

Creating a PHPNuke module

Cooking with Nuke

PHPNuke lets you create a master-piece or disaster depending on what tools you use and how you go about it. Even with the simplest tools and methods, PHPNuke enables you to easily create dynamic web sites that can access text-based data as well as enterprise databases.

BY JAMES MOHR

Even before finishing configuring your website, you may have already realized the need for you to develop your own PHP modules. As we discussed in the previous installment, there are very simple ways of creating modules without having to do much with PHP. However, if you want to provide modules with even a slight bit of complexity, you will need to do some more programming.

Gathering the Right Ingredients

As I mentioned, if you know Perl, then moving to PHP is fairly easy (this also applies to C and similar languages). Rather than turning this into a PHP tutorial, I am going to assume you already have a basic foundation in Perl, C, or a similar language because the code we are going to talk about is very simple.

Another thing we need to assume is a basic understanding of SQL. We are not going to build any complex queries, but you should at least know some very basic syntax, such as how queries are constructed.

One thing we haven't yet addressed is the actual editing of code. Although you could use any editor that saves ASCII

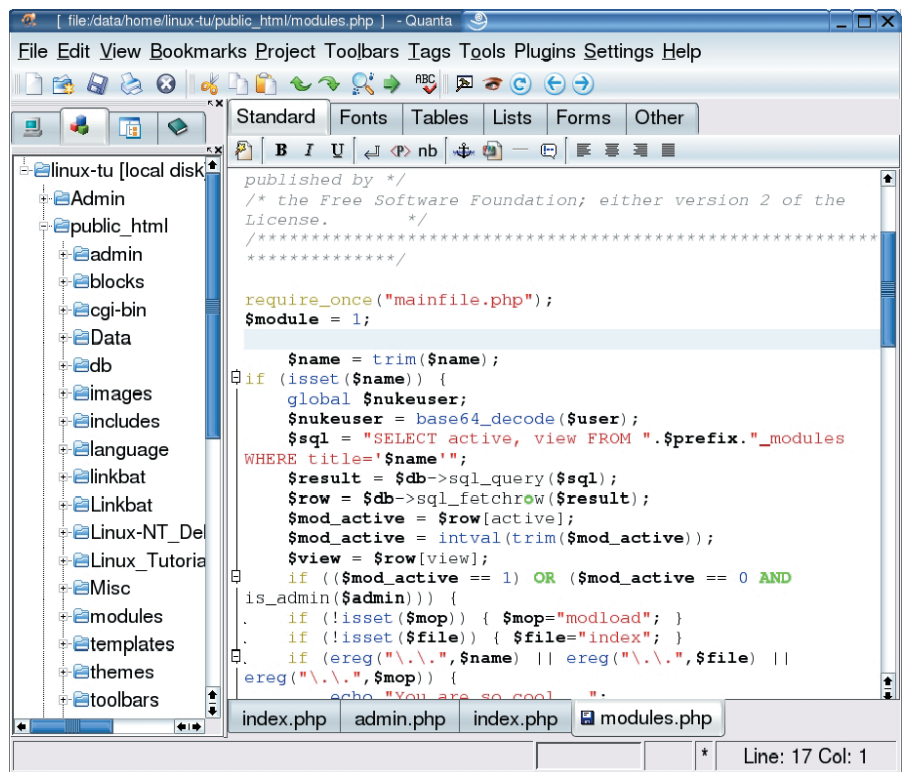


Figure 1: The Quanta Plus Web development environment.

text (such as *vi*), you are probably going to want something more, and there are a number of tools available to help you do your programming. For all of my web development, be it HTML, Perl or PHP, I use Quanta Plus (<http://quanta.sourceforge.net/>). In fact, Quanta Plus is designed for developing in dozens of different programming languages.

One very useful feature is Quanta's ability to collapse and expand blocks of code as needed. For example, *while* loops, *if*-blocks, and functions can be collapsed so that you have a better overview of the code. This is enhanced by Quanta Plus's ability to highlight your code in different colors. For example, light-gray for comments, blue for functions, and black for variables. (See Figure 1.) The colors are configurable and depend on the type of code (for example, HTML is different from PHP).

More than just an editor, Quanta Plus has many project management features.

For example, files can be edited locally and then uploaded to your web server using a number of different protocols, such as ftp, fish, webdav(s), and many more. Quanta Plus also keeps track of which files have been modified, allowing you to upload just those files that have been changed. Therefore, you can develop on a test server and then upload to a live server when you are sure your code is ready.

