By Team Melonfire

Table of Contents

Looking For Alternatives

One of the nice things about PHP − and one of the primary reasons for its popularity as a rapid application development (RAD) tool − is the fact that PHP code can be inserted into regular HTML markup to turn otherwise static HTML content into dynamic, intelligent Web pages. This feature makes it possible to quickly write PHP scripts that build Web pages on the fly from a database (or other external data source), and to create "smart" Web applications more efficiently than would otherwise be possible with traditional programming languages like Java or Perl.

However, this ease of use comes with a price: most PHP−based Web sites are a mush of intermingled HTML markup and PHP function calls, making them hard to decipher and maintain. This problem most commonly rears its ugly head when interface designers need to alter the user interface presented to Web site visitors − since the presentation information is entwined with PHP code, changes to it typically require handholding by a developer with sufficient expertise in the language. Which ultimately means more people, more time and more money...

There is, however, an alternative.

This alternative solution involves using "templates" to separate presentation and layout information from program code, and a template engine to combine the two to create the final product. This two−tiered approach affords both developers and designers a fair degree of independence when it comes to maintaining a Web site, and can substantially reduce the time and effort required in the post−release phases of a development project.

Despite these advantages, this template−based approach is not that popular − or even that well−known − amongst developers, especially those that are new to Web development. And so, over the course of this two−part article, I will be attempting to demystify how it works, in the hope that it will encourage you to use it in your next development effort.

Which is where patTemplate comes in...

Hard Sell

patTemplate is a PHP−based template engine designed, in the author's words, to "help you separate program logic or content from layout". Developed by Stephan Schmidt, it is freely available for download and is packaged as a single PHP class which can be easily included in your application.

patTemplate uses templates to simplify maintenance of PHP code, and to separate data from page elements. It assumes that a single Web application is made up of many smaller pieces − it calls these parts "templates" − and it provides an API to link templates together, and to fill them with data.

In patTemplate lingo, a "template" is simply a text file, typically containing both static elements (HTML code, ASCII text et al) and patTemplate variables. When patTemplate parses a template file, it automatically replaces the variables within it with their values (this is referred to as "variable interpolation"). These values may be defined by the developer at run−time, and may be either local to a particular template, or global across all the templates within an application.

As you will see, patTemplate also makes it possible to "nest" one template within another, adding a whole new level of flexibility to this template−based method of doing things. By allowing you to split up a user interface into multiple smaller parts, patTemplate adds reusability to your Web application (a template can be used again and again, even across different projects) and makes it easier to localize the impact of a change.

As if all that wasn't enough, patTemplate also makes it possible to hide or show individual sub−templates, to repeat templates as many times as needed, to iteratively build templates, and to use conditional tests within a template. Since it's not HTML−specific − your template can contain code in any format you like − it can even generate output in ASCII, CSV or XML.

Before proceeding further, you should visit the patTemplate home page at http://www.php−tools.de/ and download a copy of the latest version (2.4 at the time of writing). The package contains the main class file, documentation outlining the exposed methods and variables, and some example scripts.

Message In A Bottle

With the advertising out of the way, let's take a simple example to see how patTemplate works. Consider the following template:

```
<patTemplate:tmpl name="message">
<html>
<head>
<basefont face="Arial">
</head>

<body>

<h2>{TITLE}</h2>

{MESSAGE}

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

The block above represents a single template, identified by the opening and closing <patTemplate:tmpl> tags, and by a unique name ("message", in this case).

Within the opening and closing tags comes the actual template body; to all intents and purposes, this is a regular HTML document, except that the markup also contains some special patTemplate variables, written in uppercase and enclosed within curly braces. These special variables will be replaced with actual values once the template engine gets its mitts on it. Let's look at that next.

Anatomy Of A Template Engine

Next, it's time to initialize the template engine and have it populate the template created on the previous page with actual data. Here's how:

```
<?php

// include the class
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("message.tmpl");

// set values for template variables
$template->addVar("message", "TITLE", "Error");
$template->addVar("message", "MESSAGE", "Total system meltdown
in
progress");

