www.phparch.com

php | architect™

introducing Zend FRAMEWORK

propelling the open source community to new levels of innovation

PLUS

DEVELOPING A PLUGIN ARCHITECTURE FOR PHP

ASPECT ORIENTED SOFTWARE DEVELOPMENT

PRODUCT REVIEW: phpMyAdmin

"HARMLESS" FUN WITH SQL INJECTION

VOLUME 5 ISSUE 4

Thoughts from an Open Source Recruiter...
...with a PHP Twist

TEST PATTERN:
The Matter of Properties

Developing a Plugin Architecture for PHP Applications

If you want to see your software's functionality triple before your very eyes, it's a good idea to build support for plugins and extensions into your application. This article explains the intricate details of designing an extensible plugin architecture for any PHP application.

by TITUS BARIK

PHP: *5+*

CODE DIRECTORY: *phpplugin*

TO DISCUSS THIS ARTICLE VISIT:
http://forum.phparch.com/298

Generally speaking, a plugin is a module that adds a specific feature to a larger system. A plugin is named so, simply stated, because it plugs functionality into an existing system. In all likelihood, you have already encountered software plugins in your everyday work. GIMP, an image editor, offers a variety of plugins to modify graphics. XMMS, an audio player for Linux, supports plugins that enable the application to support additional audio formats. You may have encountered or even developed plugins for PHP applications, such as WordPress, Drupal, or PHP Nuke.

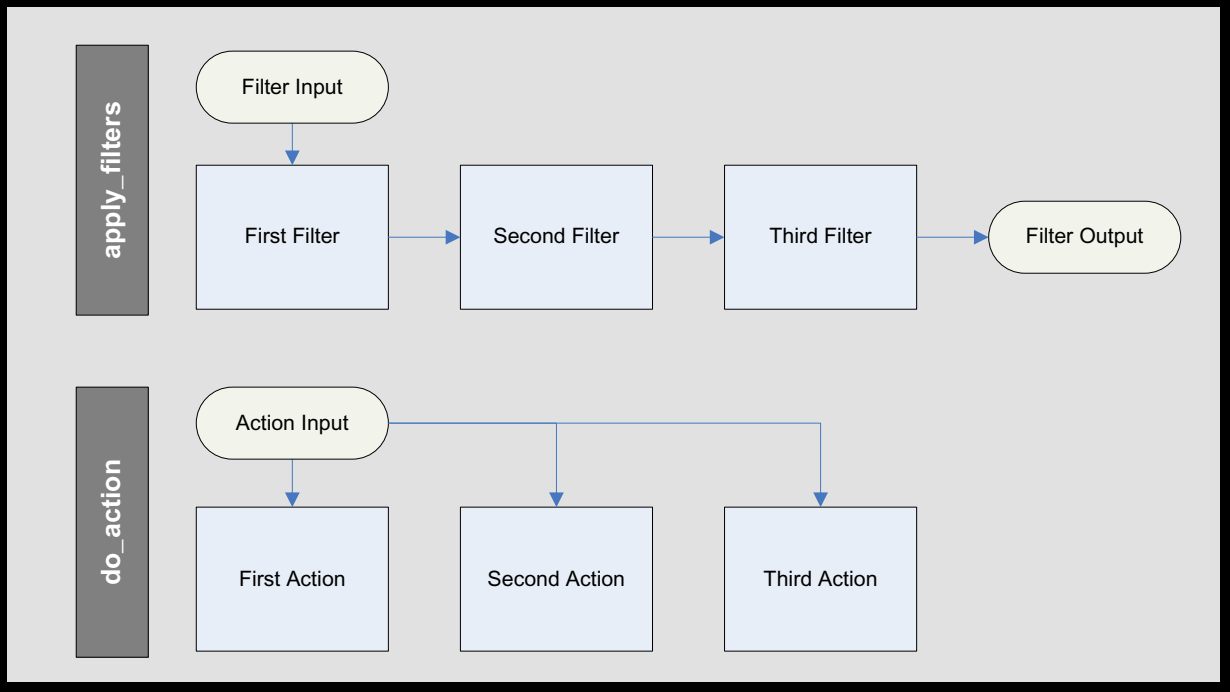
These days, support for plugins and extensions in PHP applications is widely considered to be an essential requirement of an application. One of the advantages of using a plugin system is that even after the initial development of the application has been completed, third-party developers can add or customize the functionality of the finished product without having to modify the application, itself. For open source applications, this may not seem like such a big deal, but for proprietary applications, a plugin architecture can increase the useful life of the application, long after its product maintenance cycle is terminated.

