Apress™
Books for Professionals by Professionals

Chapter Nine: "PHP and Dynamic Site Development"

A Programmer's Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

info@apress.com

CHAPTER 9

PHP and Dynamic Site Development

If you're anything like myself, chances are you typically flip through the chapters of any computer textbook that do not apply immediately to what you're interested in learning, instead skipping directly to those chapters that apply to what you *do* want to know. Chances are, that's just fine; computer textbooks usually aren't meant to be read from cover to cover anyway. You are probably smiling as you read this paragraph, because you indeed did just this, skipping past the first eight chapters, settling on this one because it has the most interesting title. After all, no time to read the details when the big bossman is screaming in your ear, right?

Luckily, your eagerness won't inhibit you too much in regard to your comprehension of much of the material covered in Part II of this book, which turns our attention directly toward how PHP is used to build and interact with the Web. This chapter introduces how PHP's dynamic properties can easily change the content and navigability of Web pages through the use of links and various predefined functions. The next chapter will build on what you've learned here, delving into how PHP perpetuates user interaction via HTML forms. Chapter 11 then turns to PHP's database-interfacing capabilities. And onward we march toward the conclusion of Part II, covering several of the more advanced subjects of PHP's Web development capability.

However, keep in mind that the material covered in Part I is *indispensable* to the language. I will make use of many of the concepts covered in my code examples and will assume that you have read Part I. So if you have skipped ahead, be prepared to occasionally refer to previous chapters to bring yourself up to speed regarding a few topics.

Simple Linking

Just as a link can be used to direct a user to an HTML page, it can be used to lead to a PHP-enabled page, as shown here:

```
<a href = "date.php">View today's date</a>
```

Clicking the link will transport you to the page entitled "date.php". Simple enough, right? Building on this example, you can use a variable to construct a dynamic link:

```
<?
$link = "date.php";
print "<a href = \"$link\">View today's date</a> <br>\n";
?>
```

You're probably wondering why the double quotation marks ("") are preceded by backslashes (\). This is because double quotation marks are special characters in PHP, since they are also used to enclose print statements. Thus, they must be escaped with the backslash.

> **TIP** *If you find escaping double quotation marks annoying, just enable magic_quotes_gpc in the php.ini file. Once this is enabled, all single quotation marks, double quotation marks, backslashes, and NULL characters will be automatically escaped!*

Building on this example, you can use an array to quickly display a list of links to the browser:

```
<?
// create array of content titles
$contents = array("tutorials", "articles", "scripts", "contact");
// loop through and display each element of the array.
for ($i = 0; $i < sizeof($contents); $i++)
     print " &#149; <a href = \"".$contents[$i].".php\">".$contents[$i]."</a>
<br>\n";
     // The &149; is the special character for a bullet.
endfor;
?>
```

The result:

```
&#149; <a href = "tutorials.php">tutorials</a> <br>
&#149; <a href = "articles.php">articles</a> <br>
&#149; <a href = "scripts.php">scripts</a> <br>
&#149; <a href = "contact.php">contact</a> <br>
```

File Components (Basic Templates)

At this point, I turn to one of my favorite capabilities of PHP: templates. A template as it pertains to Web building is essentially a particular part of the Web document that you would like to use in more than one page. A template, just like a PHP function, saves you from needing to redundantly type or cut and paste sections of page content and code. This concept will become particularly important as your site grows in size, as use of templates will allow you to quickly and easily perform sitewide modifications quickly. In the next few pages, I will introduce the advantages of building basic templates.

Typically, these code/content sections (or templates) are placed in one or more separate files. As you build your Web document, you simply "include" these pages into their respective position on the page. Two PHP functions provide for doing just this: `include()` and `require()`.

include() and require()

One of the powerful aspects of PHP is the ability to build templates and code libraries that can be easily inserted into new scripts. In the long run, using code libraries can drastically minimize time and tears when you need to reuse common functionality across Web sites. Those of you with backgrounds in other languages (such as C, C++, or Java) are familiar with this concept of code libraries (hereafter code libraries and templates will be singly referred to as templates) and including them in your program to extend functionality.

You can build basic templates easily by including one or a series of files via two of PHP's predefined functions, `require()` and `include()`. Each has its specific application, as you will learn in the next section.

