# PHP 101 (part 11): Sinfully Simple

## Easy Peasy

Unless you've been hiding in a cave for the last few years, you've heard about XML – it's the toolkit that more and more Web publishers are switching to for content markup. You may
even have seen an XML document in action, complete with user-defined tags and markup, and
you might have wondered how on earth one converts that tangled mess of code into
human-readable content.

The answer is, not easily.

While PHP has included support for the two standard methods of parsing (read: making sense
of) XML – SAX and DOM – since version 4.0, the complexity and inherent geekiness of these
methods often turned off all but the most dedicated XML developers. All that has changed,
however, with PHP 5.0, which introduces a brand-spanking-new XML extension named SimpleXML that takes all (and I do mean all) the pain out of processing XML documents. Keep reading, and find out how.

## The Bad Old Days

In order to understand why SimpleXML is so cool, a brief history lesson is in order.

In the days before SimpleXML, there were two ways of processing XML documents. The first,
**SAX** or the Simple API for XML, involved traversing an XML document and calling

specific functions as the parser encountered different types of tags. For example, you might
have called one function to process a starting tag, another function to process an ending tag,
and a third function to process the data between them. The second, **DOM** or the Document
Object Model, involved creating a tree representation of the XML document in memory, and then
using tree-traversal methods to navigate it. Once a particular node of the tree was reached,
the corresponding content could be retrieved and used.

Neither of these two approaches was particularly user-friendly: SAX required the developer
to custom-craft event handlers for each type of element encountered in an XML file, while
the DOM approach used an object-oriented paradigm which tended to throw developers off, in
addition to being memory-intensive and thus inefficient with large XML documents. In the
larger context also, PHP 4 used a number of different backend libraries for each of its
different XML extensions, leading to inconsistency in the way different XML extensions
worked and thus creating interoperability concerns (as well as a fair amount of confusion
for developers).

With PHP 5.0, a concerted effort was made to fix this problem, by adopting the libxml2

library (http://www.xmlsoft.org/)

as the standard library for all XML extensions and by getting the various XML

extensions

to operate more consistently. The biggest change in the PHP 5 XML pantheon,

though, is

the SimpleXML extension developed by Sterling Hughes, Rob Richards and Marcus

Börger,

which attempts to make parsing XML documents significantly more user-friendly

than it

was in PHP 4.

SimpleXML works by converting an XML document into an object, and then

turning the

elements within that document into object properties which can be accessed

using standard

object notation. This makes it easy to drill down to an element at any level of the

XML

hierarchy to access its content. Repeated elements at the same level of the

document tree

are represented as arrays, while custom element collections can be created using

XPath

location paths (of which, more later); these collections can then be processed

using PHP's

standard loop constructs. Accessing element attributes is as simple as accessing

the keys

of an associative array – there's nothing new to learn, and no special code to

write.

In order to use SimpleXML and PHP together, your PHP build must include support for
SimpleXML. This support is enabled by default in both the UNIX and Windows versions of
PHP 5. Read more about this at http://www.php.net/manual/en/ref.simplexml.php. If you’re a PHP 4
user, you’re out of luck – SimpleXML is only available for PHP 5.

## Petting Zoo

To see how SimpleXML works, consider the following XML file:

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

    <name>Polly Parrot</name>

    <age>3</age>

    <species>parrot</species>

    <parents>

        <mother>Pia Parrot</mother>
              <father>Peter Parrot</father>
    </parents>

</pet>
```

Now, you need a way to get to the content enclosed between the `<name>`, `<age>`, `<species>` and `<parents>` elements. With SimpleXML, it’s a snap:

```
<?php
// set name of XML file

$file = "pet.xml";
// load file

$xml = simplexml_load_file($file) or die ("Unable to load XML file!");
// access XML data

echo "Name: " . $xml->name . "\n";

echo "Age: " . $xml->age . "\n";
echo "Species: " . $xml->species . "\n";

echo "Parents: " . $xml->parents->mother . " and " .  $xml->parents-
>father . "\n";
```