// parse and display the template
$template->DisplayParsedTemplate("message");
?>
```

This might seem a little complicated, so let me dissect it for you:

1. The first step is, obviously, to include all the relevant files for the template engine. Since patTemplate is packaged as a single class, all I need to do is include that class file.

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

Once that's done, I can safely create an object of the patTemplate class.

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

This object instance will serve as the primary control point for the template engine, allowing me to do all kinds of nifty things.

2. Next, the object's setBaseDir() and readTemplatesFromFile() methods are used to point the engine in the direction of the templates to be read. The setBaseDir() method sets the default location for all template files; the readTemplatesFromFile() method specifies which template file to process.

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

// set name of template file
$template−>readTemplatesFromFile("message.tmpl");
```

patTemplate allows multiple templates to be stored in the same physical file; the engine uses each template's name to uniquely identify it.

3. Once all the templates in the specified file have been read, it's time to do a little variable interpolation. This is accomplished via the addVar() object method, which attaches values to template variables.

```
// set values for template variables
$template−>addVar("message", "TITLE", "Error");
$template−>addVar("message", "MESSAGE", "Total system meltdown
in
progress");
```

In English, this translates to "find the template named 'message' and assign the value 'Error' to the placeholder {TITLE} within it ".

You can run addVar() as many times as you like to perform variable interpolation.

4. Once you're done with all the variable replacement, all that's left is to display the final product.

```
// parse and display the template
$template−>DisplayParsedTemplate("message");
```

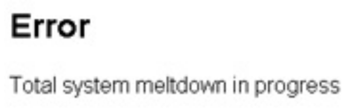
The DisplayParsedTemplate() object method parses the specified template, replacing all variables within it with their specified values, and outputs it to the output device.

In this specific example, the call to DisplayParsedTemplate() could also be broken down into two separate components, which combine to produce an equivalent result.

```
// parse and display the template
$template−>parseTemplate("message");
echo $template−>getParsedTemplate("message");
```

In this version, the call to parseTemplate() merely parses the named template and replaces variables within it with appropriate values, while the call to getParsedTemplate() gets the contents of the parsed template and places it in a string variable.

As you will see, this alternative version has its uses, especially when it comes to iteratively building or repeating a Web page. This is discussed in detail a little further along − for the moment, just feast your eyes on the result of all the work above:



Error

Total system meltdown in progress

The nice thing about this approach? The page interface and page elements are separated from the program code that actually makes the page function − and can therefore be updated independently by Web designers and PHP developers.

Slice And Dice

It's also possible to split a single page into smaller, modular templates, and link these templates together to create new and interesting shapes and patters. Take a look at how this might be accomplished:

```
<!-- main page -->
<patTemplate:tmpl name="main">

<patTemplate:link src="header" />

<patTemplate:link src="body" />

<patTemplate:link src="footer" />

</patTemplate:tmpl>

<!-- page header -->
<patTemplate:tmpl name="header">
<html>
<head>
<basefont face="Arial">
</head>
<body bgcolor="navy" text="white" link="white" vlink="white"
alink="white">
</patTemplate:tmpl>

<!-- page body -->
<patTemplate:tmpl name="body">
<center>
Feelin' blue? How about a little <a
href="http://www.melonfire.com/community/columns/boombox/">music</a>?
</center>
</patTemplate:tmpl>

<!-- page footer -->
<patTemplate:tmpl name="footer">
<p>&nbsp;<p align="right">
<font size="-2">{COPYRIGHT}</font>
</body>
</html>
</patTemplate:tmpl>
```

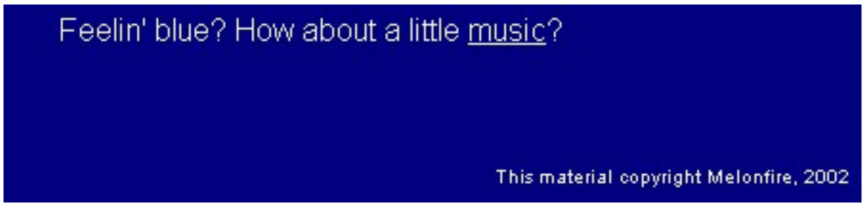
In this case, I have three different templates, one each for the page header, body and footer. I also have a fourth template, this one merely containing links to the remaining three templates. When patTemplate parses this container template, it will find and follow the links to the other templates, parse them and display them.