The Functions

There are four functions that can be used to include files in a PHP script:

- `include()`
- `include_once()`
- `require()`
- `require_once()`

Although some of these seem equivalent in function due to the similarity of their names, be forewarned that each has a distinct purpose.

include()

The include() function does exactly what its name implies; it includes a file. This is its syntax:

```
include (file insertion_file)
```

An interesting characteristic of include() is that you can execute it conditionally. For example, if an include is placed in an *if* statement, *insertion_file* will be included only if the *if* statement in which it is enclosed evaluates to true. Keep in mind that if you decide to use include() in a conditional, the include() construct *must* be enclosed in statement block curly brackets or in the alternative statement enclosure. Consider the difference in syntax between Listings 9-1 and 9-2.

Listing 9-1: Incorrect usage of include()

```
…
if (some_conditional)
     include ('some_file');
else
     include ('some_other_file');
…
```

Listing 9-2: Correct usage of include()

```
. . .
if (some_conditional) :
     include ('some_file');
else :
     include ('some_other_file');
endif;
. . .
```

One misleading aspect of the include() statement is that any PHP code in the included file *must* be escaped with valid PHP enclosure tags. Therefore, you could not just place a PHP command in a file and expect it to parse correctly, such as the one found here:

```
print "this is an invalid include file";
```

Instead, any PHP statements must be enclosed with the correct escape tags, as shown here:

```
<?
print "this is an invalid include file";
?>
```

include_once()

The `include_once()` function has the same purpose as `include()`, except that it first verifies whether or not the file has already been included. If it has been, `include_once()` will not execute. Otherwise, it will include the file as necessary. Other than this difference, `include_once()` operates in exactly the same way as `include()`. Its syntax follows:

```
include_once (file insertion_file)
```

require()

For the most part, `require()` operates like `include()`, including a template into the file in which the `require()` call is located. It has this syntax:

```
require(file insertion_file)
```

However, there is one important difference between `require()` and `include()`. The *insertion_file* will be included in the script in which the `require()` construct appears regardless of where `require()` is located. For instance, if `require()` were placed in an if statement that evaluated to false, *insertion_file* would be included anyway!

> **TIP** *A URL can be used with* `require()` *only if "URL fopen wrappers" has been enabled, which by default it is.*

It is often useful to create a file containing variables and other information that may be used throughout the site and then require it where necessary. Although you can name this file anything you like, I like to call mine "init.tpl" (short for "*init*ialization.*t*em*pl*ate"). Listing 9-3 shows what a very simple init.tpl file would look like. Listing 9-4 subsequently uses `require()` to include the init.tpl information into its script.

Listing 9-3: A sample file to be inserted (init.tpl)

```
<?
$site_title = "PHP Recipes";
$contact_email = "wjgilmore@hotmail.com";
$contact_name = "WJ Gilmore";
?>
```

Listing 9-4 inserts the init.tpl information into its script and then uses the variables in it to dynamically change the page contents.

Listing 9-4: Making use of init.tpl

```
<? require ('init.tpl'); ?>
<html>
<head>
<title><?=$site_title;?></title>
</head>
<body>
<?
print "Welcome to $site_title. For questions, contact <a href =
\"mailto:$contact_email\">$contact_name</a>."; ?>
</body>
</html>
```

As your site grows in size, you may find yourself redundantly including particular files. While this might not always be a problem, sometimes you will not want modified variables in the included file to be overwritten by a later inclusion of the same file. Another problem that arises is the clashing of function names should they exist in the inclusion file. And thus I introduce the next function, `require_once()`.

require_once()

The `require_once()` function ensures that the insertion file is included only once in your script. After `require_once()` is encountered, any subsequent attempts to include the same file will be ignored. Its syntax follows:

```
require_once(file insertion_file)
```

Other than the verification procedure of `require_once()`, all other aspects of the function are the same as for `require()`.

You will probably use these functions extensively as your Web applications grow in size. You will regularly see these functions in my examples throughout the

remainder of this book in order to eliminate code redundancies. The first practical use of these functions occurs in the next section, where I introduce basic template construction strategies.

Building Components

When referring to the structure of a typical Web page, I generally like to break it up into three distinct parts: header, footer, and body. Usually, most well-organized Web sites have a top section that remains largely unchanged; a middle section that displays the requested content, thus changing regularly; and finally a bottom section containing copyright and general link information that, like the header, generally does not change. Don't get me wrong; I'm not trying to stifle creativity. I've seen many fantastic sites that do not follow this structure. I'm just attempting to set up a framework from which we can begin.

