By Team Melonfire

This article copyright Melonfire 2000−2002. All rights reserved.

Table of Contents

The Advanced Course

In the first part of this article, I demonstrated the basics of the patTemplate system, explaining how it could be used to build template−based Web applications. I showed you how to organize your Web pages into modular templates, and use variable placeholders in conjunction with the patTemplate API to populate and display those templates.

In this concluding article, I will be demonstrating a few of patTemplate's lesser−known features, together with a few more examples, in order to give you a better understanding of the system's capabilities. Among these features: conditional sub−templates, global variables and variable inheritance, and template visibility. Keep reading.

Scoping It Down

You've already seen how to use the AddVar() method to replace template variables with actual values. However, template variables are "local" to a template, and so, the value assigned to a template variable cannot be accessed from other templates (actually, that's a little white lie which I'm going to recant on the next page, but bear with me for a moment).

In the event that you need a variable which is "global", and whose value can be accessed from more than one template, patTemplate offers the addGlobalVar() method call, which makes the value of a template variable available to all other templates.

Take a look at the following templates, all of which use the template variable {IMAGES}:

```
<!-- app1.tmpl -->
<patTemplate:tmpl name="main">
<html>
<head>
<basefont face="Arial">
</head>

<body>

<table border="1" cellspacing="0" cellpadding="0">

<patTemplate:link src="top" />

<patTemplate:link src="middle" />

<patTemplate:link src="bottom" />

</table>

</body>
</html>
</patTemplate:tmpl>

<patTemplate:tmpl name="top">
<tr>
<td colspan="3"><img src="{IMAGES}/top.gif"></td>
</tr>
</patTemplate:tmpl>

<patTemplate:tmpl name="middle">
<tr>
<td valign="top"><img src="{IMAGES}/left.gif"></td>
<td>
<form action="login">
```

```
<table border="0" cellspacing="4" cellpadding="0">
<tr>
<td><font size="−1">Username:</font></td>
<td><input type="text" size="6"></td>
</tr>
<tr>
<td><font size="−1">Password:</font></td>
<td><input type="password" size="6"></td>
</tr>
</table>
</form>
</td>
<td valign="top"><img src="{IMAGES}/right.gif"></td>
</tr>
</patTemplate:tmpl>

<patTemplate:tmpl name="bottom">
<tr>
<td colspan="3"><img src="{IMAGES}/bottom.gif"></td>
</tr>
</patTemplate:tmpl>
```

Now, rather than making three calls to AddVar() − one for each of the templates referencing the variable {IMAGES} − I can save myself time with a single call to AddGlobalVar(), which makes the value of {IMAGES} available in the global namespace. Here's the script:

```
<?php

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template−>setBasedir("templates");

// add templates to the template engine
$template−>readTemplatesFromFile("app1.tmpl");

// add global variable
$template−>AddGlobalVar("IMAGES", "/app1/images");

// parse and display the template
$template−>displayParsedTemplate("main");
?>
```

Speaking In Tongues

Related, though in a tangential manner, to the material discussed on the previous page, is the "varscope" attribute. This attribute allows you to import the values of template variables from other templates into the current template, thereby reducing the number of calls you have to make to addVar(). Here's a quick example:

```
<patTemplate:tmpl name="main" varscope="lang">
Would you like me to speak with you in {LANG}?
</patTemplate:tmpl>

<patTemplate:tmpl name="lang">
{LANG}
</patTemplate:tmpl>
```

And here's the script:

```
<?php

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("lang.tmpl");

// define a value for the LANG variable in the "lang" template
$template->addVar("lang", "LANG", "Japanese");

// parse and display the "main" template
// since this template inherits values from "lang"
// the value of LANG in this template will automatically be
set
$template->displayParsedTemplate("main");
?>
```

In this case, I have two templates, "main" and "lang". Both contain references to the template variable {LANG}. However, although the template variable {LANG} has been assigned a value in the "lang" template, no such assignment has been made for the "main" template. Despite this, the "main" template will inherit the correct value from the "lang" template, via the "varscope" attribute specified in its opening tag.

