Using PHP with XML (part 2)

By icarus

Table of Contents

Climbing A Tree

In the first part of this article, I said that there were two approaches to parsing an XML document. SAX is one; the other uses the DOM to build a tree representation of the XML data structures in the document, and then offers built−in methods to navigate through this tree. Once a particular node has been reached, built−in properties can be used to obtain the value of the node, and use it within the script.

PHP4 comes with support for this "DOM parser" as well, although you may need to recompile your PHP binary with the "−−with−dom" parameter to activate it. Windows users get a free ride, since a compiled binary is included with their distribution; however, they will still need to activate the extension in the "php.ini" configuration file.

Over the next few pages, I'm going to take a look at PHP's DOM functions, illustrating an alternative method of parsing XML data. I'll also attempt to duplicate the previous examples, marking up books and recipes with HTML tags, in order to demonstrate how either technique can be used to achieve the same result. Finally, I'll briefly discuss the pros and cons of the SAX and DOM approaches, and point you to some links for your further education.

Let's get started, shall we?

Meet Joe Cool

As before, the first order of business is to read the XML data into memory. PHP offers three functions to do this: the xmldoc() and xmltree() functions accept a string containing XML data as argument, and build a tree structure from that string, while the xmldocfile() function accepts a filename as argument, and builds a DOM tree after reading the data in the file.

```
<?

// create an XML-compliant string
$XMLDataString = "<?xml version=\"1.0\"?" . "><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

// create a document object
$XMLDoc = xmldoc($XMLDataString);
// create a tree object
$XMLTree = xmltree($XMLDataString);

?>
```

You can also load a file directly.

```
<?

// data file
$XMLFile = "me.xml";
// create a document object
$XMLDoc = xmldocfile($XMLFile);

?>
```

Windows users should note that xmldocfile() needs the complete path to the file, including drive letter, to work correctly.

Parents And Their Children

Once the document has been read in, a number of methods become available to traverse the document tree.

If you're using a document object, such as the one returned by xmldoc() and xmldocfile(), you can use the following methods to obtain references to other nodes in the tree,

```
<?

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

// create a document object
$dom = xmldocfile($file);

// echo these values to see the object type
// get root node
$dom->root();

// get children under the root node as array
$dom->children();

?>
```

or to properties of the document itself.

```
<?

// data file
$file = "library.xml";
// create a document object
$dom = xmldocfile($file);

// get XML version
echo $dom->version;

// get XML encoding
echo $dom->encoding;

// get whether standalone file
echo $dom->standalone;
```

```
// get XML URL
echo $dom->url;

// get XML character set
echo $dom->charset;

?>
```

Once you've obtained a reference to a node, a number of other methods and properties become available to help you obtain the name and value of that node, as well as references to parent and child nodes. Take a look:

```
<?

// data file
$str = "<?xml version=\"1.0\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";
// create a document object
$dom = xmldoc($str);

// get reference to root node
$root = $dom->root();

// get name of node - "me"
echo $root->name;

// get children of node, as array
$children = $root->children();

// get first child node
$firstChild = $children[0];

// let's see a few node properties
// get name of first child node - "name"
echo $firstChild->name;

// get content of first child node - "Joe Cool"
echo $firstChild->content;

// get type of first child node - "1", or "XML_ELEMENT_NODE"
echo $firstChild->type;

// go back up the tree!
// get parent of child - this should be <me>
$parentNode = $firstChild->parent();
```

```
// check it...yes, it is "me"!
echo $parentNode->name;
?>
```

A quick note on the "type" property above: every node is of a specific type, and this property returns a numeric and textual code corresponding to the type. A complete list of types is available in the PHP manual.