This article explains the intricate details of designing an extensible plugin architecture for a PHP application.

Actions, Filters, Modifiers, and More

Before I begin the development of our PHP plugin system, let's first examine the terminology that is most often encountered when developing plugins or modules. A large amount of confusion stems from the fact that

FIGURE 1



Support for plugins and extensions in PHP applications is widely considered to be an essential requirement.

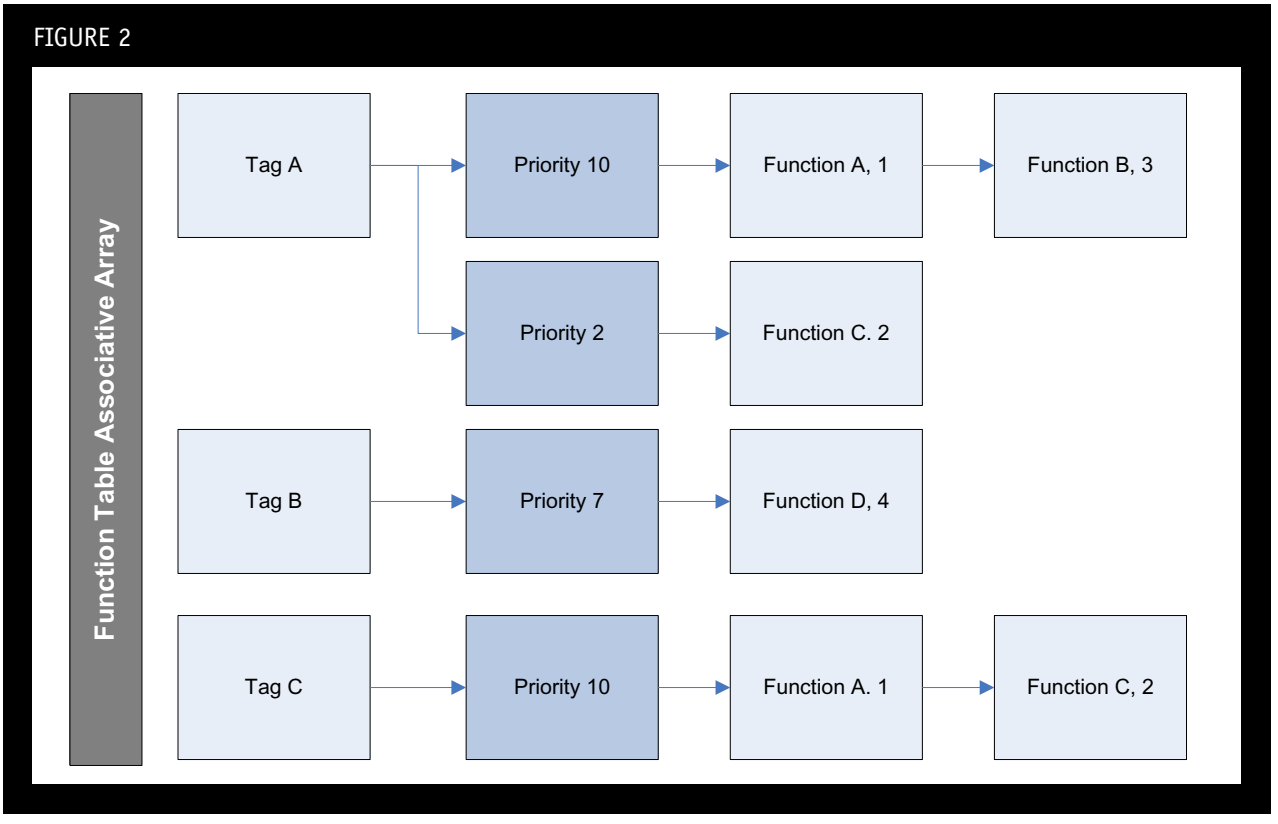
different plugin systems use different names for the same functionality. Thus, it's important, if for nothing else than consistency, to disambiguate all the terminology as it applies to this article.