The Header

One thing I like to use in almost all of my PHP-enabled Web sites is a header file, such as the one shown in Listing 9-5. This file holds several pieces of information that will be applied sitewide, such as the title, contact information, and actual initial HTML components of the page.

Listing 9-5: A sample header file

```
<?
// filename: header.tpl
// purpose: site header file for PhpRecipes site
// date: August 22, 2000

$site_name = "PHPRecipes";
$site_email = "wjgilmore@hotmail.com";
$site_path = "http://localhost/phprecipes";
?>
<html>
<head>
<title> <?=$site_name;?> </title>
</head>
<body bgcolor="#7b8079" text="#ffffff" link="#e7d387" alink="#e7d387"
vlink="#e7f0e4">
<table width = "95%" cellpadding="0" cellspacing="0" border="1">
     <tr>
     <td valign = "top">
     PHPRecipes
     </td>
```

```
    <td valign = "top" align="right">
    <?
    // output current date and time
    print date ("F d, h:i a");
    ?>
    </td>
    </tr>
</table>
```

You may often want to ensure that unwanted visitors do not view your included files, particularly if they hold information such as access passwords. In Apache, you can wholly deny the viewing of certain files by modifying your http.conf or htaccess file. This is an example of how you can prevent the viewing of any file with a .tpl extension:

```
<Files "*.tpl">
  Order allow,deny
  Allow from 127.0.0.1
  Deny from all
</Files>
```

NOTE *PHP and site security are discussed in more detail in Chapter 16.*

The Footer

What is typically deemed the "footer" of a site is the information at the bottom of site pages, generally the contact, linking, and copyright information. This information can be placed in a single file and included as a template just as easily as the header information can. Consider the need to change the copyright information to read "Copyright © 2000-2001" You have two choices: spend your New Year's Eve frantically changing hundreds of static pages *or* use a footer template like the one in Listing 9-6. Make one simple change and voilà! Back to the festivities.

Listing 9-6: A sample footer file (footer.tpl)

```
<table width="95%" cellspacing="0" cellpadding="0" border="1">
<tr><td valign="top" align="middle">
Copyright &copy; 2000 PHPRecipes. All rights reserved.<br>
<a href = "mailto:<?=$site_email;?>">contact</a> | <a href =
"<?=$site_path;?>/privacy.php">your privacy</a>
</td></tr>
</table>
</body>
</html>
```

Take note that I am using one of the global variables (`$site_email`) within the footer file. This is because that variable will propagate throughout the entire page, since it is assumed that the header.tpl and footer.tpl files will be assimilated into one cohesive page. Also, notice that I output `$site_path` in the "privacy" link. I always want to use a complete path to any link in a template file because if I use this footer in any child directories, the path would not be correct if I were only to use privacy.php as the link URL.

The Body

The page body connects the header to the footer. The body section of a Web document is basically the "meat-and-bones" section of the page—that is, the page that the readers care about. Sure, the header is cool, the footer is helpful, but it is the body that keeps readers returning. Although I can't provide any pointers as to the content of your page structure, I can help in terms of your ease of page administration by providing Listing 9-7.

Listing 9-7: A simple body section (index_body.tpl)

```
<table width="95%" cellspacing="0" cellpadding="0" border="1">
<tr>
<td valign="top" width="25%">
<a href = "<?=$site_path;?>/tutorials.php">tutorials</a> <br>
<a href = "<?=$site_path;?>/articles.php">articles</a> <br>
<a href = "<?=$site_path;?>/scripts.php">scripts</a> <br>
<a href = "<?=$site_path;?>/contact.php">contact</a> <br>
</td>
<td valign="top" width="75%">
Welcome to PHPRecipes, the starting place for PHP scripts, tutorials, and
information about gourmet cooking!
</td>
</tr>
</table>
```

Putting It Together: Incorporating the Header, Footer, and Body

My feelings are perhaps best phrased as Colonel "Hannibal" Smith (George Peppard) put it on the famous *A-Team* television show, "I love it when a good plan comes together." In my nerdy way, I feel the same when I see several template files come together to form a complete Web document. Combining the three document sections, header.tpl, index_body.tpl, footer.tpl, you can quickly build a basic page like the one in Listing 9-8.

Listing 9-8: Various includes compiled together to produce index.php

```
<?
// file: index.php
// purpose: Home page of PHPRecipes
// date: August 23, 2000
// Include the header
include ("header.tpl");
// Include the index body
include ("index_body.tpl");
// Include the footer
include ("footer.tpl");
?>
```

How about that? Three simple commands, and the page is built. Check out the resulting page in Listing 9-9.

