PHPNuke

Setting the Sails

We've climbed aboard and begun our journey to distant lands, leaving familiar shores behind. The task seems daunting, but we have the tools and knowledge to reach our goal. A journey of a thousand miles begins with a single step, and we have just taken that step.

BY JAMES MOHR

www.photocase.de

Despite being quick to install, PHPNuke is a very powerful system with a lot of features. Basically all of these are useful right from the start, but you really do not begin to appreciate them until you start configuring your system to more closely reflect your needs.

Testing the Water

In the first article we talked about the basic structure and functionality of PHPNuke. For many sites this might be enough.

In this installment we will be looking into some more details of how to administer the basic components, how they interact and how easy it is to add your own. Although this is probably not necessary for many sites, learning some of the details of PHPNuke's internals will better enable you to add components, as well as take advantage of the functionality that PHPNuke provides.

One might think of PHPNuke as a traditional "3-column" portal (which it is referred to in many places). This typically means people forget about the header and footer. These are two aspects that are useful in providing information to your visitors.

To fully understand what I mean, let's first talk about the concept of a 3-column portal. Typically you have the primary data area in the center column, with the columns on each side providing supplemental information, navigation and similar features. Often, you find additional information at the top of the page (the header) and at the bottom (the footer). In the case of PHPNuke, you can configure both of these as needed.

The 3-column format that that PHPNuke provides is not static. You can easily configure it so that there are just two columns (e.g. content and navigation). In fact, a number of themes have a default of just two columns.

Repeating on almost every page, you will find a number of different functions that are grouped together. These are called blocks and are configured in the administration module. For the most part, blocks appear on the sides of your page, but you can configure blocks to appear at the top or bottom. For example, the *modules* block (list of active modules) is in the left-hand column by default and the *login* block is in the left-hand column.

You can also change the relationship of the blocks to each other. In figure 1, we see an excerpt from the block administration panel. Modules are grouped by their location on the pages, which is listed after the block's name (center up, center down, left or right). Following this is the block's "weight", which more or less means the order in which the blocks appear. By clicking the arrows, you

change a block's weight and thus its relative position.

When you first configure your system and move blocks around, you will find that blocks end up with the same weight. This is corrected by clicking the link "Fix Block's Weight Conflicts". This does not always put things in the order you want, but you can simply click the arrows again to put this right.

Next, you see whether the block is active or not, followed by the class of user that can see the block: administrators, registered users, anonymous users and all users. Certain blocks are only visible if you are an administrator, or only if you are a registered user. Some, like the user login, are only visible if you are an anonymous user.

On the right-hand side, you see the basic administration functions, such as activating, deleting and editing the blocks. In this case, editing means defining things like who can see it, and so on.

At the bottom of the page (not depicted) you have a form labeled "Add a New Block", which is used just for that. Here, you define the basic behavior of the block, where it is located and even how long it is valid. We'll get into the details of adding blocks shortly.

Modules are the next key aspect of your system. Clicking the *Modules* icon in the administration panel brings you to a page that is similar to the blocks page. The big difference is that you do not put the modules in any particular location. Instead the output is always in the center column.

Unlike for blocks, there is no form to create new modules. Modules are added by creating a new sub-directory below the *modules* directory with the appropriate files. Like the blocks, you can determine which users can see the module. We'll also get into the details of adding modules shortly.

Several of the other administration options allow you to administer individual modules. One module that is very common on PHPNuke sites is the Forum module. Currently, PHPNuke is using a port of phpBB. This was an important factor in my decision to switch to PHPNuke, as I had already been running phpBB for a while and I did not want to lose my users and their posts. Fortunately, there is a conversion script which ran flawlessly.

The management of your forums is a lot more complex than the management of other modules. Here you need to find the various forums, sub-forums, and so forth. In contrast to other administrative functions, administration of phpBB takes place within the module itself. In other cases, there are files in the *admin/modules* directory for each module.

