Templates, The PHPLIB Way

David Orr

If you're wondering what templates are, first go read the first few paragraphs of Sascha Schumann's excellent article Templates - why and how to use them in PHP3. In general, templates let you completely separate your PHP code from the HTML, which makes the HTML graphic designers very happy and keeps them from messing up your precious programming.

It's Not FastTemplates

So, do we really need another article on PHPBuilder about using templates? Well, yes, because there is more than one way to do templates with PHP. Sascha's article talks about using FastTemplates, but the PHP Base Library ("PHPLIB") has its own implementation of templates.

So how are they different? FastTemplates was originally a Perl library that was ported to PHP. FastTemplates works well for Perl programs, but it's not ideal for PHP. Kristian Koehntopp wrote PHPLIB Template from the ground up as a pure PHP library to better take advantage of the capabilities of PHP. One advantage to Kristian's design is that it parses templates with `preg_replace()`, which is said to be faster than FastTemplate's reliance on `ereg_replace()`. Another advantage of PHPLIB Template is it allows dynamic blocks to be nested, unlike FastTemplates.

Both libraries do have a very similar set of features and capabilities, but if you are already using FastTemplates and you want to learn to use PHPLIB Template, you should start by trying to forget everything you know about using FastTemplates. Their features may be similar, but PHPLIB Template does everything just a little differently than FastTemplates.

Templates, The PHPLIB Way

Using PHPLIB Template

Let's start with a cheesy example with some sample code. We'll assume that there is a template in the /home/mydir/mytemplates/ named MyTemplate.ihtml that has some text that reads something like this:

Congratulations! You won a new {some_color} Honda Prelude!

Notice that "{some_color}" has curly braces around it. The curly braces indicate that some_color is a template variable. We may want to write a PHP script that will load the template, insert the value of the PHP variable `$my_color` where the {some_color} template variable tag is, and then output the new text. If $my_color happens to be set to "blue", the final output should look like:

Congratulations! You won a new blue Honda Prelude!

Here's the PHP script that will do just that:

```
<?php

include "template.inc";

$my_color = "blue"; // we'll use this later
```

```
$t = new Template("/home/mydir/mytemplates/");
    // create a template object named $t
$t->set_file("MyFileHandle","MyTemplate.ihtml");
    // set MyFileHandle = our template file
$t->set_var("some_color",$my_color);
    // set template variable some_color = $my_color value
$t->parse("MyOutput","MyFileHandle");
    // set template variable MyOutput = parsed file
$t->p("MyOutput"); // output the value of MyOutput (our parsed
data)

?>
```

The first line is an include command to give you PHPLIB Template functionality. Of course PHPLIB does a lot more than templates, but if you only want to use the template feature just include template.inc (template.inc is one of the files that comes with PHPLIB). PHPLIB Template uses object-oriented programming, so the next thing we do is create a Template object. The code `<? $t = new Template("/home/mydir/mytemplates/"); ?>` creates a new Template object `$t`. This `$t` object will be your handle for accessing all of the template functions for the rest of the PHP script. If you wanted to, you could create additional Template objects (each with their out template variable namespace), but one is usually all you need. The path ("/home/mydir/mytemplates/") in the Template constructor call sets the root path where your templates are located, but if you leave it out it defaults to the same directory as your PHP script.
Then we call `set_file()` to define a handle "MyFileHandle" linked to MyTemplate.ihtml (the template isn't actually loaded until `parse()` is called). By the way, the .ihtml extension of the template filename is customary for PHPLIB templates, but you can use .html, .tpl, or any other extension. Then we call `set_var()` to set the template variable `some_color` to the value of `$my_color` (which is "blue"), which means that all occurrences of `{some_color}` in the template will be replaced with the word blue when we call `parse()`.
And then we call `parse()`, which parses MyFileHandle by loading MyFileHandle (MyTemplate.ihtml) and replacing any template variables `("{some_variable}")` with their template variable values, and the resulting parsed text is placed in MyOutput. Nothing is output to the webserver until `p("MyOutput")` is called, which outputs the final parsed text.