```
?>
```

The action begins with the `simplexml_load_file()` function, which accepts

the path and name of the XML file to be parsed. The result of parsing the file is a

PHP object, whose properties correspond to the elements under the root element.

The

character data within an element can then be accessed using standard

`object->property` notation, beginning with the root element and moving

down the hierarchical path of the document.

Just as you can read, so also can you write. SimpleXML makes it easy to alter the

contents of a particular XML element – simply assign a new value to the

corresponding

object property. Here’s an example:

```
<?php
// set name of XML file

$file = "pet.xml";
// load file

$xml = simplexml_load_file($file) or die ("Unable to load XML file!");
// modify XML data

$xml->name = "Sammy Snail";

$xml->age = 4;
```

```
$xml->species = "snail";
$xml->parents->mother = "Sue Snail";

$xml->parents->father = "Sid Snail";
// write new data to file

file_put_contents($file, $xml->asXML());
?>
```

Here, the original XML file is first read in, and then the character data enclosed within
each element is altered by assigning new values to the corresponding object property. The
`asXML()` method, typically used to dump the XML tree back out to the standard output device, is in this instance combined with the `file_put_contents()` function to overwrite the original XML document with the new data.

## Sin City

Repeated elements at the same level of the XML hierarchy are represented as array elements,
and can be accessed using numeric indices. To see how this works, consider the following XML file:

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

```
    <sin>pride</sin>

    <sin>envy</sin>

    <sin>anger</sin>

    <sin>greed</sin>
        <sin>sloth</sin>

    <sin>gluttony</sin>

    <sin>lust</sin>

</sins>
```

Here’s the PHP script that reads it and retrieves the data from it:

```
<?php
// set name of XML file

$file = "sins.xml";
// load file
```

```
$xml = simplexml_load_file($file) or die ("Unable to load XML file!");
// access each <sin>

echo $xml->sin[0] . "\n";

echo $xml->sin[1] . "\n";
echo $xml->sin[2] . "\n";

echo $xml->sin[3] . "\n";
echo $xml->sin[4] . "\n";

echo $xml->sin[5] . "\n";
echo $xml->sin[6] . "\n";
?>
```

If you'd prefer, you can even iterate over the collection with a `foreach()` loop, as in this next, equivalent listing:

```
<?php
// set name of XML file

$file = "sins.xml";
// load file

$xml = simplexml_load_file($file) or die ("Unable to load XML file!");
// iterate over <sin> element collection
```

```
foreach ($xml->sin as $sin) {

    echo "$sin\n";

}
?>
```

## The Shape Of Things To Come

SimpleXML handles element attributes as transparently as it does elements and their
content. Attribute-value pairs are represented as members of a PHP associative array,
and can be accessed like regular array elements. To see how this works, take a look
at this script:

```
<?php
// create XML string
$str = <<< XML  //以下內容會被寫进 str 中
<?xml version="1.0"?>
<shapes>
    <shape type="circle" radius="2" />
    <shape type="rectangle" length="5" width="2" />
    <shape type="square" length="7" />
</shapes>
XML;
```

```
// load string
$xml = simplexml_load_string($str) or die ("Unable to load XML string!");
// for each shape
// calculate area
foreach ($xml->shape as $shape) {
    if ($shape['type'] == "circle") {
        $area = pi() * $shape['radius'] * $shape['radius'];
        }
    elseif ($shape['type'] == "rectangle") {
        $area = $shape['length'] * $shape['width'];
        }
    elseif ($shape['type'] == "square") {
        $area = $shape['length'] * $shape['length'];
        }
    echo $area."\n";
}
?>
```

Unlike previous examples, which used an external XML file, this one creates the XML

dynamically and loads it into SimpleXML with the `simplexml_load_string()`

method. The XML is then parsed with a `foreach()` loop, and the area for

each shape calculated on the basis of the value of each `<shape>`

element's type attribute. The listing above demonstrates how attribute values can

be accessed as keys of the attribute array associated with each element property.

## X Marks The Spot

SimpleXML also supports custom element collections, through XPath location paths. For
those of you new to XML, **XPath** is a standard addressing mechanism for an XML document, allowing developers to access collections of elements, attributes or text
nodes within a document. Read more about XPath
at http://www.w3.org/TR/xpath.html and
http://www.melonfire.com/community/columns/trog/article.php?id=83.

To see how this works, consider the following XML document:

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

<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</quantity>

    </item>
       <item>

        <desc>Garlic</desc>

        <quantity>1</quantity>

    </item>

    <item>
```

```
        <desc>Red chili powder</desc>
               <quantity>1</quantity>

    </item>

    <item>

        <desc>Coriander seeds</desc>

        <quantity>1</quantity>

    </item>
       <item>

        <desc>Lime juice</desc>

        <quantity>2</quantity>

    </item>
```

```
</ingredients>
```