Hook—A hook, or extension point, is a point in the flow of the execution of the program to which a plugin can latch its additional functionality. For example, WordPress provides a `delete_post` hook that occurs directly after a post has been deleted. A plugin that hooks to `delete_post` might log the time and date of all post deletions. Multiple plugins can latch to a hook. Similarly, the same plugin may latch to multiple hooks. The order in which the plugins are executed is dependent on the plugin system design.

Filter—A plugin that acts as a filter transforms content. A filter plugin takes an input and returns an output. This output may then be passed to the next filter plugin as an input, and so on, until all of the filters have processed the text. This sequence of events is similar to that of a filter chain, much like those found in Java Servlets. In the Smarty Template Engine, filters are called modifiers. Smarty modifiers include string tasks such as `capitalize`, `word wrap`, `truncate`, and `indent`. A developer might use one or more of these modifiers in a filter chain to transform a string.

Action—An action plugin works like a filter, with slight semantic differences. Unlike a filter, every plugin in the action chain uses the original input to the hook. Whereas filter plugins typically transform content, action plugins perform tasks that are triggered on a hook. The semantic difference between a filter and an action is illustrated in Figure 1.

FIGURE 2



Contract—A contract, commonly implemented as an interface, is a set of specifications that a plugin must follow in order to be loaded and executed successfully by the plugin system. A contract may specify certain methods that a class must have for an object-oriented system, or a specific naming convention to avoid namespace collisions. A contract may also specify the mechanism that determines the order in which plugins are loaded for the same hook, or version conflict resolution, just to name a few examples.

Plugin Magic with PHP

Regardless of how extravagant a plugin system may be, at its core, the system can only be built from a few fundamental functions available to PHP. Consequently, a plugin system incorporates one or more of the following elements: a `call_user_func()` family of functions, the `func_get_arg()` family of functions, `eval()`, and PHP's implicit ability to call functions through the use of standard string variables.

The simplest of these techniques is the use of standard variables. Its primary disadvantage is that it requires either a static number of arguments, or forces the called function to accept its parameters through a single array data structure, provided by the `func_get_args()` call. The example for `execute_dynamic()` clarifies:

```
function execute_dynamic($function_to_execute) {
  $data = func_get_args();
  $function_to_execute($data);
}
```

The `func_get_args()` call returns all of the arguments passed to `execute_dynamic()` as an indexed array. The name of the function to execute is placed as the first argument in the array, and any variable arguments ("varargs") that are passed to the function are placed in subsequent elements in that array. Here, it's easy to see that the burden of unwrapping the array arguments is placed on `function_to_execute()`. Thus, a developer that implements a plugin for your PHP application could only deal with functions that accept a single array parameter with all of its arguments. Despite these disadvantages, applications such as SquirrelMail use this technique to dynamically execute functions because of its potential performance benefits over alternative techniques.

