PHPLib Templates

By Benjamin D. Smith

Table of Contents

Introduction

Hopefully, you've read Mr David Orr's introduction to working with PHPLIB templates. If you haven't, I recommend that you do so.
(http://www.phpbuilder.com/columns/david20000512.php3)

The most common use of templates is to allow you to change the look and feel of a site quickly without having to delve immediately into alot of PHP variable assignments and print statements.

And it's true that templates allow you to decouple your PHP code and the presentation HTML that's used to display its results but to limit PHPLIB templates to just this mundane chore would be a terrible disservice.

Used properly, PHPLIB templates can grant you an amazing ability to abstract the manipulation of data (in the database as well as in PHP) from its final format, whether that format is HTML, XML, WML, or a formatted e−mail, and some of these ways will be explored here.

There is usually an argument here about whether FastTemplates is really fast, or faster than PHPLIB, and which is better, etc.

I find that PHPLIB templates are more flexible than FastTemplate, don't require that you break nested blocks out into multiple files, and I don't really worry about performance that much, though I've been told that PHPLIB is also supposed to be faster, despite the other's name 'FastTemplate'.

This can almost get religeous, like arguing about using BSD vs Linux, or MySQL vs PostgreSQL, and I really don't want to get into that. Try both, and use the one you prefer:
PHPLIB: http://phplib.netuse.de
FastTemplates http://www.thewebmasters.net/

Hopefully, in either event, this article will be useful to you =)

Simplify

Templates need not be complex. For an example of a simple template, click here. This is a file that can be turned into a template with the greatest of ease − code like the following would easily support this:

```
<?
$time='time';
$people='men';
$nation='country';
include template01.txt;
?>
```

This is a template − it allows you to keep your code in one file and the output in another, and allows you to change the template without having to alter (much of) your PHP coding.

This is a most simple example, and only allows you to decouple really basic parts − which is why I bring up the powerful, flexible, and fast PHPLIB template class.

It's more complicated than the above example for simple variables − but when you get into the more advanced features, you'll quickly find that the additional complexity is a very small price compared to the benefits gained.

Home is where the heart is

At its heart, PHPLIB's template class works by defining variables that you stuff information into.

Although variables can be defined in any number of ways, it is important to remember that, once defined, all variables are dealt with in exactly the same way.

As with any programming language, variables contain information of some kind, and most of working with this templates class is defining what the variables are, what you want in them, and how you are going to assemble these variables into some output.

Once again, don't forget that these variables, once defined, are just logical buckets into which we'll dump text. Variables consists of strings of text, which can contain still other variables inside of which we can put more text.

This means that variables frequently exist "inside" others, much like directories can be contained in other directories.

Let's get started

Let's say you have a file, we'll call it 'one.ihtml', (The name, as well as the extension is completely arbitrary) and the template class is already instantiated inside object variable $T:

one.ihtml:

```
My name is {username} and I want to tell you {message}.
```

First off, we need to define a variable for the entire file..

```
<?
$T->set_file('input_one', 'one.ihtml');
?>
```

And there's two variables in this file, 'username' and 'message'. We'll need to define these:

```
<?
$T->set_var('username', 'Henry');
$T->set_var('message', 'a message');
?>
```

Now that we've set the variables in memory, we want to take the file variable and create some output:

```
<?
$T->parse('Output', 'input_one');
?>
```

Now, we have a total of four variables:

1) 'input_one' contains the original file we loaded. It contains two other variables within it.

2) 'username' contains the text string 'Henry'.

3) 'message' contains the string 'a message'.

4) 'Output' contains the original file with the variables replaced with the values we assigned... 'My name is Henry and I want to tell you a message.'

Likely as not you want to print the 'Output' variable to the screen. To do this, use pparse instead of parse:

```
<?
$T->pparse('Output', 'input_one');
?>
```

or just print the feecback of the parse function:

```
<?
echo $Tparse('Output', 'input_one');
?>
```

Simple so far? Read on...

Dealing with blocks

Now we get to dealing with blocks.

Really, it's just more of the same...

We define some variables with some stuff in them, and then we shuffle the variables around to get what we want.

Let's take a file called 'two.ihtml'...

```
A list of what I want for Christmas:
<OL>
<!-- BEGIN AccessBlock -->
<LI> {item}
<!-- END AccessBlock -->
</OL>
```

Now, we have two types of buckets we want to manipulate text in: a variable, and a block.

First, we define the variable that contains the entire file...

```
<?
$T->set_file('input_two', 'two.ihtml');
?>
```

and then we want to break out the block...

```
<?
$T->set_block('input_two', 'AccessBlock', 'ABlock');
?>
```

What did we just do here?

We now have three PHPLIB template variables defined here:

1) 'AccessBlock', which contains the text '<LI> {item}'

2) 'ABlock', which is a place−holder for where AccessBlock USED to be, and is now a variable, dealt with like any other variable!

3) 'input_two', which contained the original text of file two.ihtml MINUS the block we've just removed, which has been replaced with the variable tag 'ABlock'.

