Using PHP with XML (part 1)

By icarus

Table of Contents

Haalp!

You've probably already heard about XML, Extensible Markup Language − it's been the subject of many water−cooler conversations over the past year and a half. You may even have seen an XML document in action, complete with user−defined tags and markup, and you've probably heard about the Holy Grail of the XML effort − using a single marked−up XML data source to generate complex documents in HTML, WML or other formats.

The strange thing about XML, though, is that it's very hard to figure out where to start. It doesn't take long to figure out the basics of creating an XML file, or the rules for XML markup. However, that's where most novice developers hit a brick wall; XML documentation often fails to clearly explain the logical next step, preferring instead to focus on the technical aspects of the language itself. The situation can be frustrating − imagine an automobile manual filled with technical details on the types of nuts and bolts used, rather than driver instructions and cautions − and the excessive jargon only adds to the overall confusion.

Over the next few pages, I will be attempting to rectify this a little bit, with an explanation of how you can use PHP to read your XML data and convert it into browser−readable HTML. I'll also spend some time explaining the different methods of parsing XML data, and the PHP implementation of each of these, together with a brief note on how all the pieces fit together.

I'll try and keep it simple − I'm going to use very simple XML sources, so you don't have to worry about namespaces, DTDs and PIs − although I will assume that you know the basic rules of XML markup, and of PHP scripting. And, of course, I'll assume that you have a sense of humour and will laugh at the appropriate places.

Getting Down To Business

Before we get into the nitty−gritty of XML parsing with PHP, I'd like to take some time to explain how all the pieces fit together.

In case you don't already know, XML is a markup language created to help document authors describe the data contained within a document. This description is accomplished by means of tags, very similar in appearance to regular HTML markup. However, where HTML depends on pre−defined tags, XML allows document authors to create their own tags, immediately making it more powerful and flexible. There are some basic rules to be followed when creating an XML file, and a file can only be processed if these rules are followed to the letter.

Once a file has been created, it needs to be converted, or "transformed", from pure data into something a little more readable. XSL, the Extensible Style Language, is typically used for such transformations; it's a powerful language that allows you to generate different output from the same XML data source. For example, you could use different XSL transformations to create an HTML Web page, a WML deck, and an ASCII text file...all from the same source XML.

There's only one problem here: most browsers don't come with an XML parser or an XSL processor. The latest versions of Internet Explorer and Netscape Gecko do support XML, but older versions don't. And this brings up an obvious problem: how do you use an XML data source to generate HTML for these older browsers

The solution is to insert an additional layer between the client and the server, which takes care of parsing the XML and returning the rendered output to the browser. And that's where PHP comes in − PHP4 supports XML parsing, through its DOM and XML extensions, and even includes an XSL processor, through its Sablotron extension.

Through this article, I'll limit my discussion to PHP's XML parsing functions; I will not be covering XSL transformations or the PHP−Sablotron extension. As you will see, there are two methods to parse XML data with PHP, and each one has advantages and disadvantages; I'll explain both approaches, together with simple examples to demonstrate how to use them in your own applications.

Let's Talk About SAX

The first of these approaches is SAX, the Simple API for XML. A SAX parser works by traversing an XML document and calling specific functions as it encounters different types of tags. For example, I might call a specific function to process a starting tag, another function to process an ending tag, and a third function to process the data between them.

The parser's responsibility is simply to parse the document; the functions it calls are responsible for processing the tags found. Once the tag is processed, the parser moves on to the next element in the document, and the process repeats itself.

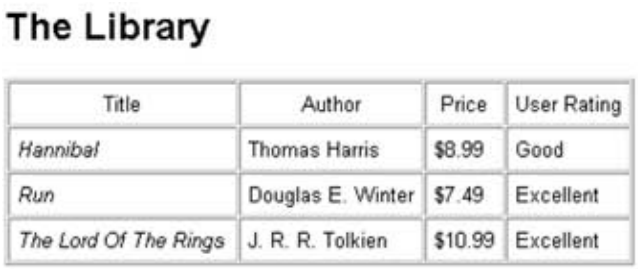
PHP comes with the expat parser, which you can also download from http://www.jclark.com/xml/. You may need to recompile your PHP binary with the "−−with−xml" parameter to activate XML support (Windows users get a pre−built binary with their distribution.)