This is clearly demonstrated in the output of the script above:

```
Would you like me to speak with you in Japanese?
```

Looping The Loop

You can force a particular template to loop a specific number of times by adding the "loop" attribute to it. Here's an example:

```
<patTemplate:tmpl name="main" loop="12">
It is now 1 o'clock
</patTemplate:tmpl>
```

Here's the script that powers it:

```
<?php

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("loop.tmpl");

// parse and display the template
$template->displayParsedTemplate("main");
?>
```

And here's the output:

```
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
It is now 1 o'clock
```

Fairly simple, and useful when all you need to do is display a piece of text or markup a specified number of times (you can use the setAttribute() method, discussed a little later, to set the loop counter dynamically). And you can make it even more useful by incorporating the current value of the loop counter within your template − like I've done below, with the special {PAT_ROW_VAR} variable:

```
<patTemplate:tmpl name="main" loop="12">
It is now {PAT_ROW_VAR} o'clock
</patTemplate:tmpl>
```

In this case, the same PHP script will generate slightly different output:

```
It is now 1 o'clock
It is now 2 o'clock
It is now 3 o'clock
It is now 4 o'clock
It is now 5 o'clock
It is now 6 o'clock
It is now 7 o'clock
It is now 8 o'clock
It is now 9 o'clock
It is now 10 o'clock
It is now 11 o'clock
It is now 12 o'clock
```

Legal Eagles

You can source an external file into your template via the "src" and "parse" template attributes. Let's suppose I have a copyright notice, stored in the file "copyright.txt", which looks like this:

```
Everything here is copyright Melonfire, 2002. Be good. We have
lawyers.
```

I can source this file by attaching the "src" attribute to a template,

```
<patTemplate:tmpl name="copyright" src="copyright.txt" />
```

and tell the engine whether or not to parse the sourced file for template variables with the additional "parse" attribute.

```
<patTemplate:tmpl name="copyright" src="copyright.txt"
parse="off" />
```

Once that's done, I can call this template from within another template, like this:

```
<patTemplate:tmpl name="main">
<html>
<head>
<basefont face="Arial">
</head>
<body>
This is my Web site. It has lots of interesting stuff on it
that you
might want to use for your own nefarious purposes. But before
you do,
read the notice at the bottom of this page. <p> <p> <hr>
<patTemplate:link src="copyright" />
</body>
</html>
</patTemplate:tmpl>
```

And now, when I parse and display the "main" template, the external file "copyright.txt" will be read and incorporated in the final output by the template engine. Here's what it looks like:

This is my Web site. It has lots of interesting stuff on it that you might want to use for your own nefarious purposes. But before you do, read the notice at the bottom of this page.

Everything here is copyright Melonfire, 2002. Be good. We have lawyers.

Hide And Seek

You can alter the visibility of a particular template via its "visibility" attribute. Consider the following example:

```
<patTemplate:tmpl name="main">
I spy, with my little eye...
<br>
<patTemplate:link src="toaster" />
</patTemplate:tmpl>

<patTemplate:tmpl name="toaster" visibility="hidden">
...a template beginning with T
</patTemplate:tmpl>
```

Here's the script:

```
<?php
// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("toaster.tmpl");

// parse and display the template
$template->displayParsedTemplate("main");
?>
```

And here's the output:

```
I spy, with my little eye...
```

Since the "visibility" attribute of the second template is set to "hidden", it will never be displayed by the template engine. In order to display it, you'll need to turn visibility to "visible" (or just remove the "visibility" attribute altogether):

```
<?php
// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("toaster.tmpl");

// turn visibility on for "toaster" template
// comment out the next line to have the template vanish
$template->setAttribute("toaster", "visibility", "show");

// parse and display the template
$template->displayParsedTemplate("main");
?>
```

And now the output will include that missing section:

```
I spy, with my little eye...
...a template beginning with T
```

As you will see, though this might not seem like a big deal right now, it becomes extremely powerful when combined with the ability to programmatically alter the "visibility" attribute on the fly. That's discussed on the next page.

Setting Things Right

All the examples you've seen thus far have had their attributes set at design time. This is not very useful in the majority of the cases; most often, you'd prefer to alter attributes like the loop counter or the visibility at run time, on the basis of logic in your script.