Adding to its advanced development features, Quanta Plus is integrated with the Concurrent Versions System (CVS) through the Cervisia program (which also works stand-alone). Therefore, should something go terribly wrong with the code on your test system, you can easily revert to a previous version.

Quanta is also highly configurable. Among other things, Quanta Plus allows you to create so-called actions. These can be anything, from special tags and text blocks, to program output which is

inserted directly into your code. You also can add any number of toolbars to your systems, which can include existing actions as well as those you create yourself. The toolbars can be also be exported and imported, allowing everyone working on a project to have the same tools.

Getting Ready

Before you begin building your ultimate module, you need to lay down a firm foundation. Many of the basic principles apply to any programming language or project, such as using self-explanatory variable names and commenting your code (something tragically missing from the PHPNuke code).

Unless you explicitly specify another file, *modules.php* will automatically load the *index.php* file. A customized file may be useful in some cases (like very complex modules), but for now, let's just work with the default.

Other files, such as images, are included using the *modules.php* file. Therefore, paths need to be defined relative to the location of *modules.php* rather than that of your modules. Typically, *modules.php* resides in the root directory of your server, so you can use an absolute path (i.e., */images*). Even if you put your images within the module's directory, you can still use an absolute path (i.e., */modules/modulename/images*).

Accessing an SQL Database

As we mentioned in the first installment, one advantage of PHPNuke is its data abstraction layer, which allows you to access data from any SQL without needing to write database-specific code. This feature allows you to use the same code, no matter what database you are using.

Tables within the Nuke database always have a prefix that is defined in the *config.php* file. The default prefix is *nuke*, but I have used other prefixes. In fact, you need to use a different prefix if you have multiple Nuke instances on a single machine. The default table containing all of the user data is thus *nuke_users*.

One point of contention is where to put data that is not part of the standard PHPNuke database. Some people will tell you to include it in the same database as the default PHPNuke tables. This does

make administration a *little* easier in some cases.

The other group of people (which includes myself) says to keep your data separate from the Nuke tables and create a separate database. I have over 60 of my own tables and including them with the 90 or so Nuke tables gets confusing really fast. By splitting the data like this, I also avoid potential problems if I ever need to upgrade PHPNuke or move my data to another database (i.e., from MySQL to Oracle).

The disadvantage is that you have to deal with making the connection to the database yourself. If the data is stored in the nuke tables, then the connection is already made for you through the PHPNuke files. With just a few tables it might be worth having just a single table and thus less programming overhead.

If you do this, I would *highly* recommend that you use a prefix different from the default. Remember, as we mentioned in the first installment, when you install PHPNuke, you define a prefix for the PHPNuke tables. By default this is "nuke." So, I might define my non-Nuke table with a prefix of "jimmo". My recipes table would then be *jimmo_recipes*.

To avoid turning this into an article on administering databases, I will go with the simpler case where all the data is in a single table. Amazingly enough, we can access an SQL database with just a few simple lines. For example, if we wanted to pull out the name of our first recipe, we might have something like this:

```
$sql_a = "SELECT name
FROM jimmo_recipes
WHERE recipe_id='1' ";
$result_a = $db->
    sql_query($sql_a);
$row_a = $db->
    sql_fetchrow($result_a);
print "Recipe name:
    ".$row_a[name]."\n";
```

By replacing the *print* in the first example of the previous article, we would have a single line showing us the recipe's name.

In the first line of our example, we define the query that we are going to make. This is then used in the second line to make the actual query. The results of that query are not actually

what is returned. Instead, a pointer is returned. In the next line, we use the pointer to fetch a single row (with all of the fields specified), which is then returned to the array *\$row_a*.

In the last line, we print out the contents of a *single* element of the array. In this case, the name field. Note how we actually use the name of the field as the offset into the array and not something like *\$row_a[0]*. If we had changed the original query to something like *SELECT name, ingredients...*, then we could have also used something like this *\$row_a[ingredients]*.

To be able to access every row, we make a very simple modification:

```
while ( $row_a = $db->
    sql_fetchrow($result_a) ) {
    print "Recipe name:
    ".$row_a[name]."\n";
}
```

Like in other languages, the *while* statement repeats the specific block for as long as the statement is true. In this case, the condition we are checking is whether we were able to retrieve a new row from the database. Each time through we print out the name of the recipe.

Creating an Application

With a couple of simple changes, we can turn the list of names into hyperlinks, which lead you to the details of the respective recipe. There are a couple of ways of doing this. I think the simplest is to include all of the functionality within a single file (i.e., *index.php*), which is the case in many of the default modules (at least the simpler ones). To do this we wrap the code we have written so far into a *function*.