I'll begin by putting together a simple XML file:

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

<library>
<book>
<title>Hannibal</title>
<author>Thomas Harris</author>
<genre>Suspense</genre>
<pages>564</pages>
<price>8.99</price>
<rating>4</rating>
</book>

<book>
<title>Run</title>
<author>Douglas E. Winter</author>
<genre>Thriller</genre>
<pages>390</pages>
<price>7.49</price>
<rating>5</rating>
</book>

<book>
<title>The Lord Of The Rings</title>
<author>J. R. R. Tolkien</author>
<genre>Fantasy</genre>
<pages>3489</pages>
<price>10.99</price>
<rating>5</rating>
```

```
</book>

</library>
```

Once my data is in XML−compliant format, I need to decide what I'd like the final output to look like. Let's say I want it to look like this:



The Library

Title	Author	Price	User Rating
Hannibal	Thomas Harris	$8.99	Good
Run	Douglas E. Winter	$7.49	Excellent
The Lord Of The Rings	J. R. R. Tolkien	$10.99	Excellent

As you can see, this is a simple table containing columns for the book title, author, price and rating. (I'm not using all the information in the XML file.) The title of the book is printed in italics, while the numerical rating is converted into something more readable.

Next, I'll write some PHP code to take care of this for me.

Breaking It Down

The first order of business is to initialize the XML parser, and set up the callback functions.

```
<?
// data file
$file = "library.xml";

// initialize parser
$xml_parser = xml_parser_create();

// set callback functions
xml_set_element_handler($xml_parser, "startElement",
"endElement");
xml_set_character_data_handler($xml_parser, "characterData");

// open XML file
if (!($fp = fopen($file, "r")))
{
die("Cannot locate XML data file: $file");
}

// read and parse data
while ($data = fread($fp, 4096))
{
// error handler
if (!xml_parse($xml_parser, $data, feof($fp)))
{
die(sprintf("XML error: %s at line %d",
xml_error_string(xml_get_error_code($xml_parser)),
xml_get_current_line_number($xml_parser)));
}
}

// clean up
xml_parser_free($xml_parser);

?>
```

The xml_parser_create() function is used to initialize the parser, and assign it to a handle, which may be used by subsequent function calls. Once the parser is initialized, the next step is to specify callback functions for the different types of tags.

The xml_set_element_handler() function is used to specify the functions to be executed when the parser

encounters the opening and closing tags of an element − in this case, the functions startElement() and endElement() respectively. The xml_set_character_data_handler() function specifies the function to be called when the parser encounters character data − in this case, characterData().

In addition, PHP also offers the xml_set_processing_instruction_handler(), for processing instructions; xml_set_unparsed_entity_decl_handler(), for unparsed entities; xml_set_external_entity_ref_handler(), for external entities; xml_set_notation_decl_handler(), for notation declarations; and xml_set_default_handler(), for all other entities within a document. In case you don't know what any of these are, don't worry about it − I'm not planning to use any of them here.

The next step in the script above is to open the XML file (as defined in the $file variable), read it and parse it via the xml_parse() function. The xml_parse() function will call the appropriate handling function each time it encounters a specific tag type. Once the document has been completely parsed, the xml_parser_free() function is called to free used memory and clean things up.

Errors can be displayed by means of the xml_error_string() function, which returns a description of the error encountered by the parser. Additional functions like xml_get_error_code(), xml_get_current_line_number(), xml_get_current_column_number(), and xml_get_current_byte_index() provide some additional information on the error.

Call Me Back

Now, the startElement(), endElement() and characterData() functions will be called by the parser as it progresses through the document. We haven't defined these yet − let's do that next:

```
<?
// use this to keep track of which tag the parser is currently
processing
$currentTag = "";

function startElement($parser, $name, $attrs) {
global $currentTag;
$currentTag = $name;

// output opening HTML tags
switch ($name) {
case "BOOK":
echo "<tr>";
break;

case "TITLE":
echo "<td>";
break;

case "AUTHOR":
echo "<td>";
break;

case "PRICE":
echo "<td>";
break;

case "RATING":
echo "<td>";
break;

default:
break;
}
}
?>
```

Each time the parser encounters a starting tag, it calls startElement() with the name of the tag (and attributes, if any) as arguments. The startElement() function then processes the tag, printing corresponding HTML

markup in place of the XML tag.

I've used a "switch" statement, keyed on the tag name, to decide how to process each tag. For example, since I know that <book> indicates the beginning of a new row in my desired output, I replace it with a <tr>, while other elements like <title> and <author> correspond to table cells, and are replaced with <td> tags.

Finally, I've also stored the current tag name in the global variable $currentTag − this can be used to identify which tag is being processed at any stage, and it'll come in useful a little further down.

The endElement() function takes care of closing tags, and looks similar − note that I've specifically cleaned up $currentTag at the end.

```
<?
function endElement($parser, $name) {
global $currentTag;

// output closing HTML tags
switch ($name) {
case "BOOK":
echo "</tr>";
break;

case "TITLE":
echo "</td>";
break;

case "AUTHOR":
echo "</td>";
break;

case "PRICE":
echo "</td>";
break;

case "RATING":
echo "</td>";
break;

default:
break;
}

// clear current tag variable
$currentTag = "";
}
?>
```

So this takes care of replacing XML tags with corresponding HTML tags...but what about handling the data between them?