Luckily, patTemplate knows this − and it allows you to accomplish this goal via the very cool setAttribute() method, which allows you to dynamically set template attributes on the fly. Consider the following example, which demonstrates:

```
<!-- index.tmpl -->
<patTemplate:tmpl name="index">
<html>
<head><basefont face="Arial"></head>
<body>
Hello, and welcome to my Web site!
<!-- blah blah -->
<patTemplate:link src="tip" />
</body>
</html>
</patTemplate:tmpl>

<patTemplate:tmpl name="tip" visibility="hidden">
<p><hr>
<font size="-1">New user tip: Use the Find box at the top
right corner
of your screen to quickly search this site.</font>
</patTemplate:tmpl>
```

As you can see, there are two templates in the file above; the second one is initially hidden from view. However, I can turn it on in certain cases − such as, for example, when a user visits the site for the first time. Here's the script that takes care of this for me.

```
<?php

// alter this to see how the script functions
// when the variable is unset
$newUser = true;

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();
```

```
// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("index.tmpl");

// turn tips on if new user
if ($newUser == true)
{
$template->setAttribute("tip", "visibility", "visible");
}

// parse and display template
$template->displayParsedTemplate("index");
?>
```

Depending on the value of a particular variable, I can turn the second template on or off, via a call to setAttribute().

You can use the setAttribute() method to manipulate other template attributes as well − try it with the "loop" or "varscope" attributes to see how it works.

Fortune Favours The Brave

Thus far, I've been working with what patTemplate refers to as "standard" templates. However, the patTemplate class also comes with some decidedly non−standard − and rather cool − alternatives...and one of the neatest ones has to be its support for conditional templates.

Conditional templates function much like a series of "if−else" conditional statements − they allow you to display different output depending on how a particular, user−defined condition is evaluated. A conditional template typically contains a number of sub−templates, each keyed against a particular variable; depending on the value of that variable, the appropriate sub−template is extracted and used.

In order to better understand this, consider the following simple example:

```
<patTemplate:tmpl name="fortune" type="condition"
conditionvar="DAY">
<html> <head> <basefont face="Arial"> </head>

<body>

And today's fortune is:
<br>

<patTemplate:sub condition="Mon">
Never make anything simple and efficient when a way can be
found to make
it complex and wonderful. </patTemplate:sub>

<patTemplate:sub condition="Tue">
Life is a game of bridge −− and you've just been finessed.
</patTemplate:sub>

<patTemplate:sub condition="Wed">
What sane person could live in this world and not be crazy?
</patTemplate:sub>

<patTemplate:sub condition="Thu">
Don't get mad, get interest.
</patTemplate:sub>

<patTemplate:sub condition="Fri">
Just go with the flow control, roll with the crunches, and,
when you get
a prompt, type like hell. </patTemplate:sub>

</body>
</html>
</patTemplate:tmpl>
```

A little analysis, and you'll see that this isn't as complicated as it looks. The outer "fortune" template has been defined as a conditional template by the addition of two attributes to the <patTemplate:tmpl> tag − the "type" attribute, which is set to the value "condition", and the "conditionvar" attribute, which is set to the name of the decision variable to be used during the evaluation process.

This conditional template is then broken up into individual sub−templates, enclosed within <patTemplate:sub>...</patTemplate:sub> tags, and each possessing a "condition" attribute. This condition attribute specifies the value of the decision variable that the template engine will use when deciding which sub−template to display.

Here's the other half of the puzzle − the PHP script that actually sets a value for the decision variable so that the template engine can select an appropriate sub−template.

```
<?php

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("fortune.tmpl");

$template->AddVar("fortune", "DAY", date("D", mktime()));

// parse and display the template
$template->displayParsedTemplate("fortune");
?>
```

In this case, the PHP script merely sets the value of the template variable {DAY} − you'll remember that this is the decision variable defined in the conditional template − to the current day of the week, and then displays the parsed template. Internally, the template engine will match the value of {DAY} to the options available in the various sub−templates, and pick the one that fits.

As you can see, this is identical to a "switch" statement, or a series of "if−else" conditional statements − and it can come in fairly handy at times.