When 'AccessBlock' was removed from 'input_two', the text of '<LI> {item}' was taken out of the 'input_two' variable and replaced with the 'ABlock' variable.

Now, let's assume that we have an array of stuff we want for Christmas:

```
$want_list=array(
0 => 'Baseball bat',
1 => 'Remote Control car',
2 => 'Wagon'
);
```

We can make that Christmas list very easily here...

```
<?
for ($i=0; $i<sizeof($want_list); $i++) {
$T−>set_var('item', $want_list[$i]);
$T−>Parse('ABlock', 'AccessBlock', true);
}
$T−>pparse('Output', 'input_two');
?>
```

Pretty slick, eh?

So long as you keep the template block definitions and variable names the same within the source (.ihtml) file, you can use table rows, ordered/unordered lists, BR tags, whatever suits your fancy.

Nested blocks...

It's not hard to nest blocks inside other blocks. Using this methodology, you can have a single template file that contains any number of logical template 'pages'. You just have to remember to start from the innermost loop first.

Take some HTML source:

three.ihtml

```
Contents of this book:

<UL>
<!-- BEGIN ChapterBlock -->
<LI> Chapter {chapter}
<OL>
<!-- BEGIN PageBlock -->
<LI> Page {page}
<!-- END PageBlock -->
</OL>
<!-- END ChapterBlock -->
</UL>
```

We start by defining the file

```
<?
$T->set_file('input_three', 'three.ihtml');
?>
```

And then we need to break out the blocks, starting with the innermost block.

```
<?
$T->set_block('input_three', 'PageBlock', 'PBlock');
$T->set_block('input_three', 'ChapterBlock', 'CBlock');
?>
```

It is VERY VERY IMPORTANT that you always start by setting the innermost block, and work out from there! Otherwise, you've removed the block in question, and the next block assignment won't work.

Let's say we have some arrays:

```
<?
$chapters=array(
0 => 'Beginning',
1 => 'End'
);

$pages[0]=array(
0 => 'Getting Started',
1 => 'Setting it up',
2 => 'Positioning'
);

$pages[1]=array(
0 => 'Wrapping it up',
1 => 'Clean up'
);
?>
```

And we might handle these arrays like this:

```
<?
$T->set_file('input_three', 'three.ihtml');
$T->set_block('PageBlock', 'PBlock');
$T->set_block('ChapterBlock', 'CBlock');
for ($i=0; $i<sizeof($chapters); $i++) {
$T->set_var('PBlock', '');
for ($j=0; $j<sizeof($pages[$i]); $j++) {
$T->set_var('page', $pages[$i][$j]);
$T->parse('PBlock', 'PageBlock', true);
}
$T->set_var('chapter', $chapters[$i]);
$T->parse('CBlock', 'ChapterBlock', true);
}
```

```
$T->pparse('Output', 'input_three');
?>
```

Notice that just before we parse the chapters (the '$j' loop) that we set the 'PBlock' variable to ''. That's because we use the template block more than once, and if we don't clean the slate, the values from the previous loop will remain.

To see this, just rem out this line and try the code again:

```
$T->set_var('PBlock', '');
```

Aren't we done YET?

There's one last detail that I'd like to reinforce that I mentioned earlier.

Within the PHPLIB all variables are equal, even if they aren't created the same way. This is true of variables defined with set_file, set_var, or set_block!

To demonstrate this, let's take a look at four.ihtml:

```
<!-- BEGIN BlockOne -->
"I want to know what you think!"
<!-- END BlockOne -->

... so the wife says to her husband, {statement}.
```

and run this with the following code:

```
<?
$T->set_file('input_four', 'four.ihtml');
$T->set_block('BlockOne', 'BOne');
$T->parse('statement', 'BlockOne');
$T->set_var('BOne', '');
$T->pparse('Output', 'input_four');
?>
```

You'll note that the block of text gets put in the right place! Also note this line:

```
$T->set_var('BOne', '');
```

which clears out the BOne variable. This may or may not be necessary depending on how you have PHPLIB configured to run by default. Take a look at the documentation at http://phplib.netuse.de/documentation/documentation−4.html#ss4.2 and look for the class variable "unknowns". There's a function, set_unknowns to deal with this variable.

Closing

Some ideas for using this class:

1) Use a template to to give a site a consistent look and feel. With most sites, you want to have a consistent presentation of links, logos, and other information regardless of what portion of the site they are using. Templating is a drop−dead shoe in.

2) Use templates to allow for rapid and painless rebranding. If you work independantly, you can write your ASP application once, and rebrand the entire site by editing a single template file.

3) Use templates to automatically generate e−mails. This allows you to change your message at any time without diving into code.

4) Use templates for user preferences. A user could change the look and feel of your entire site by choosing the style of his/her choice.

5) Build an XML version of dynamic sites without touching a single bit of PHP code − instead, define a template variable, and use the same block of code with a different template file to generate the HTML or XML version of the site.

And the list goes on...

I hope this has given you some insights on this powerful and useful class.

Once again, documentation and downloads for the PHPLIB library are available at http://phplib.netuse.de .