf8string2`, are then used as values for the content of the `DOMElement` objects. Naturally these elements are added to the document within the omitted code. When reading the content of these objects, the reverse conversions are performed and stored in `$isostring1` and `$isostring2`. These strings will contain the same string as the original `$isostring` variable.

Whether you need to worry about internal encoding depends upon the character set of the data you are using. In many cases, you will be using the UTF-8 and ASCII character sets, and in these cases you do not need any conversions. When working with documents containing language-specific data or when working with internationalization and XML, you must deal with encoding properly.

Figuring Out the libxml2 Version

In some cases, the version of libxml2 used determines whether you can use certain functionality within an extension. For example, namespace support within the xml extension is functional only when running libxml2 2.6.*x*. Although 2.6.0 is the minimum version for PHP 5.1, PHP 5.0 can use XML functionality with at least 2.5.10. Attempting to use namespace support through the `xml_parser_create_ns` function when running PHP 5.0 with a 2.5.*x* version of libxml2 results in an error message, "Please upgrade to libxml2 2.6." You may also find that other extensions require other minimum versions to utilize certain functionality and methods.

This can make writing software difficult, because it is impossible to guess what version someone else may be running. Luckily, you can retrieve the version of libxml2 and use it programmatically. The libxml extension offers two constants for this purpose: `LIBXML_VERSION` and `LIBXML_DOTTED_VERSION`. `LIBXML_VERSION` is a numeric value indicating the major, minor, and micro version. `LIBXML_DOTTED_VERSION` indicates the same information but in dotted notation. Using these notations, libxml2 version 2.6.19 would result in the following:

```
/* 2.6.19 using LIBXML_VERSION */
20619

/* 2.6.19 using LIBXML_DOTTED_VERSION */
2.6.19
```

Using this programmatically with the `xml_parser_create_ns` function as an example, you could test whether the functionality is supported and provide an alternative in the event it is not:

```
<?php
if (LIBXML_VERSION >= 20600) {
     $xml = xml_parser_create_ns(…);
} else {
     $xml = xml_parser_create(…);
}
?>
```

Introducing Parser Options

As of PHP 5.1, the libxml extension contains new constraints that you can use in the DOM and SimpleXML extensions to control parser behavior. The parser uses these constants, listed in Table 5-4, at the time of document load to offer finer control over how the parser loads and parses the document.

Table 5-4. *Parser Option Constants*

Constant	Description
LIBXML_NOENT	Substitutes entities found within the document with their replacement content.
LIBXML_DTDLOAD	Loads any external subsets but does not perform validation. This flag also ensures that IDs set in a DTD are created within the document.
LIBXML_DTDATTR	Creates attributes within the document for any attributes defaulted through a DTD.
LIBXML_DTDVALID	Loads subsets and validates a document while parsing.
LIBXML_NOERROR	Suppresses errors from libxml2 that may occur while parsing.
LIBXML_NOWARNING	Surprises warnings from libxml2 that may occur while parsing.
LIBXML_NOBLANKS	Removes all insignificant whitespace within the document.
LIBXML_XINCLUDE	Performs all XIncludes found within the document.
LIBXML_NSCLEAN	Removes redundant namespace declarations found while parsing the document.
LIBXML_NOCDATA	Merges CDATA nodes into text nodes. A document using CDATA sections will be created with no CDATA nodes, because these will now be converted into plain-text nodes. This flag is useful when loading a document to be used for an XSL transformation.
LIBXML_NONET	Disables network access when loading documents. You can use this flag to increase security from untrusted documents so resources cannot be fetched from the network.

You can combine flags when parsing. For example, you can load a document that validates and suppresses all warnings while parsing using the following options:

```
LIBXML_DTDVALID | LIBXML_NOWARNING
```

These options would be passed as a single parameter to the function or method accepting a libxml parser option. I will demonstrate how to use these flags within the specific extensions in their respective chapters. Note the use of flags when working with XSL. CDATA sections

often make working in XSL difficult; specifically, certain XSL functions do not work correctly when a document contains CDATA sections, because the functions are specific to text nodes. Entities are also typically substituted within the XML document being transformed.