Here's the script which does all the work:

```
<?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()));

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

As you can see, it's almost identical to the one on the previous page − which only serves to demonstrate how transparent and simple the process of linking templates together is.

Here's what the output looks like:



Feelin' blue? How about a little music?

This material copyright Melonfire, 2002

If you have a Web page which consists of many individual pieces, you'll find this ability to create independent, modular templates immensely valuable − and once you start using it, you'll wonder how you ever worked without it!

Music To Your Ears

While on the subject, it's important to note that it isn't even necessary for all your templates to be stored in one physical file − patTemplate allows you the flexibility to store your templates in as many physical containers as you like, and to organize them in the manner that best suits your project. The template engine comes with a fairly well−developed API to read and combine the templates in the different files together − as the following example demonstrates:

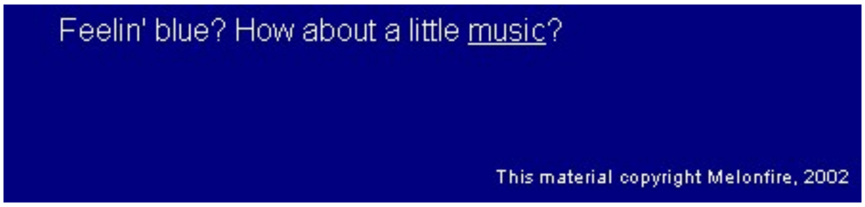
```
<!-- common page elements -->
<!-- stored in file common.tmpl -->

<!-- page header -->
<patTemplate:tmpl name="header">
<html>
<head>
<basefont face="Arial">
</head>
<body bgcolor="navy" text="white" link="white" vlink="white"
alink="white">
</patTemplate:tmpl>

<!-- page footer -->
<patTemplate:tmpl name="footer">
<p>&nbsp;<p align="right">
<font size="-2">{COPYRIGHT}</font>
</body>
</html>
</patTemplate:tmpl>

<!-- page body -->
<!-- stored in file music.tmpl -->
<patTemplate:tmpl name="body">
<center>
Feelin' blue? How about a little <a
href="http://www.melonfire.com/community/columns/boombox/">music</a>?
</center>
</patTemplate:tmpl>

<!-- main page -->
<!-- stored in file main.tmpl -->
<patTemplate:tmpl name="main">

<patTemplate:link src="header" />

<patTemplate:link src="body" />

<patTemplate:link src="footer" />
```

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

Here's the script which puts them all together:

```
<?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("music.tmpl");
$template->readTemplatesFromFile("common.tmpl");
$template->readTemplatesFromFile("main.tmpl");

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

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

And here's the output:



Feelin' blue? How about a little music?

This material copyright Melonfire, 2002

Simple, huh?

Watching The Clock

Another interesting application of patTemplate involves using a single template to iteratively generate a sequence of markup elements; this comes in particularly handy when creating HTML constructs like lists and table rows.

Here's a simple example, a template containing a single item:

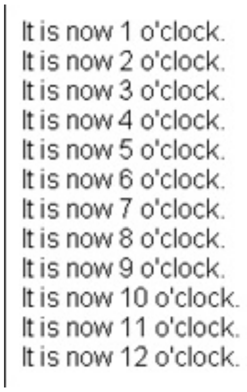
```
<!-- main page -->
<patTemplate:tmpl name="body">
<html>
<head>
<basefont face="Arial">
</head>

<body>

<patTemplate:link src="sequence" />

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

<!-- item to be repeated -->
<patTemplate:tmpl name="sequence">
It is now {TIME} o'clock.
<br>
</patTemplate:tmpl>
```

Looks harmless, doesn't it? But patTemplate lets you turn that insipid−looking template into a full−fledged sequence, parsing it multiple times and adding the output generated at each pass to that generated in previous passes. Take a look:

```
<?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("sequence.tmpl");

for ($x=1; $x<=12; $x++)
{
// assign values to template variables
$template−>AddVar("sequence", "TIME", $x);
$template−>parseTemplate("sequence", "a");
}

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

Here's the 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.
```

In this case, every time parseTemplate() is invoked, the template variable {TIME} is replaced with a new value, and the resulting output is appended to the output generated in previous calls to parseTemplate(). This is made possible via the additional "a" − which stands for "append" − parameter in the call to parseTemplate().