Nested Templates

A neat feature of the `parse()` function is that the MyOutput handle that it created is actually a template variable, just as some_color is a template variable. So if you have another template with a `{MyOutput}` tag, when you parse that second template, all of the `{MyOutput}` tags will be replaced with the parsed text from MyOutput. This lets you embed the text of one template file into another template. So, we could have another template called wholePage.ihtml that contains the text:

Sorry you didn't win. But if you had won, we would have told you: {MyOutput}

And after wholePage.ihtml is parsed, the final output would be:
Sorry you didn't win. But if you had won, we would have told you: Congratulations! You won a new blue Honda Prelude!
Here is the PHP code to parse both templates:

```
<?php

$t = new Template("/home/mydir/mytemplates/");

// These three lines are the same as the first example:
$t->set_file("MyFileHandle","MyTemplate.ihtml");
$t->set_var("some_color",$my_color);
$t->parse("MyOutput","MyFileHandle");
// (Note that we don't call p()
//here, so nothing gets output yet.)

// Now parse a second template:
$t->set_file("WholeHandle","wholePage.ihtml");
    // wholePage.ihtml has "{MyOutput}" in it
$t->parse("MyFinalOutput","WholeHandle");
    // All {MyOutput}'s get replaced
$t->p("MyFinalOutput");
    // output the value of MyFinalOutput

?>
```

The last two lines with calls to `parse()` and `p()` can actually be combined using the shorthand function `pparse()` by replacing the last two lines with :

```
<?php
pparse("MyFinalOutput","SecondHandle");
?>
```

Another feature of PHPLIB Template is that the functions `set_file()` and `set_var()` can also accept multiple sets of values at a time by passing an array of handle/value pairs. Here are examples:

```
<?php

$t->set_file(array(
    "pageOneHandle" => "pageone.ihtml",
    "pageTwoHandle" => "pagetwo.ihtml"));

$t->set_var(array(
    "last_name" => "Gates",
    "first_name" => "Bill",
    "net_worth" => $reallybignumber));

?>
```

Appending Template Text

There is also a third parameter that you can pass to `parse()` and `pparse()` if you want to append data to the template variable rather than overwrite it. Simply call `parse()` or `pparse()` with the third parameter as true, such as:

```
<?php
```

```
$t->parse("MyOutput","MyFileHandle", true);
?>
```
If MyOutput already contains data, MyFileHandle will be parsed and appended onto the existing data in MyOutput. This technique is useful if you have a template where you want the same text to be repeated multiple times, such as listing multiple rows of results from a database query. You could also display the variables in an array, such as in this example:

```
<?php

$t = new Template("/home/mydir/mytemplates/");

$t->set_file(array(
  "mainpage" => "mainpage.ihtml",
  "each_element" => "each_element.ihtml"));

reset($myArray);

while (list($elementName, $elementValue) = each($myArray)) {

  // Set 'value' and 'name' to each element's value and name:
  $t->set_var("name",$elementName);
  $t->set_var("value",$elementValue);

  // Append copies of each_element:
  $t->parse("array_elements","each_element",true);
}

$t->pparse("output","mainpage");

?>
```
This example uses two templates, mainpage.ihtml and each_element.ihtml. The mainpage.ihtml template could look something like this:
```
<HTML>
  Here is the array:
  <TABLE>
    {array_elements}
  </TABLE>
</HTML>
```
The `{array_elements}` tag above will be replaced with copies of each_element.ihtml, which is repeated for each element of the array `($myArray)`. The each_element.ihtml template might look like this:

```
<TR>
  <TD>{name}: {value}</TD>
</TR>
```
The result is a nicely formatted table of the elements of `$myArray`. But wouldn't it be nice if these two templates could be combined into one template file? In fact, they can be combined using template blocks. Template blocks let you extract a block of text from a template so you can repeat it multiple times, or do anything else you would like with it. But I'll save that feature for another article.