Listing 9-9: Resulting HTML constructed from Listing 9-8 (index.php)

```
<html>
<head>
<title> PHPRecipes </title>
</head>
<body bgcolor="#7b8079" text="#ffffff" link="#e7d387" alink="#e7d387"
vlink="#e7f0e4">
<table width = "95%" cellpadding="0" cellspacing="0" border="1">
   <tr>
   <td valign = "top">
   PHP Recipes
   </td>
   <td valign = "top" align="right">
```

```
   August 23, 03:17 pm
   </td>
   </tr>
</table>
<table width="95%" cellspacing="0" cellpadding="0" border="1">
<tr>
<td valign="top" width="25%">
<a href = "http://localhost/phprecipes/tutorials.php">tutorials</a> <br>
<a href = "http://localhost/phprecipes/articles.php">articles</a> <br>
<a href = "http://localhost/phprecipes/scripts.php">scripts</a> <br>
<a href = "http://localhost/phprecipes/contact.php">contact</a> <br>
</td>
<td valign="top" width="75%">
Welcome to PHPRecipes, the starting place for PHP scripts, tutorials, and gourmet
cooking tips and recipes!
</td>
</tr>
</table><table width="95%" cellspacing="0" cellpadding="0" border="1">
<tr><td valign="top" align="middle">
Copyright &copy; 2000 PHPRecipes. All rights reserved.<br>
<a href = "mailto:wjgilmore@hotmail.com">contact</a> | <a href =
"http://localhost/phprecipes/privacy.php">your privacy</a>
</td></tr>
</table>
</body>
</html>
```

Figure 9-1 shows you the resulting page as viewed in the browser. Although I detest table borders, I set them to 1 so that you can more easily differentiate the three sections of the page.

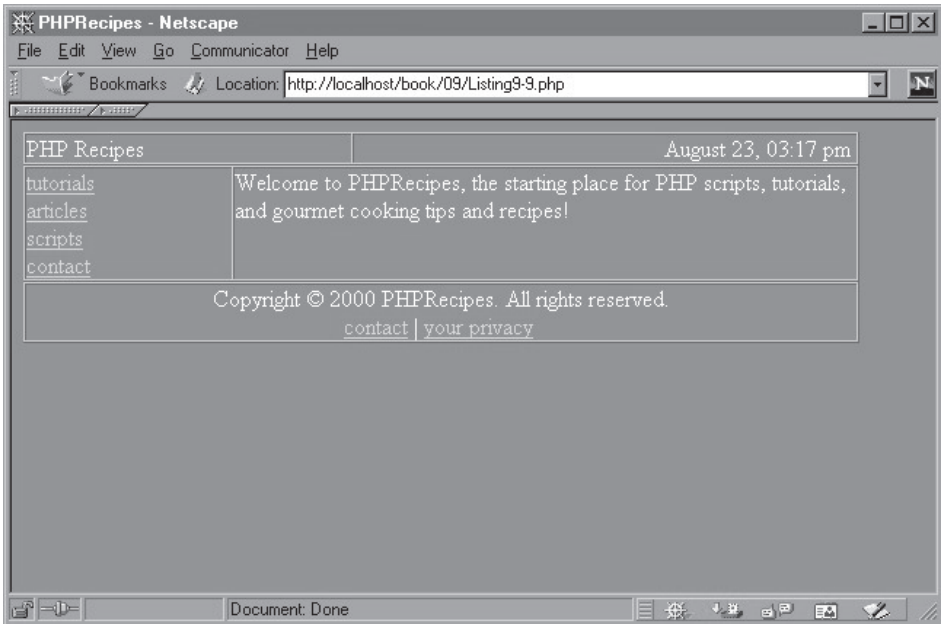

Figure 9-1. Resulting page as constructed from Listing 9-8

Optimizing Your Site's Templates

A second, and arguably preferred, method of using your templates is to store them in functions, placed in a single file. This further organizes your template, making a "template of templates." I also call this my initialization file, as I tend to store other useful information in it. Since you already have been exposed to a relatively lengthy header and footer example, I'll abbreviate the ones in Listings 9-10 and 9-11 for the sake of illustrating this new idea.