As you might imagine, this can come in particularly handy when you're building a Web page dynamically from a database, and need to repeat a similar sequence of markup elements a specific number of times. Here's another, more useful example:

```
<!−− addressbook.tmpl −−>
<!−− main page −−>
<patTemplate:tmpl name="body">
<html>
<head>
<basefont face="Arial">
</head>

<body>

<h2>Address Book</h2>
```

```
<table border="1" cellspacing="0" cellpadding="5">
<tr>
<td align="center"><i>Name</i></td>
<td align="center"><i>Address</i></td>
<td align="center"><i>Tel</i></td>
<td align="center"><i>Fax</i></td>
</tr>

<patTemplate:link src="row" />

</table>

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

<!-- item to be repeated -->
<patTemplate:tmpl name="row">
<tr>
<td>{NAME}</td>
<td>{ADDRESS}</td>
<td>{TEL}</td>
<td>{FAX}</td>
</tr>
</patTemplate:tmpl>
```

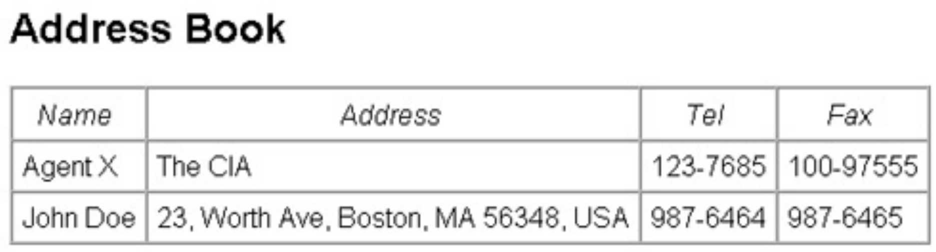
Here's the PHP script that brings it together:

```
<?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("addressbook.tmpl");

// open database connection
$connection = mysql_connect("localhost", "someuser",
"somepass") or die
("Unable to connect!");
```

```
// select database
mysql_select_db("data") or die ("Unable to select database!");

// generate and execute query
$query = "SELECT * FROM addressbook ORDER BY name";
$result = mysql_query($query) or die ("Error in query: $query.
" .
mysql_error());

// if records present
if (mysql_num_rows($result) > 0)
{
// iterate through resultset
// assign values to template variables from resultset fields
// iteratively build sequence of <tr>s
while($row = mysql_fetch_object($result))
{
$template−>AddVar("row", "NAME", $row−>name);
$template−>AddVar("row", "ADDRESS", $row−>address);
$template−>AddVar("row", "TEL", $row−>tel);
$template−>AddVar("row", "FAX", $row−>fax);
$template−>parseTemplate("row", "a");
}
}

// close connection
mysql_close($connection);

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

This is similar to the previous example, except that, this time, I'm using a result set from a MySQL database query as the data source for the template. As this result set is processed, a table is iteratively constructed and rendered using basic units like table cells and rows.

Here's what the end result looks like:



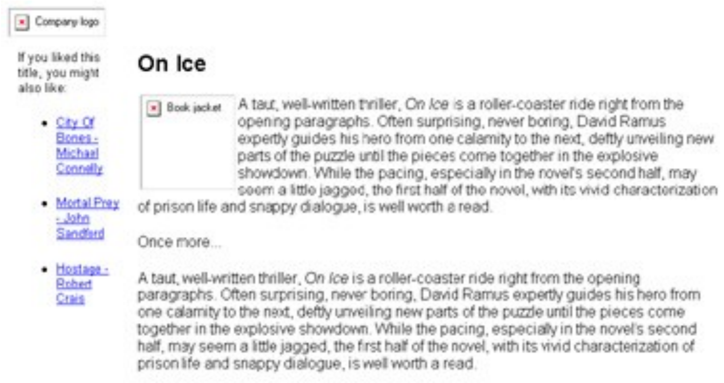
Address Book

Name	Address	Tel	Fax
Agent X	The CIA	123-7685	100-97555
John Doe	23, Worth Ave, Boston, MA 56348, USA	987-6464	987-6465

A Bookworm In The Ointment

I'd like to wrap up this introductory article with a comprehensive example, which builds on what you've learned so far to demonstrate how easy it is to use patTemplate to quickly construct different types of page layouts.

Let's suppose I wanted to generate a Web page for an item in an online book store, and let's further suppose that I wanted it to look like this:



If you liked this title, you might also like:

- City Of Bones - Michael Connelly
- Mortal Prey - John Sandford
- Hostage - Robert Crais

On Ice

A taut, well-written thriller, *On Ice* is a roller-coaster ride right from the opening paragraphs. Often surprising, never boring, David Ramus expertly guides his hero from one calamity to the next, deftly unveiling new parts of the puzzle until the pieces come together in the explosive showdown. While the pacing, especially in the novel's second half, may seem a little jagged, the first half of the novel, with its vivid characterization of prison life and snappy dialogue, is well worth a read.