■ **Tip** When parsing a document to be used within an XSL transformation, it is recommended that you use the flags `LIBXML_NOENT` and `LIBXML _NOCDATA` to avoid any potential problems with calls made upon the XML document from the XSL style sheet.

If you are still using PHP 5.0.*x*, these options are not available. Under this version, the DOM extension does provide a few properties that can be used for controlling the parser, but they do not include all the options listed in Table 5-4. SimpleXML, on the other hand, does not offer any additional functionality to control the parser during document loading. The interoperability within PHP 5 may be useful in this case, assuming the DOM extension has been built, because you can load a document via the DOM extension and manipulate it using SimpleXML.

■ **Tip** Under PHP 5.0.*x*, limited parser options are available, even when manipulating the tree using SimpleXML. Documents can be loaded using the DOM extension and a few of the document properties that control the parser; and through the interoperability of the extensions, you can manipulate the resulting tree using SimpleXML.

Introducing PHP Streams

Resource input/output (I/O) for XML has completely changed with PHP 5. Under PHP 4, XML-based extensions used their native I/O mechanisms for the input and output of resources. If you recall from the domxml extension, the only protocols available would be specified as `file`, `http`, and, as an input-only protocol, `ftp`. The old xslt extension would allow support for additional I/O handlers, but it was not all that easy to accomplish because programmers had to deal with setting handlers and adding the functionality to make this work.

PHP 5 is much different. Built-in PHP streams support now serves as the foundation for I/O handling within the XML-based extensions. The advantages of this are numerous for both developers and system administrators. The advantages include the following:

- Built-in support for numerous protocols as well as user-defined streams
- Consistent I/O handling
- Support for PHP file security checks

Protocols

PHP includes many protocols, and the XML extensions by default have access to them all. No longer are the extensions limited to the protocols defined within their base libraries. Files can

now be accessed not only from the file system but also via `http`, `https`, `ftp`, `ftps`, PHP I/O streams, `zlib`, `compress.zlib`, and `compress.bzip2`. Prior to using PHP streams, unsupported protocols needed to have the file loaded into a string using PHP functions and that data sent to the extension to be processed as an in-memory string. This could get quite cumbersome for large documents. Not only did you have the overhead of the entire document loaded into memory for a tree parser, but the document was loaded in its string representation as well. You ended up getting penalized twice this way.

XML extensions can now take advantage of user-defined streams. If you are familiar with the streams functionality within PHP, you probably know that user-defined streams can be registered and used natively through the functions supporting stream usage. So, if you would like to define your own protocol—for example, `xyz://`—that uses your own defined I/O functionality, once registered, the XML extensions would have direct access to it.

Consistent I/O Handling

Using domxml in the past created a pathing issue. Depending upon whether PHP was run via the command line or an Apache module, as well as depending upon the operating system it was executing under, the base directory for files that an XML document accessed was not the same. For example, if your XML document contained relative paths for external entities or even for the location of XIncludes, the base directory did not always end up being the directory you assumed it would be. This problem even manifested itself depending upon the version of Apache being used. For instance, using Apache 2 under Windows, the base directory for an XML file sometimes ended up being the directory where the Apache binary lived.

This problem caused many headaches. It was difficult for developers to write cross-platform code. The domxml extension was eventually was fixed in some regard through workarounds, but it still exhibits some differences between operating systems. The move to PHP stream-based I/O now removes this problem. Pathing using streams is universal. The base directory will not change if your code is run from the command line or as a module under Apache—or even under a different operating system.

PHP File Security Support

Another advantage to using PHP streams comes from the built-in support for Safe Mode, which includes the `open_basedir` and `allow_url_fopen` `php.ini` options. These settings are typically employed in a shared server setting. Through the `php.ini` settings, a system administrator can control different aspects of file access. Prior to PHP 5, XML-based extensions used their internal I/O functionality based upon their base libraries. These libraries, having no concept of PHP streams, bypassed all the security settings.

By default, Safe Mode checks the user ID of the running script against the user ID of the file to be accessed. You can also relax the check using `safe_mode_gid` to compare group IDs as well. If the checks failed, access to the file would be denied. Accessing files using any of the XML extensions now follows the same rules, thus adding security checks when the extensions are accessible on the server.