Welcome To The Human Race

Many of the object methods you just saw are also available as regular functions; for example,

```
<?

// data file
$str = "<?xml version=\"1.0\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";
// create a document object
$dom = xmldoc($str);

// get reference to root node
$root = $dom−>root();

// get name of node − "me"
echo $root−>name;

// you could also do this!
// get reference to root node
$root = domxml_root($dom);

// get name of node − "me"
echo $root−>name;

?>
```

Finally, element attributes may be obtained using either the $node−>attributes() object method, or the dom_attributes() function.

```
<?

// data file
$str = "<?xml version=\"1.0\"?><me species=\"human\"><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";
// create a document object
$dom = xmldoc($str);

// get reference to root node
$root = $dom−>root();
```

```
// get attribute collection
// $attributes = $root->attributes();

// you could also do this!
// $attributes = domxml_attributes($root);

// get first attribute name - "species"
echo $attributes[0]->name;

// get first attribute value - "human"
echo $attributes[0]->children[0]->content;

// you could also do this!
// echo domxml_getattr($root, "species");
?>
```

Building A Library

Using this information, it's pretty easy to re−create our first example using the DOM parser. Here's the XML data,

```
<?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>
```

and here's the script which does all the work.

```
<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>

<?

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

// data file
$file = "library.xml";
// create a document object
$dom = xmldocfile($file);

// get reference to root node
$root = $dom->root();
// array of root node's children - the <book> level
$nodes = $root->children();

// iterate through <book>s
for ($x=0; $x<sizeof($nodes); $x++)
{
// new row
echo "<tr>";

// check type
// this is to correct whitespace (empty nodes)
if ($nodes[$x]->type == XML_ELEMENT_NODE)
{
$thisNode = $nodes[$x];
// get an array of this node's children - the <title>,
<author> level
$childNodes = $thisNode->children();

// iterate through children
for($y=0; $y<sizeof($childNodes); $y++)
{
// check type again
if ($childNodes[$y]->type == XML_ELEMENT_NODE)
{
// appropriate markup for each type of tag
// like a switch statement
if ($childNodes[$y]->name == "title")
{
echo "<td><i>" . $childNodes[$y]->content . "</i></td>";
}

if ($childNodes[$y]->name == "author")
{
echo "<td>" . $childNodes[$y]->content . "</td>";
}

if ($childNodes[$y]->name == "price")
{
echo "<td>$" . $childNodes[$y]->content . "</td>";
}

if ($childNodes[$y]->name == "rating")
{
echo "<td>" . $ratings[$childNodes[$y]->content] . "</td>";
}

}
}

}
// close the row tags
echo "</tr>";
}


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

This may appear complex, but it isn't really all that hard to understand. I've first obtained a reference to the root of the document tree, $root, and then to the "children" of that root; these children are structured as elements of a regular PHP array. I've then used a "for" loop to iterate through the array, navigate to the next level, and print the content found in the nodes, with appropriate formatting.

The numerous "if" loops you see are needed to check the name of each node, and format it appropriately; in fact, they're equivalent to the "switch" statements used in the previous article.

Anyone For Chicken?

I can do the same thing with the second example as well. However, since there are quite a few levels to the document tree, I've decided to use a recursive function to iterate through the tree, rather than a series of "if" statements.

Here's the XML file,

```
<?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>
```

and here's the script which parses it.

```
<html>
<head>
</head>
<body bgcolor="white">
<hr>
<?
// data file
$file = "recipe.xml";
```

```
// hash for HTML markup
$startTags = array(
"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>"
);

$endTags = array(
"name" => "</font><br>",
"date" => ")</i>",
"author" => "</b>",
"servings" => "</i>",
"ingredients" => "</ul>",
"quantity" => ")",
"process" => "</ol>",
);

// create a document object
$dom = xmldocfile($file);

// get reference to root node
$root = $dom->root();

// get children
$children = $root->children();

// run a recursive function starting here
printData($children);