```
<?
// process data between tags
function characterData($parser, $data) {

global $currentTag;
// text ratings
$ratings = array("Words fail me!", "Terrible", "Bad",
"Indifferent",
"Good", "Excellent");

// format the data
switch ($currentTag) {
case "TITLE":
// italics for title
echo "<i>$data</i>";
break;

case "AUTHOR":
echo $data;
break;

case "PRICE":
// add currency symbol for price
echo "$" . $data;
break;

case "RATING":
// get text rating
echo $ratings[$data];
break;

default:
break;
}
}
?>
```

The characterData() function is called whenever the parser encounters data between an XML tag pair. Note, however, that the function is only passed the data as argument; there is no way of telling which tags are around it. However, since the parser processes XML chunk−by−chunk, we can use the $currentTag variable

to identify which tag this data belongs to.

Depending on the value of $currentTag, a "switch" loop is used to print data with appropriate formatting; this is the place where I add italics to the title, a currency symbol to the price, and a text rating (corresponding to a numerical index) from the $ratings array.

Here's what the finished script, with some additional HTML, looks like:

```
<html>
<head>
<title>The Library</title>
<style type="text/css">
TD {font-family: Arial; font-size: smaller}
H2 {font-family: Arial}
</style>
</head>
<body bgcolor="white">
<h2>The Library</h2>
<table border="1" cellspacing="1" cellpadding="5">
<tr>
<td align=center>Title</td>
<td align=center>Author</td>
<td align=center>Price</td>
<td align=center>User Rating</td>
</tr>

<?
// data file
$file = "library.xml";

// use this to keep track of which tag the parser is currently
processing
$currentTag = "";

function startElement($parser, $name, $attrs) {
global $currentTag;
$currentTag = $name;

// output opening HTML tags
switch ($name) {
case "BOOK":
echo "<tr>";
break;

case "TITLE":
echo "<td>";
```

```
break;

case "AUTHOR":
echo "<td>";
break;

case "PRICE":
echo "<td>";
break;

case "RATING":
echo "<td>";
break;

default:
break;
}
}

function endElement($parser, $name) {
global $currentTag;

// output closing HTML tags
switch ($name) {
case "BOOK":
echo "</tr>";
break;

case "TITLE":
echo "</td>";
break;

case "AUTHOR":
echo "</td>";
break;

case "PRICE":
echo "</td>";
break;

case "RATING":
echo "</td>";
break;

default:
break;
}

// clear current tag variable
```

```
$currentTag = "";
}

// process data between tags
function characterData($parser, $data) {

global $currentTag;
// text ratings
$ratings = array("Words fail me!", "Terrible", "Bad",
"Indifferent",
"Good", "Excellent");

// format the data
switch ($currentTag) {
case "TITLE":
// italics for title
echo "<i>$data</i>";
break;

case "AUTHOR":
echo $data;
break;

case "PRICE":
// add currency symbol for price
echo "$" . $data;
break;

case "RATING":
// get text rating
echo $ratings[$data];
break;

default:
break;
}
}

// initialize parser
$xml_parser = xml_parser_create();

// set callback functions
xml_set_element_handler($xml_parser, "startElement",
"endElement");
xml_set_character_data_handler($xml_parser, "characterData");

// open XML file
if (!($fp = fopen($file, "r")))
{
```

```
die("Cannot locate XML data file: $file");
}

// read and parse data
while ($data = fread($fp, 4096))
{
// error handler
if (!xml_parse($xml_parser, $data, feof($fp)))
{
die(sprintf("XML error: %s at line %d",
xml_error_string(xml_get_error_code($xml_parser)),
xml_get_current_line_number($xml_parser)));
}
}

// clean up
xml_parser_free($xml_parser);