Another common approach is the use of the PHP function, `call_user_function_array()`. This function accepts a function name as a string, and an array of arguments to pass to the specified function. The advantage of this technique is that the array of arguments is unwrapped automatically before being passed to the called function. A value of `NULL` can be passed to the second argument of `call_user_function_array()` if the

LISTING 1

```
define(ABSPATH, dirname(__FILE__) . '/');

$plugins = array('plugin1', 'plugin2', 'plugin3');

include_once('plugins.php');

foreach ($plugins as $plugin) {
    $plugin_file = ABSPATH . 'plugins/' . $plugin . '.php';
    if ($plugin != '' && file_exists($plugin_file)) {
        include_once($plugin_file);
    }
}
```

LISTING 2

```
function add_filter($tag, $function_to_add,
                                        $priority = 10, $accept_args = 1) {

    global $filter_table;

    if ( isset($filter_table[$tag][$priority]) ) {
        foreach($filter_table[$tag][$priority] as $filter) {
         if ( $filter['function'] == $function_to_add ) {
             return false;
            }
        }
    }

    $filter_table[$tag][$priority][] =
        array('function'=>$function_to_add,
                'accept_args'=>$accept_args);

    return true;

}
```

LISTING 3

```
function remove_filter($tag,
                    $function_to_remove, $priority = 10) {

    global $filter_table;
    $toret = false;

    if ( isset($filter_table[$tag][$priority]) ) {
        foreach($filter_table[$tag][$priority] as $filter) {
            if ( $filter['function'] != $function_to_remove ) {
                $new_function_list[] = $filter;
            }
            else {
                $toret = true;
            }
        }

        $filter_table[$tag][$priority] = $new_function_list;
    }
    return $toret;
}
```

LISTING 4

```
function do_action($tag, $arg = '') {
    global $filter_table;
    $extra_args = array_slice(func_get_args(), 2);


    $args = array_merge(array($arg), $extra_args);

    if ( !isset($filter_table[$tag]) ) {
        return;
    }
    else {
        ksort($filter_table[$tag]);
    }

    foreach ($filter_table[$tag] as $priority => $functions) {
        if ( !is_null($functions) ) {
            foreach($functions as $function) {

                $func_name = $function['function'];
                $accept_args = $function['accepted_args'];

                if ( $accept_args == 1 ) {
                    $the_args = array($arg);
                } elseif ( $accept_args > 1 ) {
                    $the_args = array_slice($args, 0, $accept_args);
                } elseif ( $accept_args == 0 ) {
                    $the_args = NULL;
                } else {
                    $the_args = $args;
                }

                $str = call_user_func_array($func_name, $the_args);
            }
        }
    }
}
```

LISTING 5

```
function apply_filters($tag, $string) {

    global $filter_table;

    $args = array_slice(func_get_args(), 2);

    if ( !isset($filter_table[$tag]) ) {
        return $string;
    }
    else {
        ksort($filter_table[$tag]);
    }

    foreach ($filter_table[$tag] as $priority => $functions) {

        if ( !is_null($functions) ) {
            foreach($functions as $function) {

                $all_args = array_merge(array($string), $args);
                $func_name = $function['function'];
                $accept_args = $function['accepted_args'];

                if ( $accept_args == 1 )
                    $the_args = array($string);
                elseif ( $accept_args > 1 )
                    $the_args = array_slice($all_args,
                                                        0, $accept_args);
                elseif ( $accept_args == 0 )
                    $the_args = NULL;
                else
                    $the_args = $all_args;

                $str = call_user_func_array($func_name, $the_args);

            }
        }
    }
    return $string;
}
```

called function requires no parameters. In such a case, the execute_dynamic() would look as follows:

```
function execute_dynamic($function_to_execute) {
   $data = func_get_args();
   call_user_function_array($function_to_execute, $data);
}
```

There are several variations and related functions for call_user_function_array(), including call_user_function(), func_num_args(), and function_exists(). Many of these functions can even be applied to methods within objects, perhaps for an object-based plugin system. However, such techniques are beyond the scope of this article.

In this article, I'll use the call_user_function_array() technique for implementing plugin callbacks. The plugin architecture that I'll present is a close cousin of the implementation provided by WordPress. In fact, many of the plugin routines are almost verbatim extractions from the WordPress source code, with small changes as warranted by the scope of this article.

Plugin Loader

I've discussed, on a high level, the approach to dynamically calling functions. But I've yet to mention how a plugin actually becomes available to the application. The responsibility of such a task rests on the plugin loader, whose job it is to find plugins stored on disk, and load them into memory for the currently-active PHP script. The code for such a task can vary from surprisingly simple to extremely complex. A simple approach for plugin loading is illustrated in Listing 1. The config.php file would be placed at the beginning of each script that needed plugin support.