The `open_basedir` setting allows directories to be set, limiting file access to only those within the specified directories and their subdirectories. The value for the setting is actually a prefix and not a directory. For example, a setting with the value `/usr/inc` would also match the directory `/usr/include`. To limit access to only the `/usr/inc` directory, the value would

need to include the trailing slash using /usr/inc/. This setting, independent from the Safe Mode settings, also will affect how files can be accessed using the extensions.

The last setting, allow_url_fopen, can be used to limit network access. When this setting is disabled, the XML parsers will not be allowed to open or save to any remote resource using protocols such as HTTP, HTTPS, and FTP. The local file system is still available for access, but those network resources are denied. Used in conjunction with the Safe Mode and open_basedir settings, access to resources can be locked down quite effectively.

Stream Context

Stream contexts are parameters and options that can modify the behavior of a stream. Many of the stream-enabled functions within PHP accept a stream context as a parameter. The XML functions are not included in this because stream usage is almost invisible from an API perspective. The libxml extension includes the function libxml_set_streams_context that you can use for this purpose.

You can create a context with the regular PHP Streams API. You can find full documentation for this API in the PHP user manual. You then set the context using libxml_set_streams_context; the context remains active for the entire duration of the script. Consider accessing a remote XML resource, located at http://www.example.com/test.xml, while sitting behind a proxy server located at http://www.example.net:4444. Listing 5-5 illustrates the contents of the remote documents.

Listing 5-5. *Contents of* test.xml *and* testxinclude.xml

```
/* Contents of test.xml */
<test xmlns:xi="http://www.w3.org/2001/XInclude">
   <xi:include href="testxinclude.xml" parse="xml" />
</test>

/* Contents of testxinclude.xml */
<testinclude>Included Content</testinclude>
```

The first task you must perform is to create the stream context:

```
<?php
$opts = array(
   'http'=>array(
   'proxy'=>"tcp://www.example.net:4444",
   'request_fulluri'=>TRUE
   )
);

$context = stream_context_create($opts);
?>
```

In this case, the proxy server is requiring a full URI to serve the request, which requires the additional request_fulluri option set to TRUE. The next steps require setting the context with the libxml extension:

```
libxml_set_streams_context($context);
```

With the context set, the parser can now request the XML resource as it normally would. This example uses the DOM extension:

```
$dom = DOMDocument::load('http://www.example.com/test.xml');
```

The parser pulls the XML resource using the proxy set by the context. Notice that the document contains an XInclude using a relative path. It should retrieve this resource from `http://www.example.xom/testinclude.xml`. When the XInclude operation is performed via the DOM extension, the proxy continues to service the requests.

Performing Error Handling

With the first release of PHP 5, many people were excited about the addition of advanced XML functionality. The largest complaint was in regard to error handling using the XML functions. Errors using XML not only are issued from PHP for user errors but also from libxml2 itself to indicate XML errors, such as when a malformed document is being parsed. Errors from libxml2 range from simple warnings, which in many cases can be safely ignored, to fatal errors, which may cause a PHP error to also be issued if the operation completely fails.

Both the SOAP and DOM extensions offer exceptions, but at least in the DOM extension's case the exceptions were limited to those defined in the specifications. Errors deriving directly from libxml2 were issued as `E_WARNINGS` and `E_NOTICES` depending upon the severity of the error. Typically developers did not care about these errors. They just cared whether the operation failed or succeeded. For those who did care about the errors, they had no way to determine that the errors were XML-specific. So, adding a user error handler might be fine, but it still did not indicate that the error was XML-specific.

To get around this problem, many developers started suppressing the errors and just checking return values. For example, you can load a document using SimpleXML from a string using the `simplexml_load_string` function:

```
$sxe = simplexml_load_string('<root>');
print $sxe->asXML();
```

Loading a malformed document results in an error. This is an invalid document because it contains a single start element with no end element. A typical error from PHP 5 would be an `E_WARNING` containing the message "Entity: line 1: parser error: Premature end of data in tag root line 1," followed by an error indicating that the developer was trying to call a member function from a nonobject. The load failed, and `$sxe` was never created. To avoid this error, the code was often changed to this:

```
if ($sxe = @simplexml_load_string('<root>')) {
   print $sxe->asXML();
}
```

The error has been suppressed, and the `print` statement is executed only if `$sxe` exists. This is all well and good, but all errors indicating the reason of failure have now been lost.