Like functions in other programming languages, a PHP function is simply a separate block of code that can be called from other places, passed values, and which returns values. So, we might have a function creating the list of recipes that looks like this:

```
function list() {
    ... body of function ...
}
```

We would also need a second function to display the details of the recipe. In order

to know which recipe to display we would have to pass the recipe id (recipe name) to the function. Something like this:

```
function show_recipe($recid) {
    ... body of function ...
}
```

Now, within the function *show_recipe*, we can simply use the *\$recid* variable where we need it, such as including it with an SQL query. The details of this function we will get to shortly.

When passing variables like this, I like to use unique IDs rather than text. It's easy enough to automatically add them to the database and retrieve them later.

Adding the Switch

Next, we need to make use of the *switch* statement. As its name implies, it is used to "switch" the behavior of the program. In this case, we switch based on the value of an additional variable in the query string that tells us which operation to perform. Doing so might give us something like this:

```
switch ($op) {
    case "show":
        show_recipe($recid);
        break;

    default:
        list();
        break;
}
```

Within the *switch* block, the *case* statement directs the flow based on the value of the given variable (in this case *\$op*). If the *\$op* variable equals "show", then the variable *\$recid* is passed to the function *show_recipe*. This then used within the *show_recipe* function.

At the bottom we have one block labeled *default*. This is used if the value of *\$op* does not match anything else (even if it's empty, which is often the default case). At the end of each block, we have a *break* statement to tell the program to break out of the *switch* block.

If we wanted to, we could add any number of case statements that in turn call various functions. We can also pass values to the functions, just as in most other languages.

Within the body of the *show_recipe* function, the code we created is very similar to that in the first example above. Only the SQL query is different:

```
$sql_a = "SELECT
    name,ingredients,instructions
FROM recipes
where recipe_id =".$recid;
```

Here, we are pulling the values *name*, *ingredients*, and *instructions* from the one row in the *recipes* table that matches the *recipe_id*. We then load that single row into an array, as shown above, and display it.

Displaying the Fields

Displaying the fields basically works the same way as before. However, now we want to format it a little. Since the output is actually interpreted by your browser, you can include HTML code in the text your PHP program prints out and format it any way you want. This might result in code that looks like this:

```
print "<H1>".$row_a[name].
"</H1>\n"
print "<H2>Ingredients</H2>\n"
print $row_a[ingredients]."\n";
print "<H2>instructions</H2>\n"
print $row_a[instructions]."\n";
```

At this point we need to backtrack a little. We previously created a list of recipe names, but we don't yet have a way to create the links to access the recipe details. To do this we simply change the print statement a little, which gives us this:

```
print "<a href=/modules.php?
name=Recipes&
op=show&
recid=".$row_a[recipe_id].">".
$row_a[name]."</A><BR>\n";
```

As you can see, we are calling the *modules.php* like we normally would and passing it the name of the module ("Recipes" in this case). If we stopped here, no other variables would be set; the *switch* statement would go into the default case and display the list.

In this case, however, we are setting the *op* to "show", thus sending the program into the *show_recipe* function. Note

that the *recipe_id* is being pulled from the database, along with the recipe name, and these values are simply inserted at the appropriate location.

Admittedly, we could have done a lot more formatting to both the list and the output. We could have also created the query so that it sorted the list of recipes by the recipe name. However, that goes into a bit more detail than we can cover here.

This ease of accessing an SQL database is not limited to reading data. You can now take this one step further and create a form that inserts data, for example. One of the wonderful things about PHP is that you do not have to deal with processing the data passed by forms. PHP does this for you.

As with the variables in the query string we discussed above (e.g., the *recipe_id*), form variables are immediately available on the page called by the action for the form. These can be combined to create an SQL query (using the *INSERT* command, of course), which is then passed to the *\$db->sql_query* function as we did above for the *SELECT*. You could take this one step further and use the *UPDATE* command to make changes to existing rows in your database.

Secure Database Access

At this point we need to sidetrack a little and talk about security when working with databases. You might think that if the only data you are working with is your recipes, then you don't have to worry much about security. Well, with an insecure system, someone could make changes to or delete your data entirely.