The plugins array stores a list of active plugins that the plugin loader should enable. A loop iterates through the array, including the corresponding PHP file for each plugin through the use of include_once. Therefore, the complexity of a plugin loader lies in how it determines which plugins to load. In our case, we've specified the available plugins in a straight-forward array data structure. Other techniques might include pulling a list of active plugins from an SQL database, or a structured XML file. Developers may also wish to store the active plugins in a session object, or implement other performance enhancements, in order to minimize the overhead of loading plugins at the start of every page.

Filter Table

All of the information about the hooks and the associated plugin functions that bind to the application are stored in a global filter table. The filter table is a multidimensional array, whose first dimension is the name of the hook, also known as a tag. The second dimension is the priority, an integer value. The priority determines the order in which plugins are executed; a higher integer value indicates a lower priority. In this implementation, multiple functions can be bound to the same priority. In such a case, the order of plugin evaluation for that priority is determined by the order in which the function is first registered. The third and final dimension holds the actual information about the registered plugin functions. For every registered function, the function name and number of accepted arguments are stored. In this way, you can register multiple functions for a given priority and tag.

A contract may specify the order in which plugins are loaded.

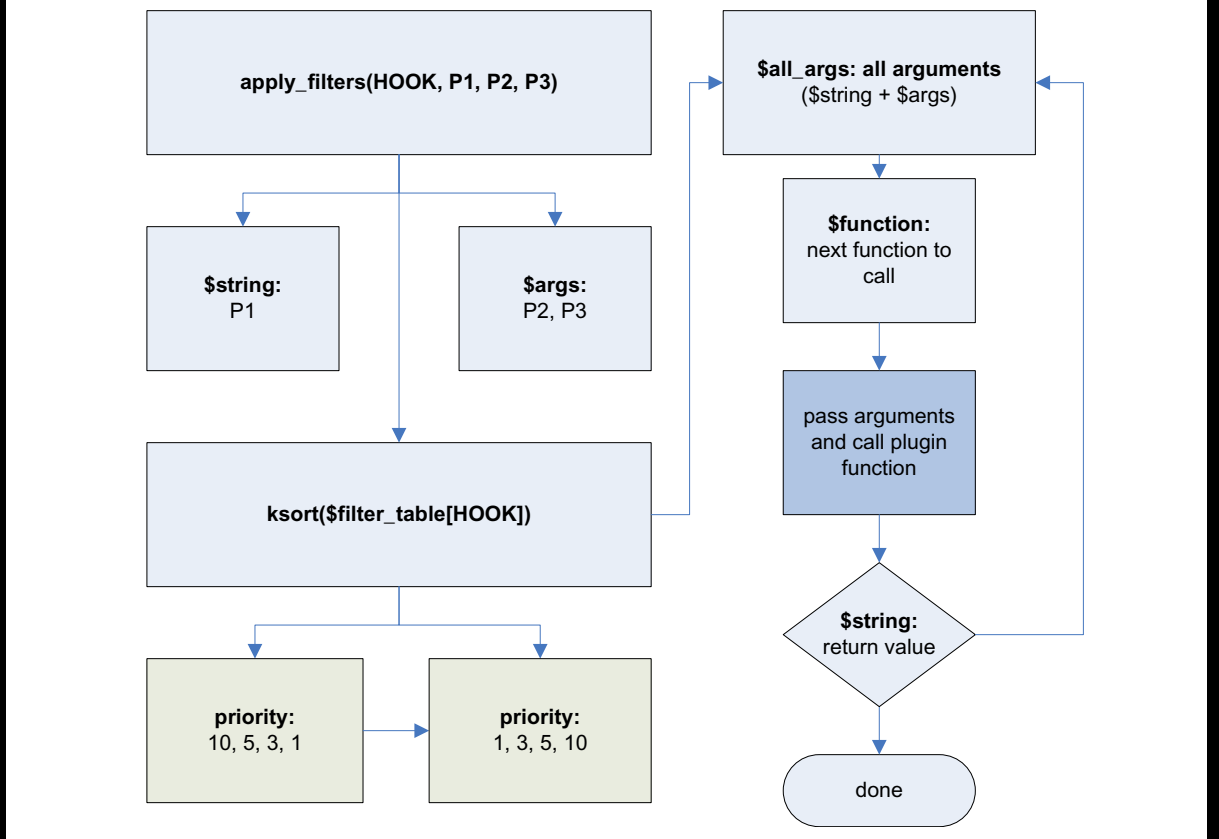
Instead of thinking of the filter table as a multi-dimensional array, it's easier if one imagines it to be a list of lists. Figure 2 depicts the filter table conceptually as nested lists.