Running On Empty

patTemplate also allows you to cover for the unexpected by specifying two additional sub−templates, one which is displayed when the decision variable cannot be matched, and one to be displayed when the decision variable is empty. Here's an example of how this might work:

```
<patTemplate:tmpl name="fortune" type="condition"
conditionvar="DAY">
<html> <head> <basefont face="Arial"> </head>

<body>

And today's fortune is:
<br>

<patTemplate:sub condition="Mon">
Never make anything simple and efficient when a way can be
found to make
it complex and wonderful. </patTemplate:sub>

<patTemplate:sub condition="Tue">
Life is a game of bridge −− and you've just been finessed.
</patTemplate:sub>

<patTemplate:sub condition="Wed">
What sane person could live in this world and not be crazy?
</patTemplate:sub>

<patTemplate:sub condition="Thu">
Don't get mad, get interest.
</patTemplate:sub>

<patTemplate:sub condition="Fri">
Just go with the flow control, roll with the crunches,
and, when you get a prompt, type like hell.
</patTemplate:sub>

<patTemplate:sub condition="default">
Sorry, closed on the weekend.
</patTemplate:sub>

<patTemplate:sub condition="empty">
Sorry, cannot determine day of week. Did the world just end?
</patTemplate:sub>

</body>
</html>
```

```
</patTemplate:tmpl>
```

In this case, the sub−template specified as "default" will appear on Saturdays and Sundays, while the sub−template specified as "empty" will appear if {DAY} is empty. Try it out and see for yourself.

You can force patTemplate to look in the global namespace for the value of the decision variable in case it's not available locally, by adding the optional "useglobals" attribute to your template definition.

Simple Simon

In case the standard template type doesn't meet your needs, and the conditional one is too complicated, patTemplate offers you the best of both worlds with its "simpleCondition" template type. This template type defines a list of variables that are required for the template to be displayed; if these variables don't exist, the template will never be displayed.

With this in mind, it's possible to simplify and rewrite the example on the previous page as an illustration of the concept:

```
<patTemplate:tmpl name="fortune" type="simpleCondition"
requiredvars="DAY"> <html> <head> <basefont face="Arial">
</head>

<body>

And today's fortune is:
<br>
What sane person could live in this world and not be crazy?

</body>
</html>
</patTemplate:tmpl>
```

And here's the script:

```
<?php

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("fortune.tmpl");

// comment the next line and the template will never be
displayed
$template->AddVar("fortune", "DAY", date("D", mktime()));

// parse and display the template
```

```
$template->displayParsedTemplate("fortune");
?>
```

Brain Dump

Finally, if you're having problems with the engine, you can use the dump() method to view detailed debugging information on the template engine. Take a look at the following example, and its output:

```
<?php

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// set name of template file
$template->readTemplatesFromFile("music.tmpl");

// assign values to template variables
$template->AddVar("footer",
"COPYRIGHT", "This material copyright Melonfire, " . date("Y",
mktime()));

// dump template information
$template->dump();
?>
```

Here's what the output looks like:

The dump() method displays information about the available templates in the engine, the values of local and global template variables, a list of template attributes, and a list of unused variables. It provides an easy way

to see how the template engine has processed your templates, and to identify and correct errors that may have occurred in your business logic.

A Well−Formed Example

Finally, here's a composite example, this one using a conditional template to iteratively build a complete HTML form. The unique thing about this form: the form fields are completely configurable via a user−defined array, with multiple types of forms possible using the same template.

First, here are the templates I'll be using:

```
<!−− form.tmpl −−>
<!−− main page −−>
<patTemplate:tmpl name="form">
<html>
<head><basefont face="Arial"></head>
<body>
<form action="processor.php" method="post">

<patTemplate:link src="fields" />

<input type="submit" value="Save">
</form>
</body>
</html>
</patTemplate:tmpl>

<!−− field list −−>
<!−− conditional template containing sub−templates for each
field type
−−> <patTemplate:tmpl name="fields" type="condition"
conditionvar="FIELD_TYPE">
<patTemplate:sub condition="text">
{LABEL}
<br>
<input type="text" name="{NAME}">
<p>
</patTemplate:sub>

<patTemplate:sub condition="textarea">
{LABEL}
<br>
<textarea name="{NAME}"></textarea>
<p>
</patTemplate:sub>

<patTemplate:sub condition="password">
{LABEL}
<br>
<input type="password" name="{NAME}">
```

```
<p>
</patTemplate:sub>
</patTemplate:tmpl>
```

I've got two templates here: one for the main page and the outer form elements, and one conditional template which uses the {FIELD_TYPE} decision variable to render the form controls. Currently, my template only knows how to handle text fields, password fields and text areas − feel free to add more constructs here.