Now, let's suppose you want to print all the `<desc>` elements. You

could do it by iterating over the array of `<item>` elements, as

discussed earlier…or you could just create a custom collection of only the

`<desc>` elements with the `xpath()` method, and

iterate over that instead:

```
<?php
// set name of XML file

$file = "ingredients.xml";
// load file

$xml = simplexml_load_file($file) or die ("Unable to load XML file!");
// get all the <desc> elements and print

foreach ($xml->xpath('//desc') as $desc) {

    echo "$desc\n";

}
?>
```

Using XPath, you can get even fancier than this – for example, by creating a

collection

of only those `<desc>` elements whose corresponding quantities are two or more.

```
<?php
// set name of XML file
$file = "ingredients.xml";
// load file

$xml = simplexml_load_file($file) or die ("Unable to load XML file!");
// get all the <desc> elements and print
foreach ($xml->xpath('//item[quantity > 1]/desc') as $desc) {

    echo "$desc\n";

}
?>
```

Without XPath, accomplishing this would be far more complicated than the five lines of code
above...try it for yourself and see!

## An Evening At The Moulin Rouge

Now that you've seen what XPath can do, let's wrap this up with an example of how you
might actually use it. Let's suppose you have a bunch of movie reviews marked up in XML,
like this:

```
<?xml version="1.0"?>
<review id="57" category="2">

    <title>Moulin Rouge</title>

    <teaser>

        Baz Luhrmann's over-the-top vision of Paris at the turn of the century

        is witty, sexy...and completely unforgettable

    </teaser>

    <cast>

        <person>Nicole Kidman</person>
                <person>Ewan McGregor</person>

        <person>John Leguizamo</person>

        <person>Jim Broadbent</person>
        <person>Richard Roxburgh</person>

    </cast>
        <director>Baz Luhrmann</director>

    <duration>120</duration>

    <genre>Romance/Comedy</genre>

    <year>2001</year>

    <body>
              A stylishly spectacular extravaganza, Moulin Rouge is hard to

        categorize; it is, at different times, a love story, a costume drama,

        a musical, and a comedy. Director Baz Luhrmann (well-known for the

        very hip William Shakespeare's Romeo + Juliet) has taken some simple

        themes - love, jealousy and obsession - and done something completely
        new and different with them by setting them to music.

    </body>

    <rating>5</rating>

</review>
```

Now, you want to display this review on your Web site. So, you need a PHP script to
extract the data from this file and place it in the appropriate locations in an HTML template. With everything you've learned so far, this is a snap…as the code below illustrates:

```
<?php
// set name of XML file

// normally this would come through GET

// it's hard-wired here for simplicity

$file = "57.xml";
// load file
```

```
$xml = simplexml_load_file($file) or die ("Unable to load XML file!");

?>


<html>


<head><basefont face="Arial"></head>

<body>

<!-- title and year -->


<h1><?php echo $xml->title; ?> (<?php echo $xml->year; ?>)</h1>

<!-- slug -->


<h3><?php echo $xml->teaser; ?></h3>

<!-- review body -->


<?php echo $xml->body; ?>




<!-- director, cast, duration and rating -->


<p align="right"/>


<font size="-2">
```

```
Director: <b><?php echo $xml->director; ?></b>
<br />


Duration: <b><?php echo $xml->duration; ?> min</b>


<br />
Cast: <b><?php foreach ($xml->cast->person as $person) { echo "$person "; } ?
></b>
<br />


Rating: <b><?php echo $xml->rating; ?></b>


</font>
</body>
</html>
```

Pretty simple, huh?

That’s about all for the moment. In
Part Twelve of PHP 101, I’ll be telling you all
about the new exception handling model in PHP 5, showing you how you can use it to
catch your scripts before they crash and burn. See you there!