SMASHING MAGAZINE

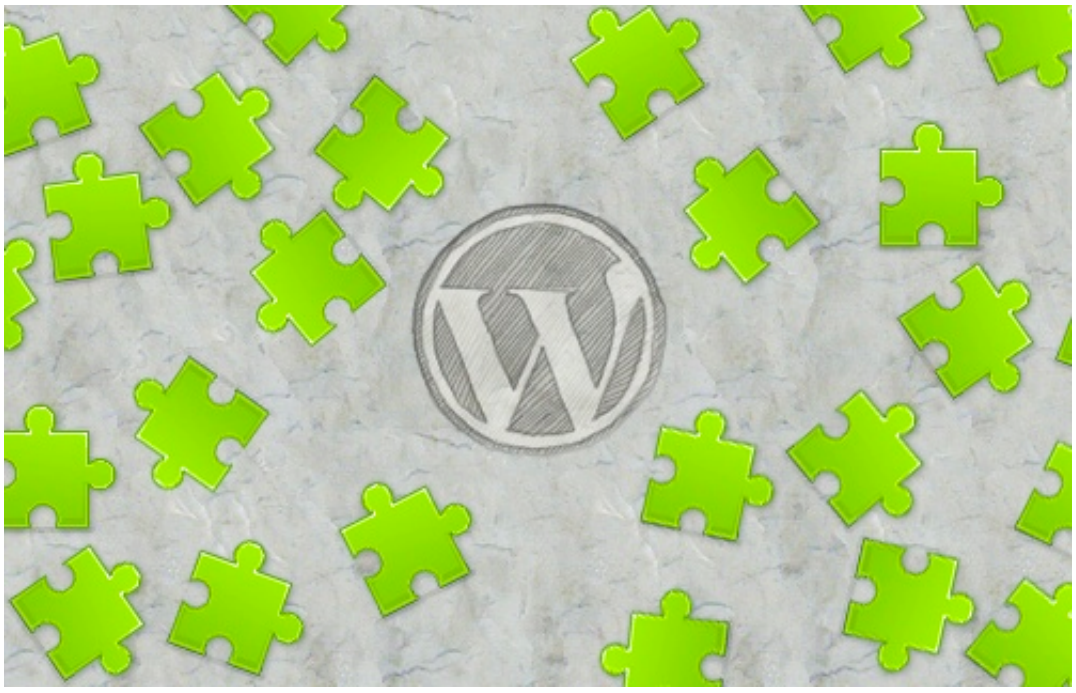
WordPress Essentials: How To Create A WordPress Plugin

By Daniel Pataki[1]

Published on September 30th, 2011 in Essentials[2], Plugins[3] with 46 Comments

WordPress plugins are PHP scripts that alter your website. The changes could be anything from the simplest tweak in the header to a more drastic makeover (such as changing how log-ins work, triggering emails to be sent, and much more).

Whereas themes modify the look of your website, plugins change how it functions. With plugins, you can create custom post types, add new tables to your database to track popular articles, automatically link your contents folder to a "CDN" server such as Amazon S3… you get the picture.



Theme Or Plugin?

If you've ever played around with a theme, you'll know it has a *functions.php* file, which gives you a lot of power and enables you to build plugin-like functionality into your theme. So, if we have this *functions.php* file, what's the point of a plugin? When should we use one, and when should we create our own?

The line here is blurrier than you might think, and the answer will often depend on your needs. If you just want to modify the default length of your posts' excerpts, you can safely do it in *functions.php*. If you want something that lets users message each other and become friends on your website, then a plugin would better suit your needs.

The main difference is that a plugin's functionality persists regardless of what theme you have enabled, whereas any changes you have made in *functions.php* will stop working once you switch themes. Also, grouping related functionality into a plugin is often more convenient than leaving a mass of code in *functions.php*.

Creating Our First PlugIn

To create a plugin, all you need to do is create a folder and then create a single file with one line of content. Navigate to the `wp-content/plugins` folder, and create a new folder named `awesomeplugin`. Inside this new folder, create a file named *awesomeplugin.php*. Open the file in a text editor, and paste the following information in it:

```
<?php
  /*
  Plugin Name: Awesomeness Creator
  Plugin URI: http://my-awesomeness-emporium.com
  Description: a plugin to create awesomeness and spread joy
  Version: 1.2
  Author: Mr. Awesome
  Author URI: http://mrtotallyawesome.com
  License: GPL2
  */
?>
```

Of all this information, only the plugin's name is required. But if you intend to distribute your plugin, you should add as much data as possible.