Before you put your site online, you should modify the *meta.php* file, which contains the metatags that PHPNuke automation adds to each page. An important metatag for most webmasters is the KEYWORDS tag. In general, these are words that describe the content of the page and are often used by search engines when people are searching for a particular topic. Particularly if your site is not related to computers, most of the words are probably not valid for your site. Leaving the keywords as they are makes your site appear in search results where it shouldn't.

The Flow of Things

As I mentioned before, the key file is *modules.php* in your PHPNuke root directory. When you first look at it, it seems to be doing a lot. However, after closer inspection you will find that most of what it does only occurs in special circumstances, for example in case of errors.

Instead of doing the work itself, the *modules.php* script calls *index.php* for the respective module. Therefore, it is extremely important that the *index.php* script has the required components and structure. This is compounded by the fact that the theme script (e.g. *themes/Sand_Journey/theme.php*) also does some of the work.

At first this appears to be convoluted and perhaps unnecessarily complex. However, you need to consider the fact that one aspect of a particular theme might be a different header layout (which is actually the case with some default themes) or changes to other characteristics of the pages. So, despite being complex, this is useful if your personal theme changes the layout, despite complicating things a bit.

Making modifications at this level goes far beyond the scope of this article. Instead, we'll stick to just the basics.

Perhaps the two most important aspects of your module's *theme.php* file are the inclusion of the *header.php* and *footer.php* files, to ensure that the page is displayed correctly. The *header.php* file creates the actual header, the left column of blocks and the top center blocks. The *footer.php* file creates the bottom center blocks, the right column of blocks and the page footer. The actual content is created within the module itself.

A very basic simplistic module would look like this:

```
<?php
   $index = 1;
 include("header.php");
   OpenTable();
   print "Welcome to PHPNuke\n";
 CloseTable();
 include("footer.php");
?>
```

(Note: That the *$index = 1;* is necessary to ensure that the right block is included.)

In addition to including *header.php* and *footer.php* files, we have two functions which are defined with the theme: *OpenTable()* and *CloseTable()*. These are what actually encloses the page content in a bordered box. You could leave these functions out of your module, but it would make the pages look substantially different than other modules.

Finally we have the the *print* function and as its name implies, it simply prints out text. The nice thing is that that this text can also be HTML code. This means you could create a whole page of HTML code and simply add it as a module using the PHP print function, something like this:

Title	Position	Weight		Type	Status	Visible to	Functions
Information	▷ Center Down ◁	1		HTML	Active	All Visitors	[Edit | Deactivate | Delete | Show]
Modules	◁ Left	1	⇩	FILE	Active	All Visitors	[Edit | Deactivate | Delete | Show]
Administration	◁ Left	2	⇧ ⇩	SYSTEM	Active	Administrators Only	[Edit | Deactivate | Delete | Show]
Who's Online	◁ Left	3	⇧ ⇩	FILE	Active	All Visitors	[Edit | Deactivate | Delete | Show]
Search	◁ Left	4	⇧ ⇩	FILE	*Inactive*	All Visitors	[Edit | Activate | Delete | Show]
Languages	◁ Left	5	⇧ ⇩	FILE	*Inactive*	All Visitors	[Edit | Activate | Delete | Show]

Figure 1: Excerpt from the block administration panel.

```
print "<H1>Welcome to ➥
PHPNuke</H1>
This is a text module.<BR>
I hope you like it.<BR>
\n";
```

Everything within the double-quotes is printed including the new-lines. Since the browser interprets the HTML code, you end up with a header and two separate lines. Note that because PHP uses double-quotes to delimit text, you need to escape double-quotes in your HTML code with a back-slash. (\")

On the surface this appears to be a very simplistic example (and it is). However, you can include most anything that you normally would in an HTML page. This might be something as simple as a list of your favorite musicians and links to their homepage, recipes you want to share, and so forth.

Using this example, you would need a single module for each new file you want to display. Fortunately, there is an easy way around this by taking advantage of the fact that PHP automatically stores variables in your query string and can then be used in your module.