// this function accepts an array of nodes as argument,
// iterates through it and prints HTML markup for each tag it
finds.
// for each node in the array, it then gets an array of the
node's
children, and
// calls itself again with the array as argument (recursion)
function printData($nodeCollection)
{
global $startTags, $endTags;
// iterate through array
for ($x=0; $x<sizeof($nodeCollection); $x++)
{
// print HTML opening tags and node content
echo $startTags[$nodeCollection[$x]->name] .
$nodeCollection[$x]->content;
// get children and repeat
$dummy = getChildren($nodeCollection[$x]);
printData($dummy);
// once done, print closing tags
echo $endTags[$nodeCollection[$x]->name];
}

}

// function to return an array of children, given a parent
node
function getChildren($node)
{
$temp = $node->children();
$collection = array();
$count=0;
// iterate through children array
for ($x=0; $x<sizeof($temp); $x++)
{
// filter out the empty nodex
// and create a new array
if ($temp[$x]->type == XML_ELEMENT_NODE)
{
$collection[$count] = $temp[$x];
$count++;
}
}
// return array containing child nodes
return $collection;
}

?>
</body>
</html>
```

In this case, I've utilized a slightly different method to mark up the XML. I've first initialized a couple of associative arrays to map XML tags to corresponding HTML markup, in much the same manner as I did last time. Next, I've used DOM functions to obtain a reference to the first set of child nodes in the DOM tree.

This initial array of child nodes is used to "seed" my printData() function, a recursive function which takes an array of child nodes, matches their tag names to values in the associative arrays, and outputs them to the browser. It also obtains a reference to the next set of child nodes, via the getChildren() function, and calls itself with the new node collection as argument.

By using this recursive function, I've managed to substantially reduce the number of "if" conditional statements in my script; the code is now easier to read, and also structured more logically.

Conclusions...

As you can see, you can parse a document using either DOM or SAX, and achieve the same result. The difference is that the DOM parser is a little slower, since it has to build a complete tree of the XML data, whereas the SAX parser is faster, since it's calling a function each time it encounters a specific tag type. You should experiment with both methods to see which one works better for you.

There's another important difference between the two techniques. The SAX approach is event−centric − as the parser travels through the document, it executes specific functions depending on what it finds. Additionally, the SAX approach is sequential − tags are parsed one after the other, in the sequence in which they appear. Both these features add to the speed of the parser; however, they also limit its flexibility in quickly accessing any node of the DOM tree.

As opposed to this, the DOM approach builds a complete tree of the document in memory, making it possible to easily move from one node to another (in a non−sequential manner.) Since the parser has the additional overhead of maintaining the tree structure in memory, speed is an issue here; however, navigation between the various "branches" of the tree is easier. Since the approach is not dependent on events, developers need to use the exposed methods and attributes of the various DOM objects to process the XML data.

...And Links

That just about concludes this little tour of parsing XML data with PHP. I've tried to keep it as simple as possible, and there are numerous aspects of XML I haven't covered here. If you're interested in learning more about XML and XSL, you should visit the following links:

You can read more about the DOM, and its different incarnations, at:

The W3C's DOM specification, at http://www.w3.org/DOM/

A JavaScript view of the DOM, at http://www.melonfire.com/community/columns/trog/article.php3?id=58

The PHP manual pages, with user comments, are a great online resource, and you'll often find answers to common problems there. Even if you don't, they're well worth bookmarking:

PHP's DOM XML functions, at http://www.php.net/manual/en/ref.domxml.php

PHP's XML parsing functions, at http://www.php.net/manual/en/ref.xml.php

A number of developers have built and released free PHP classes to handle XML data − if you're ever on a tight deadline, using these classes might save you some development time. Take a look at the following links for more information:

PHPXML, at http://www.phpxml.org/

PhpDOM, at http://devil.medialab.at/

And that's just about all for the moment. I hope you found this interesting, and that it helped to make the road ahead a little clearer. If you'd like to know more about PHP and XML, write in and tell me what you'd like to read about...and, until next time, stay healthy!