Listing 9-10: Optimized site template (site_init.tpl)

```
<?
// filename: site_init.tpl
// purpose: PhpRecipes Initialization file.
// date: August 22, 2000

$site_name = "PHPRecipes";
$site_email = "wjgilmore@hotmail.com";
$site_path = "http://localhost/phprecipes";

function show_header($site_name) {
?>
<html>
```

```
<head>
<title> <? print $site_name; ?> </title>
</head>
<body bgcolor="#7b8079" text="#ffffff" link="#e7d387" alink="#e7d387"
vlink="#e7f0e4">
This is the header
<hr>
<?
}
function show_footer() {
?>
<hr>
This Is the footer
</body>
</html>
<?
}
?>
```

Listing 9-11: Using the initialization file

```
<?
// Include site Initialization Information
include("site_init.tpl");
// display the header
show_header($site_name);
?>
This is some body information
<?
// display the footer
show_footer();
?>
```

Using functions further condenses the code and number of templates needed, ultimately allowing you to more efficiently administer your site. This strategy also makes it easier to reuse your code to build other sites without having to keep track of a number of involved files.

Project: Build a Page Generator

Although large parts of the Web sites I build make use of database information to display content, there are always a few pages that aren't going to change much.

Some of these pages may contain information about the development team, contact information, advertising information, and so on. You get the picture. I generally store this "static" information in its own folder and use a PHP script to pull it to the Web page on request. Of course, since this information is static, you may be asking yourself why you should even employ the use of a PHP script. Why not just use plain old HTML pages? The advantage of using PHP is that you can take advantage of the templates, just inserting the static part as necessary.

The link used to call the various static files is dynamic. Its general form is:

```
<a href = "<?=$site_path;?>/static.php?content=$content">Static Page Name</a>
```

To begin, create your various static pages. For sake of simplicity, I'll create three of them: About This Site (Listing 9-12), Advertising Information (Listing 9-13), and Contact Us (Listing 9-14).

Listing 9-12: About This Site (about.html)

```
<h3>About PHPRecipes</h3>
What programmer doesn't mix all night programming with gourmet cookies? Here at
PHPRecipes, hardly a night goes by without one of our coders mixing a little bit
of HTML with a tasty plate of Portobello Mushrooms or even Fondue. So we decided
to bring you the best of what we love most: PHP and food!
<p>
That's right, readers. Tutorials, scripts, souffles and more. <i>Only</i> at
PHPRecipes.
```

Listing 9-13: Advertising Information (advert_info.html)

```
<h3>Advertising Information</h3>
Regardless of whether they come to learn the latest PHP techniques or for brushing
up on how to bake chicken, you can bet our readers are decision makers. They are
the Industry professionals who make decisions about what their company purchases.
For advertising information, contact <a href = "mailto:ads@phprecipes.com
">ads@phprecipes.com</a>.
```

Listing 9-14 Contact Us (contact.html)

```
<h3>Contact Us</h3>
Have a coding tip? <br>
Know the perfect topping for candied yams?<br>
 Let us know! Contact the team at <a href =
"mailto:theteam@phprecipes.com">team@phprecipes.com</a>.
```

Now you will design the page that will house the requested information, entitled "static.php". This file acts as the aggregator of the various components of a page on our site and makes use of the site_init.tpl file, shown in Listing 9-15.

Listing 9-15: Page aggregator (static.php)

```
<?
// file: static.php
// purpose: display various requested static pages.
// IMPORTANT: It Is assumed that "site_init.tpl" and all of the static files are
located in the same directory.
// load functions and site variables
include("site_init.tpl");
// display the page header
show_header($site_name);
// display the requested content
include("$content.html");
// display the page footer
show_footer();
?>
```

OK, now you're ready to implement this script. Simply place the correct link reference relative to the page, as shown here:

```
<a href = "static.php?content=about">About This Site</a><br>
<a href = "static.php?content=advert_info">Advertising Information</a><br>
<a href = "static.php?content=contact">Contact Us</a><br>
```

Clicking any of the links will take you to the respective static page, embedded in static.php!

What's Next?

This chapter introduced you to the heart of what PHP was intended to do in the first place: dynamic Web page generation. In this chapter, you learned how to do the following:

- Manipulate URLs
- Generate dynamic content
- Include and build basic templates

The project concluding the chapter illustrated how you could build a page generator that would pull static pages into a larger template structure, making it ever so easy for you to maintain large numbers of static HTML pages.

The next chapter builds on this foundation significantly, introducing how PHP can be used in conjunction with HTML forms, adding a whole new degree of user interactivity into your site. Then, it's onward to databasing! What an exciting few chapters these are going to be!