Once more...

A taut, well-written thriller, *On Ice* is a roller-coaster ride right from the opening paragraphs. Often surprising, never boring, David Ramus expertly guides his hero from one calamity to the next, deftly unveiling new parts of the puzzle until the pieces come together in the explosive showdown. While the pacing, especially in the novel's second half, may seem a little jagged, the first half of the novel, with its vivid characterization of prison life and snappy dialogue, is well worth a read.

Here are the templates I plan to use:

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

<body>

<!-- header -->
<img src="logo.gif" alt="Company logo">
<!-- content -->
<table width="100%" cellspacing="5" cellpadding="5">
<tr>
<td valign="top">
<patTemplate:link src="recommendations" />
</td>
<td valign="top">
<patTemplate:link src="review" />
</td>
</tr>
</table>
<!-- footer -->
```

```
<hr>
<center><font size=-2>All content copyright and proprietary <a
href="http://www.melonfire.com/">Melonfire</a>, 2002. All
rights
reserved.</font></center>

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

<!-- review section -->
<patTemplate:tmpl name="review">
<h2>{TITLE}</h2>
<p>
<img src="{POSTER}" width="100" height="100" alt="Book jacket"
align="left">{CONTENT}
</patTemplate:tmpl>

<!-- reco section -->
<patTemplate:tmpl name="recommendations">
<font size="-1">If you liked this title, you might also like:
<br>
<ul>
<patTemplate:link src="recommendations_list" />
</ul>
</font>
</patTemplate:tmpl>

<!-- reco item section -->
<patTemplate:tmpl name="recommendations_list">
<li><a href="story.php?id={ID}"><font
size=-1>{ITEM}</font></a>
<p>
</patTemplate:tmpl>
```

I'm going to use these four templates to generate the layout illustrated above.

```
<?php

/*** this entire section would come from a database ***/

// book details - title, content, poster
$title = "On Ice";
$content = "A taut, well-written thriller, <i>On Ice</i> is a
roller-coaster ride right from the opening paragraphs. Often
surprising,
never boring, David Ramus expertly guides his hero from one
```

```
calamity to
the
next, deftly unveiling new parts of the puzzle until the
pieces come
together in the explosive showdown. While the pacing,
especially in the
novel's second half, may seem a little jagged, the first half
of the
novel,
with its vivid characterization of prison life and snappy
dialogue, is
well
worth a read.<p>Once more...<p>A taut, well−written thriller,
<i>On
Ice</i>
is a roller−coaster ride right from the opening paragraphs.
Often
surprising, never boring, David Ramus expertly guides his hero
from one
calamity to the next, deftly unveiling new parts of the puzzle
until the
pieces come together in the explosive showdown. While the
pacing,
especially in the novel's second half, may seem a little
jagged, the
first
half of the novel, with its vivid characterization of prison
life and
snappy dialogue, is well worth a read.<p>";
$image = "poster.gif";

// list of titles for recommendations
$items = array();
$items[0] = "City Of Bones − Michael Connelly";
$items[1] = "Mortal Prey − John Sandford";
$items[2] = "Hostage − Robert Crais";

// corresponding review ids
$ids = array();
$ids[0] = 23;
$ids[1] = 124;
$ids[2] = 65;

/*** database action ends **/

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

// initialize an object of the class
```