Registering Hooks

The *hooks* implementation is an associative array that holds a list of tags (hook names), and a list of functions to execute for the tag, at a given priority. Hooks are the key component that allows plugins to attach at various points in the program execution flow. A proper hooks system provides the following capabilities:

- the ability of a plugin to attach to a hook (add_action, add_filter).
- the ability of a plugin to detach from a hook (remove_action, remove_filter).
- the ability to dynamically execute plugin code at a point in the program flow (do_action, apply_filters)

FIGURE 3



All of these functions can be placed in a single file. For this article, I use `plugins.php`. It is also the place where the largest portion of our code resides.

First, let's examine the `add_filter()` function, shown in Listing 2. It adds the desired plugin function to the filter table. It accepts as its arguments the name of the hook to bind to, the function to call for that hook, a priority that specifies the order in which a function is called when the hook is triggered, and the number of arguments that the plugin function can accept. In my implementation, a lower priority number has higher precedence. The `accepted_args` parameter is required because it allows me to determine how to properly call the function at a later point in time when using `call_user_func_array()`.

The `add_filter()` function also ensures that a plugin function is not accidentally added twice to a given hook. If this criteria is satisfied, the function is added to the filter table.

Recall that filters and actions have very similar semantics. As a result, `add_action()` uses the same table as filters. Indeed, `add_action()` simply chains to `add_filter()`:

```
function add_action($tag, $function_to_add,
  $priority = 10, $accepted_args = 1) {
    return add_filter($tag, $function_to_add,
    $priority, $accepted_args);
}
```

The subtle differences between filters and actions only present themselves when the hook is actually triggered. The implementation for executing such a hook is presented in the following section.

The `remove_filter()` and `remove_action()` functions are similar, and are more or less the inverse operation of `add_filter()` and `add_action()`. The code for `remove_filter()` is shown in Listing 3, and unsurprisingly, the `remove_action()`function simply calls `remove_filter()`:

```
function remove_action($tag, $function_to_remove,
  $priority = 10)
{
  return remove_filter($tag, $function_to_remove, $priority);
}
```

The remove_filter() function works by building a new list that contains every function for a given priority, except the one that the user wishes to remove. It then updates the filter table.

a proper array through the use of array_merge() and array_slice(). The call_user_function_array() function is then called, which in turn triggers the action or filter.

The developer should sprinkle do_action() and

Instead of thinking of the filter table as a multidimensional array, it's easier if one imagines it to be a list of lists.

The add_filter(), add_action(), remove_filter(), and remove_action() functions are placed at the beginning of a plugin file under the plugins directory. A plugin first attaches one or more of its functions to one of the many hooks within the application. A plugin then provides callback functions for the hook to execute when triggered.

Adding Hooks to an Application

All that's left now is to actually place hooks within the application. To do so, use the do_action(), and apply_filters() functions. They are shown in Listings 4 and 5, respectively.

As previously mentioned, a filter and an action are very similar. The key difference is that a filter chains its output as an input to the next filter in the chain, whereas an action always uses the original source argument. This results in only minor variations to the functions, and thus, I shall only discuss apply_filters(). Figure 3 demonstrates a graphical flowchart of this process to complement the presented code.