Now, I'm going to define a PHP array which will contain information on the fields I'd like to display in my form, together with their labels and names. This array is completely user−configurable, and may be defined at run time, from a database, configuration file or XML data source. Here's what it looks like:

```
// field list
// in form field−name => array(field−type, field−label)
$fields = array(
'fname' => array('text', 'First name'),
'lname' => array('text', 'Last name'),
'address' => array('textarea', 'Address'),
'tel' => array('text', 'Telephone number'),
'email' => array('text', 'Email address')
);
```

Now, all I need is a script to process this array and use it to assign appropriate values to the templates above.

```
<?php

// field list
// in form field−name => array(field−type, field−label)
$fields = array(
'fname' => array('text', 'First name'),
'lname' => array('text', 'Last name'),
'address' => array('textarea', 'Address'),
'tel' => array('text', 'Telephone number'),
'email' => array('text', 'Email address')
);

// include the class
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template−>setBasedir("templates");
```

```
// add templates to the template engine
$template->readTemplatesFromFile("form.tmpl");

// get field names as array
$keys = array_keys($fields);

// iterate throough array
foreach ($keys as $k)
{
// set field type
// iteratively build list of form fields
$template->addVar("fields", "FIELD_TYPE", $fields[$k][0]);
$template->addVar("fields", "NAME", $k);
$template->addVar("fields", "LABEL", $fields[$k][1]);
$template->parseTemplate("fields", "a");
}

// parse and display the template
$template->displayParsedTemplate("form");
?>
```

In this script, I'm iterating through the array, extracting information on each field name and type, and using that information to parse and render the conditional "fields" template. The "a" parameter to parseTemplate() ensures that the contents of each run are appended to the previous run, thereby iteratively constructing a series of form fields. Finally, the complete set of fields is plugged into the main page and displayed via displayParseTemplate().

Here's what it looks like:

Want a different form? Simply alter the $fields array, and watch in awe as a new form is dynamically generated before your very eyes. You gotta admit, that's pretty cool!

Crash Bang Boom

Another common use of patTemplate involves using it to display error and success codes while processing a script. Consider the following templates, which set up error and success pages respectively:

```
<!-- common.tmpl -->
<patTemplate:tmpl name="error">
<html>
<head><basefont face="Arial"></head>
<body>
An error occurred. Please contact the <a
href="mailto:webmaster@domain.com">webmaster</a>.
</body>
</html>
</patTemplate:tmpl>

<patTemplate:tmpl name="success">
<html>
<head><basefont face="Arial"></head>
<body>
The operation was successfully executed.
</body>
</html>
</patTemplate:tmpl>
```

Here's how I might use them in a script:

```
<?php
function checkErrors()
{
global $template, $error;
if ($error)
{
$template->displayParsedTemplate("error");
die;
}
}

function raiseError()
{
global $error;
$error = true;
}

// include the class
```

```
include("include/patTemplate.php");

// initialize an object of the class
$template = new patTemplate();

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("common.tmpl");

// set error variable
$error = false;

// script starts here

// process section 1
// no errors

checkErrors();

// process section 2
// let's assume an error occurred
raiseError();

checkErrors();

// process section 3
// no errors
checkErrors();

// end of script processing

// if we get this far, it means no errors
// display success code
$template->displayParsedTemplate("success");

?>
```

In this case, since an error was raised in section two of the script, the checkErrors() function will kill the script and display the error template when it is next invoked.

If, on the other hand, no errors are raised during execution of the script, the final call to checkErrors() will have no effect, the line following it will be executed and a success template will be displayed.

This is a fairly primitive example, but it does serve to demonstrate how a template engine can assist in creating powerful, flexible error handlers for your Web applications. It works like a charm most of the time − not to mention being fairly easy to maintain.

Endgame

And that's about it for the moment. In this article, you learned about some of patTemplate's more advanced features, including the ability to assign and use global variables, to dynamically switch templates on and off, and to create conditional templates which mimic the "switch" family of conditional statements. You also put your new−found knowledge to the test with a couple of real−life examples, using the template engine to dynamically generate Web forms and to gracefully recover from errors in script execution.

That's about it for this tutorial. I hope you enjoyed it, that you learned something useful from it, and that it encouraged you to look at patTemplate as a viable, more efficient alternative to the traditional way of constructing a PHP Web application. Be good, and I'll see you soon!

Note: All examples in this article have been tested on Linux/i586 with Apache 1.3.20 and PHP 4.1.0. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!