With that out of the way, you can go into the back end to activate your plugin. That's all there is to it! Of course, this plugin doesn't do anything; but strictly speaking, it is an active, functioning plugin.

Structuring PlugIns

When creating complex functionality, splitting your plugin into multiple files and folders might be easier. The choice is yours, but following a few good tips will make your life easier.

If your plugin focuses on one main class, put that class in the main plugin file, and add one or more separate files for other functionality. If your plugin enhances WordPress' back end with custom controls, you can create the usual CSS and JavaScript folders to store the appropriate files.

Generally, aim for a balance between layout structure, usability and minimalism. Split your plugin into multiple files as necessary, but don't go overboard. I find it useful to look at the structure of popular plugins such as WP-PageNavi and Akismet.

Naming Your PlugIn And Its Functions

When creating a plugin, exercise caution in naming the functions, classes and plugin itself. If your plugin is for generating awesome excerpts, then calling it "excerpts" and calling its main function "the_excerpt" might seem logical. But these names are far too generic and might clash with other plugins that have similar functionality with similar names.

The most common solution is to use unique prefixes. You could use "acme_excerpt," for example, or anything else that has a low likelihood of matching someone else's naming scheme.

Plugin Safety

If you plan to distribute your plugin, then security is of utmost importance, because now you are fiddling with other people's websites, not just your own. All of the security measures you should take merit their own article, so keep an eye out for an upcoming piece on how to secure your plugin. For now, let's just look at the theory in a nutshell; you can worry about implementation once you grasp that.

The safety of your plugin usually depends on the stability of its two legs. One leg makes sure that the plugin does not help spread naughty data. Guarding against this entails filtering the user's input, escaping queries to protect against SQL injection attacks and so on. The second leg makes sure that the user has the authority and intention to perform a given action. This basically means that only users with the authority to delete data (such as administrators) should be able to do it. Guarding intention ensures that visitors aren't misled by a hacker who has managed to place a malicious link on your website.

All of this is much easier to do than you might think, because WordPress gives you many functions to handle it. A number of other issues and best practices are involved, however, so we'll cover those in a future article. There is plenty to learn and do until then; if you're just starting out, don't worry about all that for now.

Cleaning Up After Yourself

Many plugins are guilty of leaving a lot of unnecessary data lying around. Data that only your plugin uses (such as meta data for posts or comments, database tables, etc.) can wind up as dead weight if the plugin doesn't clean up after itself.

WordPress offers three great hooks to help you take care of this:

- `register_activation_hook()` [4]
  This hook allows you to create a function that runs when your plugin is activated. It takes the path to your main plugin file as the first argument, and the function that you want to run as the second argument. You can use this to check the version of your plugin, do some upgrades between versions, check for the correct PHP version and so on.
- `register_deactivation_hook()` [5]
  The name says it all. This function works like its counterpart above, but it runs whenever your plugin is deactivated. I suggest using the next function when deleting data; use this one just for general housekeeping.
- `register_uninstall_hook()` [6]
  This function runs when the website administrator deletes your plugin in WordPress' back end. This is a great way to remove data that has been lying around, such as database tables, settings and what not. A drawback to this method is that the plugin needs to be able to run for it to work; so, if your plugin cannot uninstall in this way, you can create an *uninstall.php* file. Check out this function's documentation for more information.

If your plugin tracks the popularity of content, then deleting the tracked data when the user deletes the plugin might not be wise. In this case, at least point the user to the location in the back end where they can find the plugin's data, or give them the option to delete the data on the plugin's settings page before deleting the plugin itself.

The net result of all our effort is that a user should be able to install your plugin, use it for 10 years and then delete it without leaving a trace on the website, in the database or in the file structure.

Documentation And Coding Standards

If you are developing for a big community, then documenting your code is considered good manners (and good business). The conventions for this are fairly well established—phpDocumentor is one example. But as long as your code is clean and has some documentation, you should be fine.

I document code for my own benefit as well, because I barely remember what I did yesterday, much less the purpose of functions that I wrote months back. By documenting code, you force good practices on yourself. And if you start working on a team or if your code becomes popular, then documentation will be an inevitable part of your life, so you might as well start now.

While not quite as important as documentation, following coding standards is a good idea if you want your code to comply with WordPress' guidelines.