To make matters worse, if your own data is using the same database as the PHPNuke tables, a user that has access

PHP Functions

Here are some PHP functions that you should look at when developing a filter for input that you are sending into your database:

- addslashes
- stripslashes
- htmlentities
- htmlspecialchars
- striphtml

Details can be found in the PHP Manual at: <http://www.php.net/manual/en/>.

to your data has easier access to the PHPNuke data. Keep in mind that once the connection is made, the user may not need any extra authorization to change data in the PHPNuke tables once they have access to your tables.

Even if you separate your data from the PHPNuke tables, the same principles apply. You don't want people to have free reign of your data. There are a couple of different things you can do to increase security.

Most databases have the ability to restrict access, both to the database and to individual tables. You can create a user (within the database) that only has access to your data, but not the PHPNuke tables. If all the data was in a single table, you could restrict access so your user did not have access to the PHPNuke tables. However, this can get complicated and confusing very quickly. I think this is one more reason to split the data into different databases, despite the slight programming effort needed to access each.

When the data is split between different databases, you only need to worry about your own data. You make the connection to your own database and its database, and you let PHPNuke handle the PHPNuke tables and data. This makes it unlikely that you are going to give inadvertent access to a PHPNuke table.

However, and this is a *big* "however," the connection to the PHPNuke tables is still open when your files are loaded. Thus you need to make sure you are accessing your tables and not PHPNuke.

SQL Injection

Another thing we need to talk about is something called "SQL injection." This is a trick used by hackers to get an SQL query to do something more than just the original query. For example, if you had a module that displayed the name of a recipe input by the user, you might have a query to return all fields for the specific entry, which might look something like this:

```
select * from recipes where
    recipename = '$userinput';
```

If the user is clever enough, he or she can create a "recipename" that is actu-

ally an additional SQL query and which is then executed in the same context.

Therefore, you should never explicitly trust data that is passed to query. This is especially true if the values in the query are being read from a form and thus are input by the user, as in this example. Even if the user does not know how your query is created, it is still possible to input values to change the query.

The simplest way is to check the variable *\$userinput* for things that do not belong, such as quotes and even the word "union". You might go so far as to write a whole function that does more complex checking. This could include several different PHP functions that convert the input text to "safer" values before inserting it into your database. See the sidebar for details. Additional details about SQL injections and more information on PHPNuke security can be found in references [2] and [3].

In our example above, someone could *theoretically* pass the recipe id directly in the URL (i.e. *&\$recid = 42*) and manipulate the value in such a way as to create a new SQL query. In this case, it would be fairly simple to check to see if the value assigned to the *\$recid* variable were a number. If not, the program would simply not issue the query (and might generate an error message).

Making a House a Home

Being able to access websites in your native language is almost always an added reason to continue visiting a particular web site. The overwhelming majority of web sites are in English, which puts a lot of users at a disadvantage.

Most non-English-native computer users have at least a working knowledge of English and can move around fairly easily through the Internet. However, in general, sites exist either in English or in that web master's native language. With the exception of large commercial sites, multi-language sites are a rarity.

PHPNuke comes with built-in multi-language capabilities which can be configured through the administration panel and provide support for over 30 languages. In the PHPNuke directory, as well as most of the modules, you will find a *language* directory that contains a


number of files of the format *language.php*. These files contain the translation for much of the text that is displayed on each page and are of the form, *define("_CONSTANT","Translation");*. For example, to define what is displayed for the constant *_YES*, you might have:

```
define("_YES","Ano"); -Czech
define("_YES","Ja"); -German
define("_YES","Kyllä"); -Finnish
```

To include this functionality in your own module, add a *language* directory in your module along with files for the languages you want to include.

The best thing is to create a *complete* file for your default language (as defined in the administration panel) and then copy it. This ensures that all the terms you use are defined. If you leave out a particular term, you get an error when the page is loaded for that language, which obviously detracts from the usability of your site. ■

INFO	
[1]	See a PHPNuke system in action with several self-made modules and blocks: http://www.linux-tutorial.info
[2]	NukeCops FAQ on SQL Injection: http://www.nukecops.com/article74.html
[3]	Article on SQL Injection from the PHPNuke HOWTO: http://www.karakas-online.de/EN-Book/sql-injection-with-php-nuke.html
[4]	The PHPNuke Site: www.phpnuke.org
[5]	Home of a huge PHPNuke forum and many other resources: ftp://www.nukecops.com
[6]	As the name implies, this site has a wide range of fixes for various releases, including a forum: www.nukefixes.com
[7]	A wide range of resources for PHPNuke: www.nukeresources.com
[8]	Security related issues and fixes: www.nukesecurity.com

THE AUTHOR	<p>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.</p>	
------------	---	---