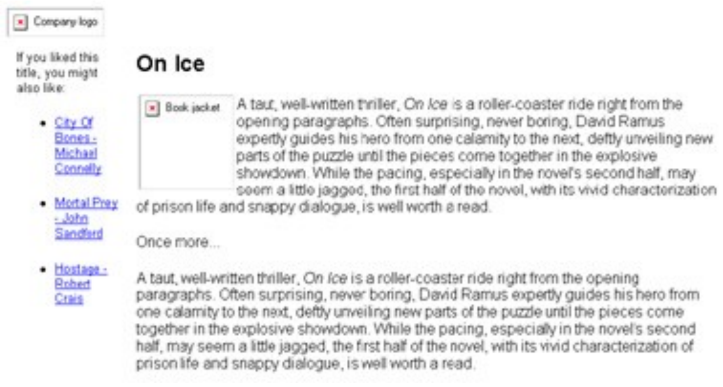
```
$template = new patTemplate();

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

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

$template−>AddVar("review", "TITLE", $title);
$template−>AddVar("review", "POSTER", $image);
$template−>AddVar("review", "CONTENT", $content);

// iterate through array
// assign values to template variables from array fields
// iteratively build list
for ($x=0; $x<sizeof($items); $x++)
{
$template−>AddVar("recommendations_list", "ID", $ids[$x]);
$template−>AddVar("recommendations_list", "ITEM", $items[$x]);
$template−>parseTemplate("recommendations_list", "a");
}

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

The first part of this script is focused solely on extracting information to display from a database − I've hard−coded the values here for demonstration purposes. Once the variables are set, the script initializes a patTemplate object and reads the four templates I plan to use into the template engine.

Next, values are assigned to the {TITLE}, {POSTER} and {CONTENT} variables within the "review" template. Once that's done, the list of recommendations is iteratively built, using the data in the $items and $ids arrays.

At the end of this process, the "recommendations_list" template stores a complete list of recommended books. This is then placed in the "recommendations" template, and, finally, both "recommendations" and "review" are transposed in the "main" template and printed to the browser.

Here's what the output looks like:


Company logo

If you liked this title, you might also like:

- City Of Bones - Michael Connelly
- Mortal Prey - John Sandford
- Hostage - Robert Crais

On Ice

Book jacket

A taut, well-written thriller, *On Ice* is a roller-coaster ride right from the opening paragraphs. Often surprising, never boring, David Ramus expertly guides his hero from one calamity to the next, deftly unveiling new parts of the puzzle until the pieces come together in the explosive showdown. While the pacing, especially in the novel's second half, may seem a little jagged, the first half of the novel, with its vivid characterization of prison life and snappy dialogue, is well worth a read.

Once more...

A taut, well-written thriller, *On Ice* is a roller-coaster ride right from the opening paragraphs. Often surprising, never boring, David Ramus expertly guides his hero from one calamity to the next, deftly unveiling new parts of the puzzle until the pieces come together in the explosive showdown. While the pacing, especially in the novel's second half, may seem a little jagged, the first half of the novel, with its vivid characterization of prison life and snappy dialogue, is well worth a read.

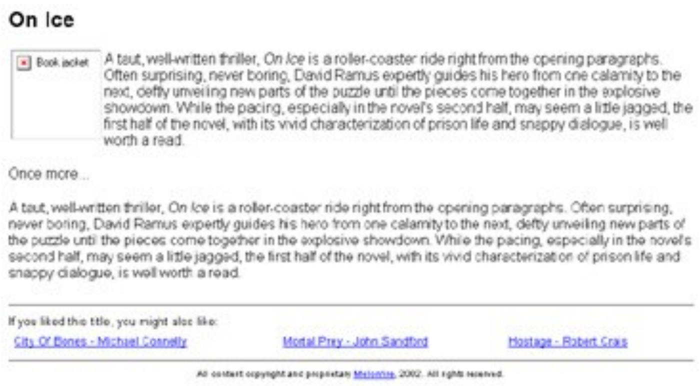
A Rose By Any Other Name...

So that's one look − but now how about changing it a little? Let's do away with the link boxes altogether, and have the links appear in neat rows at the bottom...

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

<body>

<!-- header -->
<img src="logo.gif" alt="Company logo">
<!-- content -->
<patTemplate:link src="review" />
<p><hr>
<patTemplate:link src="recommendations" />
<!-- footer -->
<hr>
<center><font size=-2>All content copyright and proprietary <a
href="http://www.melonfire.com/">Melonfire</a>, 2002. All
rights
reserved.</font></center>

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

<!-- review section -->
<patTemplate:tmpl name="review">
<h2>{TITLE}</h2>
<p>
<img src="{POSTER}" width="100" height="100" alt="Book jacket"
align="left">{CONTENT}
</patTemplate:tmpl>

<!-- reco section -->
<patTemplate:tmpl name="recommendations">
<font size="-1">If you liked this title, you might also
like:</font>
<br>
<table width="100%" cellspacing="3" cellpadding="3">
<tr>
```

```
<patTemplate:link src="recommendations_list" />
</tr>
</table>
</patTemplate:tmpl>

<!-- reco item section -->
<patTemplate:tmpl name="recommendations_list">
<td><a href="story.php?id={ID}"><font
size=-1>{ITEM}</font></a></td>
</patTemplate:tmpl>
```