For example, let's assume you have a module for your recipes. It might be accessed like this:

```
modules.php?name=Recipes
```

You could then add a variable like this:

```
modules.php?name=Recipes&recipe➥
=chicken_soup
```

Now within your module file (i.e. *index.php*) you can use the variable *recipe* which contains the value "chicken_soup". Now let's assume you have a directory underneath your document root called *recipes*, which contain one file for each recipe. We could print out the content of the page like this:

```
$file = "/document/root/recipes➥
/$recipe";
$page=join("",file($file));
print $page;
```

First, we assigned the path to our recipe file to the variable *$file*. Since the variable *$recipe* was passed to the PHP script in the query string, we can automatically access it through the *$recipe* variable. In the next line, we assign the *contents* of the file to the variable *$page*. then we simply print out the contents of the page. There are other ways of reading files and outputting them. However, I have found this to be efficient and, as we'll see in the next installment, very useful if you want to process the page once you have loaded it.

Note that these three lines *replace* the print statement in our first example and do *not* represent the whole file.

On the shores of your site

As I mentioned earlier, the sides of PHPNuke page, have blocks of information that typically repeat on every page. There are three kinds of blocks:

- RSS/RDF – Display news from other sites using a standard format.
- Text – Contain text (including HTML), which is simply inserted into the block.
- Script – PHP Scripts, which can perform most any function other PHP scripts can.

Because of space limitations I am going skip talking about the RSS/RDF blocks and concentrate just on the other two.

Text blocks are good for static text that does not change between pages. The content of text blocks is actually stored within the database and retrieved each time the page is loaded.

Because script blocks can do whatever any other PHP script can, they can also display static text. You might think that this a lot of extra work when you can simply add text blocks through the administration panel. However, I typically use script blocks to display static text. This way I can make changes to the content without having to log into the administration panel. Also, I know that all of my blocks create their content in the same way, which makes administration easier (for me, at least).

Like modules, script blocks are available as soon as you create the necessary file. Blocks are stored as individual files in the *blocks* directory and always have the *block-* prefix. For example, I have a block that displays a table of contents for my tutorial. This file is called *block-TutorialMenu.php*. When I drop this file into the *blocks* directory, it is immediately accessible through the administration panel. By default, the new module is inactive.

The key to script blocks is the *$content* variable. Whatever this variable is set to when the script exits, will be displayed within the block. To create a very simple script that simply displays some text, your block file might look like this:

```
<?php
$content = "Welcome to my site";
?>
```

By changing the *$content* variable based on different criteria, you can create a block that changes its contents. For example, I have a block that first generates a random number between 1 and 5. I then use a *switch* statement to set the *$content* variable based on which number was generated.

You can take this one step further as in the case of several existing blocks and pull information out of a database, for example, the *block-Last_10_Articles.php* which shows the last 10 articles added to the site. The "Did You Know?" block on my Linux tutorial randomly takes an entry from the "concepts" table in my database and displays it with a link to the appropriate page.

SlashOcean & other themes

One important aspect of any portal system (at least in my experience) is the ability to configure it to your own personal tastes and preferences. This is the basic concept behind themes. As with other kinds of software, a portal theme lets you define colors, layout and often even which graphics appear. By default PHPNuke provides over a dozen different themes and provides an easy mechanism to change these, as well as add new ones.

Underneath the *themes* directory there is a sub-directory for each theme. The *style* sub-directory contains the CSS file for the respective theme, which includes font sizes and colors, and so forth. I have found that the default sizes are often too small for larger monitors. On my site, I created copies of each of the themes and increased the size of the fonts in increments of 2. I ended up with themes like SandJourney Large and SandJourney VeryLarge where the only difference was the size of the fonts.

The *images* directory contains theme-specific images. Here you might have a site logo that looks different with different themes. By default PHPNuke will use the file *link-logo.gif*. So by naming your logo the same, you can copy it into all of the theme directories, and then change it as needed.

The bulk of the work is done by the *theme.php* file, including the overall layout, header, footer, and even some of the basic colors. This is an important thing to note, since one might expect that the CSS file would be responsible for defining all of the colors. However, the background colors and table edges (among other things) for various tables are defined in the *theme.php* file. So, if you are making changes to a theme and cannot figure out where to change specific colors, this is the place to look.