?>

</table>
</body>
</html>
```

And when you run it, here's what you'll see:

The Library

Title	Author	Price	User Rating
Hannibal	Thomas Harris	$8.99	Good
Run	Douglas E. Winter	$7.49	Excellent
The Lord Of The Rings	J. R. R. Tolkien	$10.99	Excellent

You can now add new items to your XML document, or edit existing items, and your rendered HTML page will change accordingly. By separating the data from the presentation, XML has imposed standards on data collections, making it possible, for example, for users with no technical knowledge of HTML to easily update content on a Web site, or to present data from a single source in different ways.

What's For Dinner?

Here's another, slightly more complex example using the SAX parser, and one of my favourite meals.

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

<recipe>

<name>Chicken Tikka</name>
<author>Anonymous</author>
<date>1 June 1999</date>

<ingredients>

<item>
<desc>Boneless chicken breasts</desc>
<quantity>2</quantity>
</item>

<item>
<desc>Chopped onions</desc>
<quantity>2</quantity>
</item>

<item>
<desc>Ginger</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Garlic</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Red chili powder</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Coriander seeds</desc>
<quantity>1 tsp</quantity>
</item>

<item>
```

```
<desc>Lime juice</desc>
<quantity>2 tbsp</quantity>
</item>

<item>
<desc>Butter</desc>
<quantity>1 tbsp</quantity>
</item>
</ingredients>

<servings>
3
</servings>

<process>
<step>Cut chicken into cubes, wash and apply lime juice and
salt</step>
<step>Add ginger, garlic, chili, coriander and lime juice in a
separate
bowl</step>
<step>Mix well, and add chicken to marinate for 3-4
hours</step>
<step>Place chicken pieces on skewers and barbeque</step>
<step>Remove, apply butter, and barbeque again until meat is
tender</step>
<step>Garnish with lemon and chopped onions</step>
</process>

</recipe>
```

This time, my PHP script won't be using a "switch" statement when I parse the file above; instead, I'm going to be keying tag names to values in a hash. Each of the tags in the XML file above will be replaced with appropriate HTML markup.

```
<html>
<head>
</head>
<body bgcolor="white">

<?
// data file
$file = "recipe.xml";

/*
hash of tag names mapped to HTML markup
```

```
"RECIPE" => start a new block
"NAME" => in bold
"INGREDIENTS" => unordered list
"DESC" => list items
"PROCESS" => ordered list
"STEP" => list items
*/
$startTags = array(
"RECIPE" => "<hr>",
"NAME" => "<font size=+2>",
"DATE" => "<i>(",
"AUTHOR" => "<b>",
"SERVINGS" => "<i>Serves ",
"INGREDIENTS" => "<h3>Ingredients:</h3><ul>",
"DESC" => "<li>",
"QUANTITY" => "(",
"PROCESS" => "<h3>Preparation:</h3><ol>",
"STEP" => "<li>"
);

// close tags opened above
$endTags = array(
"NAME" => "</font><br>",
"DATE" => ")</i>",
"AUTHOR" => "</b>",
"INGREDIENTS" => "</ul>",
"QUANTITY" => ")",
"SERVINGS" => "</i>",
"PROCESS" => "</ol>",
);

function startElement($parser, $name, $attrs) {
global $startTags;
// if tag exists as key, print value
if ($startTags[$name]) { echo $startTags[$name]; }
}

function endElement($parser, $name) {
global $endTags;
if ($endTags[$name]) { echo $endTags[$name]; }
}

// process data between tags
function characterData($parser, $data) {
echo $data;
}

// initialize parser
$xml_parser = xml_parser_create();
```

```
// set callback functions
xml_set_element_handler($xml_parser, "startElement",
"endElement");
xml_set_character_data_handler($xml_parser, "characterData");

// open XML file
if (!($fp = fopen($file, "r")))
{
die("Cannot locate XML data file: $file");
}

// read and parse data
while ($data = fread($fp, 4096))
{
// error handler
if (!xml_parse($xml_parser, $data, feof($fp)))
{
die(sprintf("XML error: %s at line %d",
xml_error_string(xml_get_error_code($xml_parser)),
xml_get_current_line_number($xml_parser)));
}
}

// clean up
xml_parser_free($xml_parser);

?>

</body>
</html>
```

In this case, I've set up two hashes (associative arrays), one for opening tags and one for closing tags. When the parse encounters an XML tag, it looks up the array to see if the tag exists as a key. If it does, the corresponding value (HTML markup) is printed. This method does away with the slightly cumbersome "switch" statements of the previous example, and is easier to read and understand.

Here's the output:

Chicken Tikka

Anonymous *(1 June 1999)*

Ingredients:

- Boneless chicken breasts (2)
- Chopped onions (2)
- Ginger (1 tsp)
- Garlic (1 tsp)
- Red chili powder (1 tsp)
- Coriander seeds (1 tsp)
- Lime juice (2 tbsp)
- Butter (1 tbsp)

Serves 3

Preparation:

1. Cut chicken into cubes, wash and apply lime juice and salt
2. Add ginger, garlic, chili, coriander and lime juice in a separate bowl
3. Mix well, and add chicken to marinate for 3-4 hours
4. Place chicken pieces on skewers and barbeque
5. Remove, apply butter, and barbeque again until meat is tender
6. Garnish with lemon and chopped onions

That's about it for the moment. Over the last few pages, I've discussed using PHP's built−in SAX parser to process an XML file and mark up the data within it with HTML tags. However, just as there's more than one way to skin a cat, there's more than one way to process XML data in PHP. In the second part of this article, I'll be looking at an alternative technique of parsing an XML file, this time using the DOM. Make sure you come back for that one!