In this case, I've altered three of the templates to remove the tables and list constructs, so that I'm left with a very simple and elegant layout. Since all I'm doing is altering the layout, no changes are required to the PHP script itself; it should function as before.except that, this time, the output will look like this:



On Ice

Book jacket

A taut, well-written thriller, *On Ice* is a roller-coaster ride right from the opening paragraphs. Often surprising, never boring, David Ramus expertly guides his hero from one calamity to the next, deftly unveiling new parts of the puzzle until the pieces come together in the explosive showdown. While the pacing, especially in the novel's second half, may seem a little jagged, the first half of the novel, with its vivid characterization of prison life and snappy dialogue, is well worth a read.

Once more...

A taut, well-written thriller, *On Ice* is a roller-coaster ride right from the opening paragraphs. Often surprising, never boring, David Ramus expertly guides his hero from one calamity to the next, deftly unveiling new parts of the puzzle until the pieces come together in the explosive showdown. While the pacing, especially in the novel's second half, may seem a little jagged, the first half of the novel, with its vivid characterization of prison life and snappy dialogue, is well worth a read.

If you liked this title, you might also like:

City Of Bones - Michael Connelly   Mortal Prey - John Sandford   Hostage - Robert Crais

All content copyright and proprietary Melonfire, 2002. All rights reserved.

As you can see, patTemplate makes it possible to separate the user interface from the program logic, thereby allowing designers with little or no programming knowledge to alter Web pages quickly and easily. Further, with its ability to nest and repeat chunks of HTML code, it can speed up development time significantly, and also reduce the effort involved in maintaining and modifying a Web application.

And that's about it for the moment. In this article, you learned why using templates to build a Web application can save you time and effort during the post−release and maintenance phases of a Web project. You saw how the patTemplate system allows you to organize your user interface into separate, modular templates, and dynamically populate them via variable placeholders. You also saw how the patTemplate system allows you to link templates together, and to construct a complete Web page iteratively from atomic units. Finally, you put all those techniques to the test to create two different page layouts merely by altering the appropriate templates, with no changes required to the business logic of the application.

In the next (and concluding part) of this tutorial, I will be introducing you to some of patTemplate's more advanced features, including the ability to switch individual templates on and off, to create global variables, to automatically have your templates inherit variables from each other, and to create conditional templates and sub−templates. Make sure you come back for that!

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!