The function apply_filters() accepts a HOOK, and parameters P1, P2, and P3. The arguments are split between $string and $args. The ksort() function orders the elements in the array for precedence. All functions in the array are then executed in a filter chain, utilizing the $accepted_args parameter.]

The apply_filters() function begins by pulling all of its variable arguments into the args variable, with the exception of the explicit string parameter and the name of the tag. The ksort() function takes all of the priorities for a given tag, and resorts them in ascending order for execution. Next, for every function of a given tag, we determine the number of arguments it requires, and form

apply_filters() calls throughout the application. For example, in a shopping cart application, a developer might specify a hook after adding a product to a database:

```
insert_product_data($db, $product_data);
do_action('add_product_data', $product_data);
```

Don't be shy about adding hooks. The most common problem found in plugin-based application is that the third-party developer wants to extend a portion of the code, but doesn't have an appropriate extension point to hook into.

Other Plugin Functions

In a complex plugin environment, plugins may have dependencies on other plugins. More importantly, plugins that are dependent on one another may only work on specific versions of that plugin. Thus, a plugin may need to know whether another plugin is installed or activated:

```
function plugin_available($plugin) {
  global $plugins;
  return in_array($plugin, $plugins);
}
```

Alternatively, a plugin architecture may specify a contract that requires each plugin to provide a version number. This might be used in a situation where plugins have incompatibilities with specific versions of the plugins that they depend on:

```
function plugin_version($plugin) {
  $function = $plugin . '_version';
```

```
    echo $function;

    if (function_exists($function)) {
      return $function();
    }
    else {
      return NULL;
    }
  }
```

The plugin_version() function checks for a version function within a specified plugin. It simply concatenates the plugin name with the version number, separated with an underscore character, and returns the result if the function exists.

In this article, I've only provided the core functionality needed for a plugin architecture. But functions like plugin_version() can be used to enhance the capabilities of a plugin system, or to force plugin authors to follow a specific interface in their plugin design.

A Simple Sample Application

Though a plugin architecture need not be complicated, the various tie-in points and terminology may obscure the bigger picture. Thus, we conclude with an implementation of an artificial application that illustrates the features I've previously discussed.

Our sample application consists of four files: page.php, plugins.php, config.php, and datetitle.php. The datetitle.php file is a plugin, and should be placed in the plugins folder. The page.php file is a typical PHP page, and illustrates how one would add hooks to an application.

I start with the example page, page.php, which contains a hook in between the HTML <title> tags:

```
<?php include_once('config.php'); ?>

<html>
<head>
  <title>
  <?=apply_filters('title', 'Main Page');?>
  </title>
</head>

<body>
</body>
</html>
```

Next, the plugins.php file is modified to activate the datetitle plugin:

```
$plugins = array('datetitle');
```

Let's now take a look at datetitle.php, which should be placed in the plugins folder:

```
<?php
add_filter('title', date_append);

function date_append($title) {
  return date('l, M j') . ': ' . $title;
}

?>
```

The datetitle filter first attaches itself to the title hook using add_filter(). It specifies a callback function of date_append. As a result, page.php will first filter its text through date_append() before presenting the result to the user. When the page is loaded, the title of the HTML page will be prepended with the current date. If the filter is deactivated, then the HTML page displays the title "Main Page" without any modification.

Conclusion

It takes very little code to add plugin capabilities to a PHP application, but the benefits of doing so can be enormous. A plugin architecture can minimize headache in design changes after deployment, and simultaneously aid in the clarity and modularity of an application during its development life cycle. Even if you're not writing a custom application, the plugin architecture presented in this article is the cornerstone of many open source projects that are available today. An understanding of the techniques used in a plugin architecture may help you as a developer when working with other third-party applications that do support plugins. ■

TITUS BARIK is a content application developer with an interest in open source enterprise solutions. He has deployed both open source and proprietary content management systems successfully, in corporate and non-profit environments. His personal weblog is available at barik.net, and he welcomes your comments and suggestions.