Putting It Into Practice

All work and no play makes Jack a dull boy, so let's do something with all of this knowledge that we've just acquired. To demonstrate, let's build a quick plugin that tracks the popularity of our articles by storing how many times each post has been viewed. I will be using hooks, which we'll cover in an upcoming installment in this series. Until then, as long as you grasp the logic behind them, all is well; you will understand hooks and plugins before long!

PLANNING AHEAD

Before writing any code, let's think ahead and try to determine the functions that our plugin will need. Here's what I've come up with:

- A function that registers a view every time an individual post is shown,
- A function that enables us to retrieve the raw number of views,
- A function that enables us to show the number of views to the user,
- A function that retrieves a list of posts based on their view count.

PREPARING OUR FUNCTION

The first step is to create the folder and file structure. Putting all of this into one file will be fine, so let's go to the `plugins` folder and create a new folder named `awesomely_popular`. In this folder, create a file named *awesomely_popular.php*. Open your new file, and paste some meta data at the top, something like this:

```
<?php
   /*
   Plugin Name: Awesomely Popular
   Plugin URI: http://awesomelypopularplugin.com
   Description: A plugin that records post views and contains functions to easily list posts by
 popularity
   Version: 1.0
   Author: Mr. Awesome
   Author URI: http://mayawesomefillyourbelly.com
   License: GPL2
   */
?>
```

RECORDING POST VIEWS

Without delving too deep, WordPress hooks enable you to (among other things) fire off one of your functions whenever another WordPress function runs. So, if we can find a function that runs whenever an individual post is viewed, we are all set; all we would need to do is write our own function that records the number of views and hook it in. Before we get to that, though, let's write the new function itself. Here is the code:

```
/**
 * Adds a view to the post being viewed
 *
 * Finds the current views of a post and adds one to it by updating
 * the postmeta. The meta key used is "awepop_views".
 *
 * @global object $post The post object
 * @return integer $new_views The number of views the post has
 *
 */
function awepop_add_view() {
   if(is_single()) {
      global $post;
      $current_views = get_post_meta($post->ID, "awepop_views", true);
      if(!isset($current_views) OR empty($current_views) OR !is_numeric($current_views) ) {
         $current_views = 0;
      }
      $new_views = $current_views + 1;
      update_post_meta($post->ID, "awepop_views", $new_views);
      return $new_views;
   }
}
```

As you can see, I have added phpDocumentor-style documentation to the top of the function, and this is a pretty good indication of what to expect from this convention. First of all, using a conditional tag, we determine whether the user is viewing a post on a dedicated page.

If the user is on a dedicated page, we pull in the `$post` object, which contains data about the post being shown (ID, title, posting date, comment count, etc.). We then retrieve the number of views that the post has already gotten. We will add `1` to this and then overwrite the orignal value with the new one. In case something goes wrong, we first check whether the current view count is what it should be.

The current view count must be set; it cannot be empty. And it must be numeric in order for us to be able to add `1` to it. If it does not meet these criteria, then we could safely bet that the current view count is `0`. Next, we add `1` to it and save it to the database. Finally, we return the number of views that the post has gotten, together with this latest number.

So far, so good. But this function is never called, so it won't actually be used. This is where hooks come in. You could go into your theme's files and call the function manually from there. But then you would lose that functionality if you ever changed the theme, thus defeating the entire purpose. A hook, named `wp_head`, that runs just before the `</head>` tag is present in most themes, so we can just set our function to run whenever `wp_head` runs, like so:

```
add_action("wp_head", "awepop_add_view");
```

That's all there is to the "mysticism" of hooks. We are basically saying, whenever `wp_head` runs, also execute the `awepop_add_view` function. You can place the code before or after the function itself.

RETRIEVING AND SHOWING THE VIEWS

In the function above, I already use the WordPress `get_post_meta()` function to retrieve the views, so writing a separate function for this might seem a bit redundant. In this case, it might well be redundant, but it promotes some object-oriented thinking, and it gives us greater flexibility when further developing the plugin. Let's take a look:

```
/**
 * Retrieve the number of views for a post
 *
 * Finds the current views for a post, returning 0 if there are none
 *
 * @global object $post The post object
 * @return integer $current_views The number of views the post has
 *
 */
function awepop_get_view_count() {
   global $post;
   $current_views = get_post_meta($post->ID, "awepop_views", true);
   if(!isset($current_views) OR empty($current_views) OR !is_numeric($current_views) ) {
      $current_views = 0;
   }

   return $current_views;
}
```

This is the same piece of code that we used in the `awepop_add_view()` function, so you could just use this to retrieve the view count there as well. This is handy, because if you decide to handle the `0` case differently, you only need to change it here. You will also be able to extend this easily and provide support for cases when we are not in the loop (i.e. when the `$post` object is not available).