In the *theme.php* file you also find the *OpenTable()* and *CloseTable()* functions we mentioned earlier. This is because the appearance of these tables is theme dependent.

Note that many of the themes consist of more than just the *theme.php*. Some, such as the 3-D-Fantasy, NukeNews and Odyssey themes contain a number of additional files. In general, these are HTML "template" files that are more or less inserted into the page at the appropriate location. The site logo is referenced in one such file, so if you change the location (like I did), you will need to change these files as well.

PHPNuke takes the concept of themes one step further and allows you to have modules that are different depending on the theme. This can be very useful if you have themes that provided different layouts. For example, one theme might provide blocks only on the left side. Blocks that are configured for the right side are not visible.

Within each theme, there is a *modules* sub-directory, where you create the sub-directory for the specific module. By default, you will find the "Addon Sample" module is present for most (if not all) existing themes. When you call up this module, you will see that the text at the top of the content block is slightly different and looks like this:

```
Addon Sample File (index.php) ➘
CALLED FROM MODULE_NAME
```

To add other modules to specific themes, you start by *copying* the entire module directory into the appropriate theme directory. Next you need to change file and directory paths to make sure they reference the new location. For example, you might have a configuration file for your module that normal resides in the */modules/YourModule* directory. You would then need to change references to point to */themes/ThemeName/modules/YourModule*.

Note that if you want to change the themes for your forums, you don't need to create a new forum module for that specific theme. Instead, you create a theme within the Forums. This is a little cumbersome, but it helps keep the codebase simpler. As mentioned previously, the PHPNuke forums are a port of phpBB. If the phpBB code were much more integrated, a great many changes would be needed.

Making the waters safer

You may have noticed that in most directories there is an empty index.html file. This serves two purposes. First, you will probably not want your visitors to list the contents of the PHPNuke directories, so you might disable it in the virtual host configuration or in a .htaccess file with *Options -Indexes*. Should a visitor input the name of a directory without a filename, the system with try to load either index.html or index.php. If neither one of these is present, you would get an error. Personally, I think that even a blank page is better than a system error message saying that I am not allowed to do something.

Another issue is the fact that the index.html is typically loaded first. If you do have an index.php file (which is the case with the modules), you typically do not want your visitors to load these files directly; instead you add the index.html, so it is loaded first. Should the index.php be loaded first, or if there is no index.html, there is a way of keeping visitors from accessing the file directly. We'll get to this in the next installment.

In your PHPNuke root directory you will find a *robots.txt* that contains a list of directories that robots (i.e. search engines) should ignore. Unfortunately, some robots ignore this file completely and scan your entire site. To limit this, I created an *.htaccess* file, which looks for a number of specific robots and *redirects* them to a special page.

The robots I try to block are not ones used by major search engines like Google, because these are generally well behaved. Instead, I include download tools like wget as I don't want people downloading the entire site at once. However, many of these tools can make it look like they are some other program (even a real browser) so this does not stop everyone.

One nasty bug that PHPNuke had was the ability to create an admistrator account. This meant a hacker had complete control over all of the functions available through the administration panel. By limiting access to the *admin* directory, you can limit who can do what. Assuming your provider allows it, you can create entries in the .htaccess file that limits access to only specific hosts. This needs to be changed each time you add an administrator, but keeps unwanted administrators away from the system. ■

INFO

[1] PHPNuke system in action: *http://www.linux-tutorial.info*

[2] The PHPNuke Site: *http://www.phpnuke.org*

[3] Home of a *huge* PHPNuke forum and many other resources: *http://www.nukecops.com*

[4] Wide range of fixes for various release, including a forum: *http://www.nukefixes.com*

[5] A number of different PHPNuke forums: *http://www.nukeforums.com*

[6] A wide range of resources for PHPNuke: *http://www.nukeresources.com*

[7] Security related issues and fixes: *http://www.nukesecurity.com*

THE AUTHOR

James Mohr is responsible for the monitoring of several datacenters for a business solutions provider in Coburg, Germany. In addition to running the Linux Tutorial web site (http://www.linux-tutorial.info), James is the author of several books and dozens of articles on a wide range of topics.