The complaints from developers did not go unnoticed. Things changed with PHP 5.1. For backward compatibility reasons, the error-handling behavior was left intact and is the default behavior. Additional error handling was added that allows XML errors to be suppressed while

also providing a mechanism for them to be accessed after the fact. The additional functions available from the libxml extensions that can access the new error-handling functionality include the following:

```
bool libxml_use_internal_errors ([bool use_errors])
void libxml_clear_errors ( void )
LibXMLError libxml_get_last_error ( void )
array libxml_get_errors ( void )
```

The function `libxml_use_internal_errors` is the central function, which turns on and off the new internal error handler. The optional `use_errors` parameter, which defaults to `FALSE`, indicates whether you should enable the internal error handler. The return value from the function contains the old value prior to calling the function. When in use, the `libxml_clear_errors` function, which takes no parameters and does not return a value, will clear all stored errors.

Errors issued from the libxml2 library are stored internally on a first-in, first-out (FIFO) basis. This means the first error in will be the first error out and will be accessed through a `LibXMLError` object. A `LibXMLError` object has no methods and has only the properties listed in Table 5-5.

Table 5-5. `LibXMLError` *Object Properties*

Property	Type	Description
`level`	int	Indicates the severity of the error. It is one of the levels defined by the libxml extension that includes `LIBXML_ERR_NONE`, `LIBXML_ERR_WARNING`, `LIBXML_ERR_ERROR`, and `LIBXML_ERR_FATAL`.
`code`	int	The error code from libxml2.
`column`	int	The column number if available from within the document the error occurred.
`line`	int	The line number if available from within the document the error occurred.
`message`	string	The textual representation of the error.
`file`	string	The filename, if available, of the XML document containing the error.

Not every property will be populated within a `LibXMLError` object. Certain values cannot always be determined, such as `file` when parsing a string containing an XML document.

You can access the errors through the `libxml_get_last_error` and `libxml_get_errors` functions. The `libxml_get_last_error` function returns the last `LibXMLError` object reported. This function is useful only if the last reported error is desired. One thing to note is that even when the new internal error handling is not enabled, this function is still available to access the last error issued from libxml2. The `libxml_get_errors` function returns an array of `LibXMLError` objects, starting with the first error issued and ending with the latest error. Modifying the SimpleXML code previously used, you can now suppress the error output while still having access to the XML errors:

```
libxml_use_internal_errors (TRUE);
if ($sxe = simplexml_load_string('<root>')) {
   print $sxe->asXML();
}
/* Was an error produced? */
if ($lasterror = libxml_get_last_error()) {
   /* Dump the last error reported */
   var_dump($lasterror);

   /* Get all errors as an array, loop through them, and dump the output */
   $arerrors = libxml_get_errors();
   foreach ($arerrors as $error) {
      var_dump($error);
   }

   /* Clear out the internal errors since they are no longer needed */
   libxml_clear_errors();
}
```

The code represented here assumes that this is the entire script or, if not, that internal errors have already been cleared. The test for errors was simply done using the `libxml_get_last_error` function. If anything was returned, then you know some type of error condition occurred. The check was not done using `libxml_get_errors`, because this function will always return an array, even when empty. If you used this function, you would need to execute the `count` function to find out whether there was at least one error in the array.

Conclusion

This chapter provided some background on the XML-related extensions in general, including the underlying libxml2 library. More important, however, is the common functionality from the libxml extension covered in this chapter. This functionality is not exclusive to any single extension but comes with any extension based on the libxml2 library. The libxml extension is a required extension, built statically within PHP, when using any of the XML extensions built with libxml2. libxml2 provides constants for use when parsing and provides access to the streams context when needed, and as of PHP 5.1, it handles and provides access to the new XML error-handling functionality.

With the knowledge of XML, many of its technologies, and the core libxml extension behind you, it is time to start looking at the parsers available in PHP 5. The first of these parsers is the tree-based DOM extension.