So far, we have just retrieved the view count. Now, let's show it. You might be thinking this is daft—all we need is `echo awepop_get_view_count() . "views"` , no? That would certainly work, but what if there is only 1 view. In this case, we would need to use the singular "view." You might also want the freedom to add some leading text or some other tidbit, which would be difficult to do otherwise. So, to allow for these scenarios, let's write another simple function:

```
/**
 * Shows the number of views for a post
 *
 * Finds the current views of a post and displays it together with some optional text
 *
 * @global object $post The post object
 * @uses awepop_get_view_count()
 *
 * @param string $singular The singular term for the text
 * @param string $plural The plural term for the text
 * @param string $before Text to place before the counter
 *
 * @return string $views_text The views display
 *
 */
function awepop_show_views($singular = "view", $plural = "views", $before = "This post has: ")
{
   global $post;
   $current_views = awepop_get_view_count();

   $views_text = $before . $current_views . " ";

   if ($current_views == 1) {
      $views_text .= $singular;
   }
   else {
      $views_text .= $plural;
   }

   echo $views_text;

}
```

SHOW A LIST OF POSTS BASED ON VIEWS

To show a list of posts based on view count, we create a function that we can place anywhere in our theme. The function will use a custom query and loop through the results, displaying a simple list of titles. The code is below, and the explanation follows.

```
/**
 * Displays a list of posts ordered by popularity
 *
 * Shows a simple list of post titles ordered by their view count
 *
 * @param integer $post_count The number of posts to show
 *
 */
function awepop_popularity_list($post_count = 10) {
   $args = array(
       "posts_per_page" => 10,
       "post_type" => "post",
       "post_status" => "publish",
       "meta_key" => "awepop_views",
       "orderby" => "meta_value_num",
       "order" => "DESC"
   );

   $awepop_list = new WP_Query($args);

   if($awepop_list->have_posts()) { echo ""; }

   while ( $awepop_list->have_posts() ) : $awepop_list->the_post();
       echo ''.the_title('', '', false).'';
   endwhile;

   if($awepop_list->have_posts()) { echo "";}
}
```

We start by passing a bunch of parameters to the `WP_Query` class, in order to create a new object that contains some posts. This class will do the heavy lifting for us: it finds 10 published posts that have `awepop_views` in the meta table, and orders them according to this property in descending order.

If posts meet this criterion, we create an unordered list element. Then, we loop through all of the posts that we have retrieved, showing each title as a link to the relevant post. We cap things off by adding a closing tag to the list when there are posts to show. Placing the `awepop_popularity_list()` function anywhere in your theme should now generate a simple list of posts ordered by popularity.

As an added precaution, place the call to this function in your theme, like so:

```
if (function_exists("awepop_popularity_list")) {
   awepop_popularity_list();
}
```

This ensures that if the plugin is disabled (and thus the function becomes undefined), PHP won't throw a big ol' error. It just won't show the list of most popular posts.

Overview

By following the theory laid down in this article and using only a handful of functions, we have created a rudimentary plugin to track our most popular posts. It could be vastly improved, but it shows the basics of using plugins perfectly well. Moreover, by learning some patterns of WordPress development (plugins, hooks, etc.), you are gaining skills that will serve you in non-WordPress environments as well.

You should now be able to confidently follow tutorials that start with “First, create a WordPress plugin…” You now understand things not just on a need-to-know basis, but at a deeper level, which gives you more flexibility and power when coding your own plugins. Stay tuned for the upcoming article on hooks, actions and filters for an even more in-depth resource on the innards of plugins.

(al)

FOOTNOTES:

[0] Daniel Pataki - http://wp.smashingmagazine.com/author/daniel-pataki/?rel=author

[1] Essentials - http://wp.smashingmagazine.com/tag/essentials/

[2] Plugins - http://wp.smashingmagazine.com/tag/plugins/

[3] register_activation_hook() - http://codex.wordpress.org/Function_Reference/register_activation_hook

[4] register_deactivation_hook() - http://codex.wordpress.org/Function_Reference/register_deactivation_hook

[5] register_uninstall_hook() - http://codex.wordpress.org/Function_Reference/register_uninstall_hook

I am a web developer in love with WordPress. I have a company called Bonsai Shed where we make webapps and premium themes like Musico and Estatement. You can follow me on Twitter or take a look at my personal site, Danielpataki.com